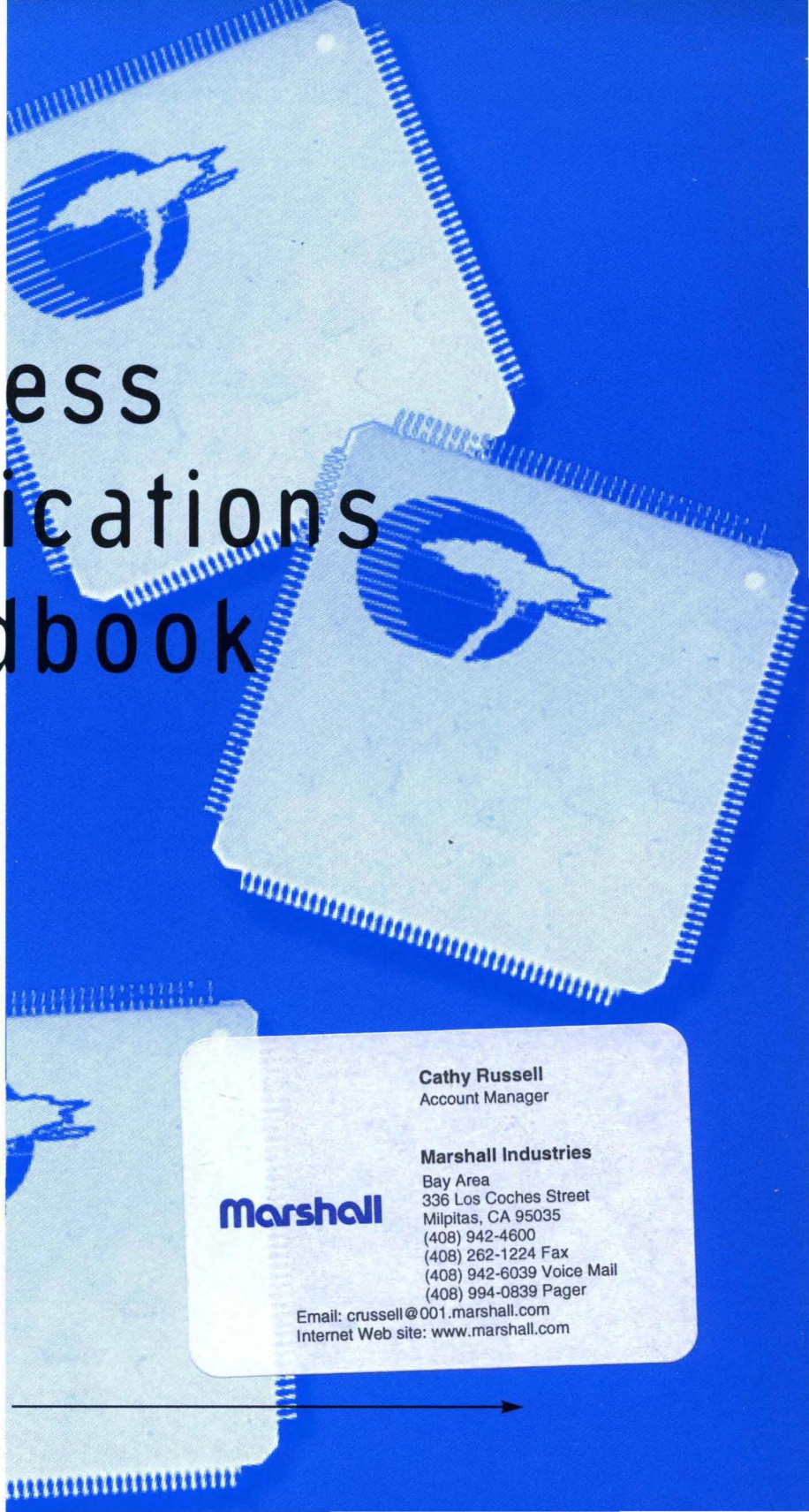
Cypress Applications Handbook

1996

CYPRESS

Cypress Applications Handbook

CYPRESS



Cypress Applications Handbook

Cypress Semiconductor is a trademark of Cypress Semiconductor Corporation.
Cypress Semiconductor, 3901 North First St., San Jose, CA 95134 (408) 943–2600
Telex: 821032 CYPRESS SNJ UD, TWX: 910 997 0753, FAX: (408) 943–2741
FAX-On-Demand: 1-800-213-5120 or 1-408-943-2798, Web Address: http://www.cypress.com

How to Use This Book

This *Applications Handbook* is a learning tool for using Cypress devices. The application notes included here range from general product overview articles, such as "Understanding Dual-Port RAMs," to specific design examples. To summarize each application note, an abstract listing has been provided at the front of each section.

The general overviews describe product-family characteristics and explain some of the products-capabilities. These application notes appear at the beginning of this *Handbook.*

Next appear application examples that show how to use specific Cypress devices in the context of real designs. The application examples are organized by product type (e.g., PROMs or CPLDs). Within each product type examples are arranged by product number, using the product that is the article's primary focus.

Although your specific application might not appear explicitly in an application note, the design examples can still be useful to you. If the design example is similar to your application, you might be able to adapt the hardware or software to your design easily. Many of the application notes provide PLD software code for design tools from a variety of vendors, so that you can copy the code and use it as a skeleton for your own PLD designs. Even if none of the examples relate directly to your design, they can stimulate new ideas by showing features or applications that might not have occurred to you. The information can also significantly reduce the learning curve normally associated with unfamiliar ICs.

Published January 1996

Contents

General Information

SRAMs

PROMs/EPROMs

UltraLogic/PLDs

UltraLogic/PLDs (continued)

Specialty Memories

Data Communications

Data Communications (continued)

Timing Products

Bus Products

General Information – 1

General Information Section Contents and Abstracts

This application note describes factors to consider when designing a digital system using high-performance CMOS integrated circuits. A formula is derived that enables the designer to predict when a trace on a PCB may become a transmission line. A simplified transmission line analysis is presented that eliminates the $j\omega t$ phase terms from the classical transmission line equations. Step function responses and pulse responses are tabulated for various line terminations. Various types of transmission lines and types of terminations are presented and analyzed. An analysis of an unterminated line is performed to illustrate the procedure.

This application note explains how to protect your CMOS circuits using an inexpensive zener diode. It also explains how to calculate the value of the decoupling capacitor for your integrated circuits and why the decoupling capacitor does not function well as a filtering capacitor. A capacitor impedance versus frequency curve is presented that shows how capacitor size is related to its series resonance frequency. The Fourier Transform of a periodic pulse is presented in order to show how high-frequency noise is generated.

This application note shows how to properly decouple a circuit from its power supply. The decoupling consists of a combination of a large decoupling capacitor and a smaller, high-frequency filtering capacitor. Design and board layout guidelines are given with specific reference to Cypress's HOTLink transmitter and receiver.

System Design Considerations When Using Cypress CMOS Circuits

This application note describes some factors to consider when either designing new systems using Cypress high-performance CMOS integrated circuits or when using Cypress products to replace bipolar or NMOS circuits in existing systems. The two major areas of concern are device input sensitivity and transmission line effects due to impedance mismatching between the source and load.

To achieve maximum performance when using Cypress CMOS ICs, pay attention to the placement of the components on the printed circuit board (PCB); the routing of the metal traces that interconnect the components; the layout and decoupling of the power distribution system on the PCB; and perhaps most important of all, the impedance matching of some traces between the source and the loads. The latter traces must, under certain conditions, be analyzed as transmission lines. The most critical traces are those of clocks, write strobes on SRAMs and FIFOs, output enables, and chip enables.

Replacing Bipolar or NMOS ICs

Cypress CMOS ICs are designed to replace both bipolar ICs and NMOS products and to achieve equal or better performance at one-third (or less) the power of the components they replace.

When high-performance Cypress CMOS circuits replace either bipolar or NMOS circuits in existing sockets, be aware of conditions in the existing system that could cause the Cypress ICs to behave in unexpected ways. These conditions fall into two general categories: device input sensitivity and sensitivity to reflected voltages.

Input Sensitivity

High-performance products, by definition, require less energy at their inputs to change state than low- or medium-performance products.

Unlike a bipolar transistor, which is a current-sensing device, a MOS transistor is a voltage-sensing device. In fact, a MOS circuit design parameter called K′ is analogous to the gm of a vacuum tube and is inversely proportional to the gate oxide thickness.

Thin gate oxides, which are required to achieve the desired performance, result in highly sensitive inputs. These inputs require very little energy at or above the device input-voltage threshold (approximately 1.5V at 25°C) to be detected. CMOS products may detect high-frequency signals to which bipolar devices may not respond.

MOS transistors also have extremely high input impedances (5 to 10 MΩ), which make the transistors' gate inputs analogous to the input of a high-gain amplifier or an RF antenna. In contrast, because bipolar ICs have input impedances of 1000Ω or less, these devices require much more energy to change state than do MOS ICs. In fact, a typical Cypress IC requires less that 10 picojoules of energy to change state. Thus, when Cypress CMOS ICs replace bipolar or NMOS ICs in existing systems, the CMOS ICs might respond to pulses of energy in the system that are not detected by the bipolar or NMOS products.

Reflected Voltages

Cypress CMOS ICs have very high input impedances and—to achieve TTL compatibility and drive capacitive loads—low output impedances. The im-

pedance mismatch due to low-impedance outputs driving high-impedance inputs might cause unwanted voltage reflections and ringing under certain conditions. This behavior could result in less-than-optimum system operation.

When the impedance mismatch is very large, a nearly equal and opposite negative pulse reflects back from the load to the source when the line's electrical length (PCB trace) is greater than

$$l = \frac{t_r}{2t_{pd}} \qquad \text{Eq. 1}$$

where t_r is the rise time of the signal at the source, and t_{pd} is the one-way propagation delay of the line per unit length.

The classical way of stating the condition for a voltage reflection to occur is that it will occur if the signal rise time is less than or equal to the round-trip (two-way) propagation delay of the line.

Input clamping diodes to ground were added to bipolar IC families (e.g., TTL, AS, LS, ALS, FAST) when the circuit designers decided that the fast rise and fall times of the outputs could cause voltage reflections. The clamping diodes to V_{CC} are inherent in the junction isolation process. For a more detailed explanation, see "Input/Output Characteristics of Cypress Products."

Historically, as circuit performance improved, the output rise and fall times of the bipolar circuits decreased to the point where voltage reflections began to occur (even for short traces) when an impedance mismatch existed between the line and the load. Most users, however, were unaware of these reflections because they were suppressed by the diodes' clamping action.

Conventional CMOS processing results in PN junction diodes, which adversely affect the ESD (electrostatic discharge) protection circuitry at each input pin and cause an increased susceptibility to latch-up. In addition, when the input pin is negative enough to forward bias the input clamping diodes, electrons are injected into the substrate. When a sufficient number of electrons are injected, the resulting current can disturb internal nodes, causing soft errors at the system level.

To eliminate the prospect of having this problem, all Cypress CMOS products use a substrate bias generator. The substrate is maintained at a negative 3V potential, so the substrate diodes cannot be forward biased unless the voltage at the input pin becomes a diode drop more negative than −3V. (See *Figure 9* in "Input/Output Characteristics of Cypress Products" for a schematic of the input protection circuit used in all Cypress CMOS products.) To the systems designer, this translates to approximately five times (3.8V divided by 0.8V =4.75) the negative undershoot safety margin for Cypress CMOS integrated circuits versus those that do not use a bias generator.

Voltage reflections should be eliminated by using impedance matching techniques and passive components that dissipate excess energy before it can cause soft errors. Crosstalk should be reduced to acceptable levels by careful PCB layout and attention to details.

Crosstalk

The rise and fall times of the waveforms generated by Cypress CMOS circuit outputs are 2 to 4 ns between levels of 0.4 and 4V. The fast transition times and the large voltage swings could cause capacitive and inductive coupling (crosstalk) between signals if insufficient attention is paid to PCB layout.

Crosstalk is reduced by avoiding running PCB traces parallel to each other. If this is not possible, run ground traces between signal traces.

In synchronous systems, the worst time for the crosstalk to occur is during the clock edge that samples the data. In most systems it is sufficient to isolate the clock, chip select, output enable, and write and read control lines from each other and from data and address lines so that the signals do not cause coupling to each other or to the data lines.

It is standard practice to use ground or power planes between signal layers on multilayered PCBs to reduce crosstalk. The capacitance of these isolation planes increases the propagation delay of the signals on the signal layers, but this drawback is more than compensated for by the isolation the planes provide.

The Theory of Transmission Lines

A connection (trace) on a PCB should be considered as a transmission line if the wavelength of the applied frequency is short compared to the line length. If the wavelength of the applied frequency is long compared to the length of the line, conventional circuit analysis can be used.

In practice, transmission lines on PCBs are designed to be as nearly lossless as possible. This simplifies the mathematics required for their analysis, compared to a lossy (resistive) line.

Ideally, all signals between ICs travel over constant-impedance transmission lines that are terminated in their characteristic impedances at the load. In practice, this ideal situation is seldom achieved for a variety of reasons.

Perhaps the most basic reason is that the characteristic impedances of all real transmission lines are not constants, but present different impedances depending upon the frequency of the applied signal. For "classical" transmission lines driven by a single frequency signal source, the characteristic impedance is "more constant" than when the transmission line is driven by a square wave or a pulse.

According to Fourier series expansion, a square wave consists of an infinite set of discrete frequency components—the fundamental plus odd harmonics of decreasing amplitude. When the square wave propagates down a transmission line, the higher frequencies are attenuated more than the lower frequencies. Due to dispersion, the different frequencies do not travel at the same speed.

Dispersion indicates the dependence of phase velocity upon the applied frequency (see Reference 1 pg. 192). The result is that the square wave or pulse is distorted when the frequency components are added together at the load.

A second reason why practical transmission lines are not ideal is that they frequently have multiple loads. The loads may be distributed along the line at regular or irregular intervals or lumped together, as close as practical, at the end of the line. The signal-line reflections and ringing caused by impedance mismatches, non-uniform transmission line impedances, inductive leads, and non-ideal resistors could compromise the dynamic system noise margins and cause inadvertent switching.

One system design objective is to analyze the critical signal paths and design the interconnections such that adequate system noise margins are maintained. There will always be signal overshoot and undershoot. The objective is to accurately predict these effects, determine acceptable limits, and keep the undershoot and overshoot within the limits.

The Ideal Transmission Line

An equivalent circuit for a transmission line appears in *Figure 1*. The circuit consists of subsections of series resistance (R) and inductance (L) and parallel capacitance (C) and shunt admittance (G) or parallel resistance, R_p. For clarity and consistency, these parameters are defined per unit length. Multiply

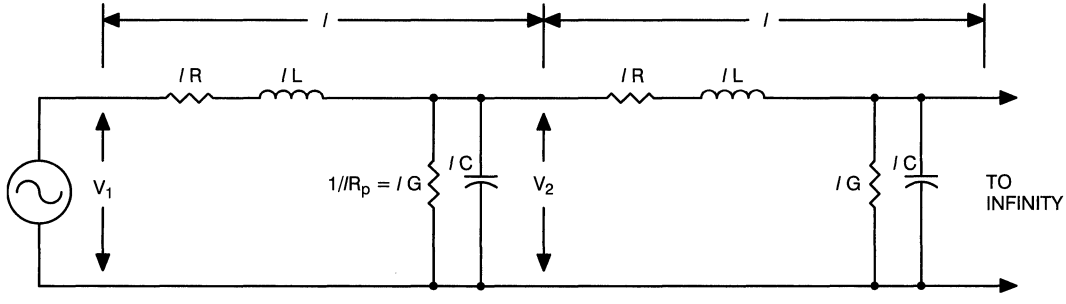


Figure 1. Transmission Line Model

the values of R, L, C, and R_p by the length of the subsection, *l*, to find the total value. The line is assumed to be infinitely long.

If the line of *Figure 1* is assumed to be lossless (R = 0, R_p = infinity), *Figure 1* reduces to *Figure 2*. A small series resistance has little effect upon the line's characteristic impedance. In practice and by design, the series resistance is quite small. For 1-ounce (0.0015-inch-thick), 1-mil-wide (0.010-inch) copper traces on G-10 glass epoxy PCBs, the trace resistance is between 0.5 and 0.3Ω per foot. 2-ounce copper has a resistance 50 percent lower than that of 1-ounce copper.

Input or Characteristic Impedance

To calculate the characteristic impedance (also called AC impedance or surge impedance) looking into terminals a-b of the circuit in *Figure 2*, use the following procedure.

Let Z_1 be the input impedance looking into terminals a-b,with Z_2 for terminals c-d, Z_3 for terminals e-f, etc. Z_1 is the series impedance of the first inductor (*l*L) in series with the parallel combination of Z_2 and the impedance of the capacitor (*l*C).

From AC theory:

$$X_L = j\omega lC \qquad \text{Eq. 2}$$

where X_L is the inductive reactance.

$$X_C + \frac{1}{j\omega lC} \qquad \text{Eq. 3}$$

where X_C is the capacitive reactance.

Then

$$Z_1 = X_L + \frac{Z_2 X_C}{Z_2 + X_C} \qquad \text{Eq. 4}$$

If the line is reasonably long, $Z_1 = Z_2 = Z_3$. Substituting $Z_1 = Z_2$ into *Equation 4* yields

$$Z_1 = X_L + \frac{Z_1 X_C}{Z_1 + X_C}$$

or

$$Z_1^2 - Z_1 X_L - X_C X_L = 0 \qquad \text{Eq. 5}$$

Substituting the expressions for X_C and X_L yields

$$Z_1^2 - j\omega lL = \frac{L}{C} \qquad \text{Eq. 6}$$

Equation 6 contains a complex component that is frequency dependent. The complex component can be eliminated by allowing *l* to become very small and by recognizing that the ratio L/C is constant and independent of *l* or ω:

$$Z_1 = \sqrt{L/C} \qquad \text{Eq. 7}$$

The AC input impedance of a purely reactive, uniform, lossless line is a resistance. This is true for AC or DC excitation.

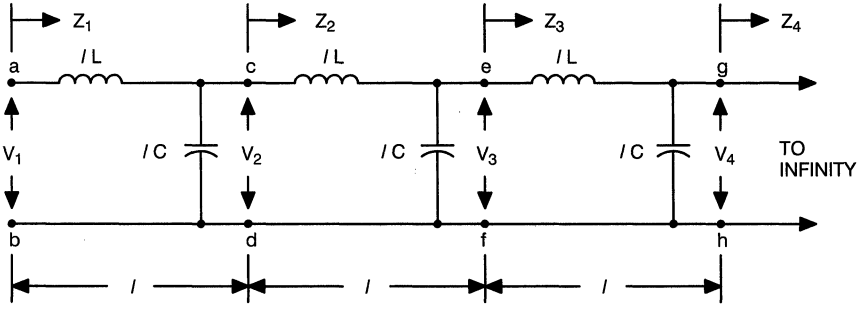


Figure 2. Ideal Transmission Line Model

Propagation Velocity and Delay

The propagation velocity (or phase velocity) of a sinusoid traveling on an ideal line (see Reference 1) is

$$\alpha = \frac{1}{\sqrt{LC}} \qquad \text{Eq. 8}$$

The propagation delay for a lossless line is the reciprocal of the propagation velocity:

$$t_{pd} = \sqrt{LC} = Z_1 C \qquad \text{Eq. 9}$$

where L and C are once again the intrinsic line inductance and capacitance per unit length.

Adding additional stubs or loads to the line (see Reference 2 of this application note) increases the propagation delay by the factor

$$\sqrt{1 + C_D/C} \qquad \text{Eq. 10}$$

where C_D is the load capacitance.

Therefore, the propagation delay of a loaded line, T_{pdL}, is

$$t_{pdL} = t_{pd}\sqrt{1 + C_D/C} \qquad \text{Eq. 11}$$

This application note shows later that a transmission line's unloaded or intrinsic propagation delay is proportional to the square root of the dielectric constant of the medium surrounding or adjacent to the line. Propagation delay is not a function of the line's geometry.

The characteristic impedance of a capacitively loaded line decreases by the same factor that the propagation delay increases:

$$Z_1' = \frac{Z_1}{\sqrt{1 + C_D/C}} \qquad \text{Eq. 12}$$

Note that the capacitance per unit length must be multiplied by the line length, *l*, to calculate an equivalent lumped capacitance.

The Condition for Voltage Reflection

It is relatively straightforward to obtain a closed-form solution for a transmission line's maximum allowable length, which, if exceeded, might cause a voltage reflection. If the line is not terminated in its characteristic impedance, a reflection is guaranteed to occur. The reflection's amplitude depends on the amount of impedance mismatch between the line and the load and whether the rise time of the signal at the source equals or is greater (slower) than two times the propagation delay of the line.

The condition for a voltage reflection to occur is

$$L \geq \frac{t_r}{2t_{pdL}} \qquad \text{Eq. 13}$$

Solving for the loaded propagation delay yields

$$t_{pdL} = \frac{t_r}{2L} \qquad \text{Eq. 14}$$

However, the actual physical length of the line is

$$l = \frac{t_r}{t_{pd}} \qquad \text{Eq. 15}$$

The intrinsic capacitance of the line from *Equation 9* is

$$C_O = \frac{T_{pd}}{Z_O} \qquad \text{Eq. 16}$$

It is standard practice to use C_O to designate the intrinsic line capacitance, L_O the intrinsic line self inductance, and Z_O the intrinsic line characteristic impedance.

Substituting *Equations 14*, *15*, and *16* into *Equation 11* gives the relationship for the line length at which voltage reflections might occur. Two conditions must be present for voltage reflections to occur: the line must be long and there must be an impedance mismatch between the line and the load.

$$\frac{t_r}{2L} = t_{pd}\sqrt{1 + \frac{C_D}{\frac{t_r}{t_{pd}} \times \frac{t_{pd}}{Z_O}}} \qquad \text{Eq. 17}$$

Solving Equation 17 for the line length, L, yields

$$L = \frac{t_r}{2t_{pd}} \times \frac{1}{\sqrt{1 + \frac{C_D Z_O}{t_r}}} \qquad \text{Eq. 18}$$

Equation 18 is very useful to the system designer. It is generic and applies to all products irrespective of circuit type, logic family, or voltage levels. The equation allows you to estimate when a line requires termination, using variables you can easily determine.

When driving a distributed or non-lumped load, the signal's rise time depends on the source—not the load, as you might expect. The intrinsic, or unloaded, line propagation delay per unit length is a function of the dielectric constant and can be easily calculated. The intrinsic line characteristic impedance is a function of the dielectric constant and the PCB's physical construction or geometry and can also be calculated. Finally, you can estimate the equivalent (lumped) load capacitance by adding up the number of loads (device inputs) being driven and multiplying by 10 pF. For I/O pins, use 15 pF per pin.

Signal Transition Times

The standard Cypress 0.8μ (L drawn) CMOS process yields output buffers whose signals transition approximately 4V in 2 ns, or, have a slew rate of 2V per nanosecond. The rise time/fall time is 2 ns. Products fabricated using the Cypress BiCMOS process have the same rise times.

The Cypress ECL process yields products with 500-ps output signal rise times and fall times, or slew rates of 1V/0.5 ns = 2V per nanosecond. Internal signal slew rates are 10V per nanosecond, but only for short (usually less than 500 mV) voltage excursions. Thus, high-frequency noise is generated on chip, which you can eliminate by using 100- to 500-pF ceramic or mica filter capacitors between V_{CC} and ground.

The values in *Table 1* come from using *Equation 18* to calculate the line length at which voltage reflections may occur. The calculations assume a 50Ω intrinsic line characteristic impedance and that the PCB is multilayer, using stripline construction on G-10 glass epoxy material (dielectric constant of 5). These conditions result in an unloaded line propagation delay of 2.27 ns per foot.

Table 1. Line Length at Which a Voltage Reflection Occurs

t_r (ns)	C_D (pF)	L (inches)
2	10	4.73
2	20	4.32
2	40	3.74
2	80	3.05
1	10	2.16
1	20	1.87
1	40	1.53
1	80	1.18
0.5	10	0.93
0.5	20	0.76
0.5	40	0.59
0.5	80	0.44

Table 1 reveals that decreasing the source rise time from 2 to 0.5 ns (a factor of 4) decreases the line length at which a voltage reflection might occur by a factor of 5 (4.73 divided by 0.93 = 5.09) for the same load (10 pF) and intrinsic propagation delay (2.27 ns/ft.). A second observation is that for signals with rise times of 0.5 ns, all lines should be terminated.

Reflection Coefficients

Another attribute of the ideal transmission line, reflection coefficients, are not actually line characteristics. The line is treated as a circuit component, and reflection coefficients are defined that measure the impedance mismatches between the line and its source and the line and its load. The reason for defining and presenting the reflection coefficients becomes apparent later when it is shown that if the impedance mismatch is sufficiently large, either a negative or positive voltage might reflect back from the load to the source, and the voltage might either add to or subtract from the original signal. A mismatch between the source and line impedance may also cause a voltage reflection, which in turn reflects back to the load. Therefore, two reflection coefficients are defined.

For classical transmission lines driven by a single frequency source, the impedance mismatches cause standing waves. When pulses are transmitted and the source's output impedance changes depending upon whether a LOW-to-HIGH or a HIGH-to-LOW transition occurs, the analysis is complicated further.

You can use classical transmission line analysis—where pulses are represented by complex variables with exponentials—to calculate the voltages at the source and the load after several back and forth reflections. However, these complex equations tend to obscure what is physically happening.

Energy Considerations

Now consider the effects of driving the ideal transmission line with digital pulses and analyze the behavior of the line under various driving and loading conditions. The first task is to define the load and source reflection coefficients.

Figure 3 shows the circuit to be analyzed. The ideal transmission line of length l is driven by a digital source of internal resistance R_S and loaded with a resistive load R_L. The characteristic impedance of the line appears as a pure resistance,

$$Z_o = \sqrt{L/C} \qquad \text{Eq. 19}$$

to any excitation.

The ideal case is when $R_S = Z_O = R_L$. The maximum energy transfer from source to load occurs under this condition, and no reflections occur. Half the energy is dissipated in the source resistance, R_S, and the other half is dissipated in the load resistance, R_L (the line is lossless).

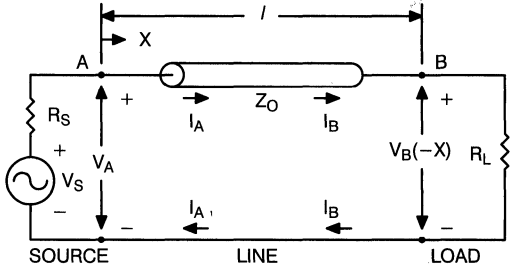


Figure 3. Ideal Transmission Line Loaded and Driven

If the load resistor is larger than the line's characteristic impedance, extra energy is available at the load and is reflected back to the source. This is called the underdamped condition, because the load underuses the energy available. If the load resistor is smaller than the line impedance, the load attempts to dissipate more energy than is available. Because this is not possible, a reflection occurs that signals the source to send more energy. This is called the overdamped condition. Both the underdamped and overdamped cases cause negative traveling waves, which cause standing waves if the excitation is sinusoidal. The condition $Z_O = R_L$ is called critically damped.

The safest termination condition, from a systems design viewpoint, is the slightly overdamped condition, because no energy is reflected back to the source.

Line Voltage for a Step Function

To determine the line voltage for a step function excitation, you apply a step function to the ideal line and analyze the behavior of the line under various loading conditions. The step function response is important because any pulse can be represented by the superposition of a positive step function and a negative step function, delayed in time with respect to each other. By proper superposition, you can predict the response of any line and load to any width pulse. The principle of superposition applies to all linear systems.

According to theory, the rise time of the signal driven by the source is not affected by the characteristics of the line. This has been substantiated in practice by using a special coaxially constructed reed relay that delivers a pulse of 18A into 50Ω with a rise time of 0.070 ns (see Reference 1).

The equation representing the voltage waveform going down the line (see *Figure 3*) as a function of distance and time is

$$V_L(X,t) = V_A(t)U(t - Xt_{pd}) \text{ for } t < T_O \qquad \text{Eq. 20}$$

$$V_A(t) = V_S(t)\left(\frac{Z_O}{Z_O + R_S}\right) \quad \text{Eq. 21}$$

where

V_A = the voltage at point A

X = the voltage at a point X on the line

l = the total line length

t_{pd} = the propagation delay of the line in nanoseconds per foot

$T_O = l\, t_{pd}$, or the one-way line propagation delay

U(t) = a unit step function occurring at x = 0

$V_S(t)$ = the source voltage

When the incident voltage reaches the end of the line, a reflected voltage, V′, occurs if R_L does not equal Z_O. The reflection coefficient at the load, ϱL, can be obtained by applying Ohm's Law.

The voltage at the load is $V_L + V_L'$, which must be equal to $(I_L + I_L')R_L$. But

$$I_L = \frac{V_L}{Z_O} \quad \text{Eq. 22}$$

and

$$I_L' = -\frac{V_L'}{Z_O} \quad \text{Eq. 23}$$

(The minus sign is due to I_L being negative; i.e., I_L is opposite to the current due to V_L.) Therefore,

$$V_B = V_L + V_L' = \left(\frac{V_L}{Z_O} - \frac{V_L'}{Z_O}\right)R_L \quad \text{Eq. 24}$$

By definition:

$$\rho_L = \frac{\textit{reflected voltage}}{\textit{incident voltage}} = \frac{V_L'}{V_L} \quad \text{Eq. 25}$$

Solving for V_L'/V_L in *Equation 24* and substituting in the equation for ϱ_L yields

$$\rho_L = \frac{R_L - Z_O}{R_L + Z_O} \quad \text{Eq. 26}$$

The reflection coefficient at the source is

$$\rho_S = \frac{R_S - Z_O}{R_S + Z_O} \quad \text{Eq. 27}$$

Re-arranging *Equation 24* yields

$$V_B = V_L + V_L' = \left(1 + \frac{V_L'}{V_L}\right)V_L$$

$$= (1 + \rho_L)V_L \quad \text{Eq. 28}$$

Equation 28 describes the voltage at the load (V_B) as the sum of an incident voltage (V_L) and a reflected voltage (ϱ_L V*l*) at time t = T_O. When $R_L = Z_O$, no voltage is reflected. When $R_L < Z_O$, the reflection coefficient at the load is negative; thus, the reflected voltage subtracts from the incident voltage, giving the load voltage. When $R_L > Z_O$, the reflection coefficient is positive; thus, the reflected voltage adds to the incident voltage, again giving the load voltage.

Note that the reflected voltage at the load has been defined as positive when traveling toward the source. This means that the corresponding current is negative, subtracting from the current driven by the source.

This piecewise analysis is cumbersome and can be tedious. However, it does provide an insight into what is physically happening and demonstrates that a complex problem can be solved by dividing it into a series of simpler problems. Also, eliminating the exponentials—which provide phase information in the classical transmission line equations—simplifies the mathematics. To use the piecewise method, you must do careful bookkeeping to combine the reflections at the proper time. This is quite straightforward, because a pulse travels with a constant velocity along an ideal or low-loss line, and the time delay between reflected pulses can be predicted.

The rules to keep in mind are that at any location and time the voltage or the current is the algebraic sum of the waves traveling in both directions. For example, two voltage waves of the same polarity and equal amplitudes, traveling in opposite directions, at a given location and time add together to yield a voltage of twice the amplitude of one wave. The same reasoning applies to all points of termination and discontinuities on the line. The total voltage or current is the algebraic sum of all the incident and reflected waves. Polarities must be observed. A

positive voltage reflection results in a negative current reflection and vice versa.

Step Function Response of the Ideal Line

Before examining reflections at the source due to mismatches between the source and line impedances, consider the behavior of the ideal line with various loads when driven by a step function. The circuit for analysis appears in *Figure 3*. *Figure 4* shows the voltage and current waveforms at point A (line input) and point B (the load) for various loads. (These values are drawn from Reference 1 pg. 158 – 159.) Note that $R_S = Z_O$ and that V_A at t = 0 equals $V_S/2$. This means that no impedance mismatch exists between the source and the line; thus, there is no reflection from the source at $t = 2\ T_O$. T_O is the one-way propagation delay of the line.

The time-domain response of the reactive loads are obtained by applying a step function to the LaPlace transform of the load and then taking the inverse transform.

Note that the reflection coefficient at the load is not the total reflection coefficient (a complex number) but represents only the real part of the load. The piecewise method eliminates the complex ($j\omega t$) terms by performing the bookkeeping involving the phase relationships, which the complex terms account for in classical transmission line analysis.

Note that for the open-circuit condition in *Figure 4b*, Z_L = infinity, so that $\varrho_L = +1$. The voltage is reflected from the load to the source (at amplitude $V_O = V_S/2$). Thus, at time $t = 2\ T_O$, the reflected voltage adds to the original voltage, $V_O = V_S/2$, to give a value of $2V_O = V_S$. While the voltage wave is traveling down to and back from the load, a current of

$$I_o = \frac{V_o}{Z_o} = \frac{V_s}{2} Z_o \qquad \text{Eq. 29}$$

exists. This current charges up the distributed line capacitance to the value V_S, then the current stops.

The waveforms at the source and load for the series RC termination shown in *Figure 4g* are of particular interest because this network dissipates no DC power; you can use this network to terminate a transmission line in its characteristic impedance at the input to a Cypress IC. *Figure 4h* represents the equivalent circuit of a Cypress IC's input. Combining both networks models a Cypress IC driven by a transmission line terminated in the line's characteristic impedance, when the values of R and C are properly chosen.

Reflections Due to Discontinuities

Figure 5 illustrates three types of common discontinuities found on transmission lines. Any change in the characteristic impedance of the line due to construction, connectors, loads, etc., causes a discontinuity, which causes a reflection that directs some energy back to the source. The amount of energy reflected back is determined by the discontinuity's reflection coefficient. Because discontinuities are usually small by design, most of the energy is transmitted to the load.

In general, a discontinuity has series inductance, shunt capacitance, and series resistance. An example is a via from a signal plane through a ground plane to a second signal plane in a multilayer PCB or module. IC sockets and other connectors can also cause discontinuities.

The Ideal Transmission Line's Pulse Response

Consider next the behavior of the ideal transmission line when driven by a pulse whose width is short compared to the line's electrical length—when the pulse width is less than the line's one-way propagation delay time, T_O.

Figure 6 shows another series of response waveforms for the circuit in *Figure 3*, this time for a pulse instead of a step (drawn from Reference 1 pg. 160 – 161). Note that $R_S = Z_O$ and that V_A at t = 0 equals $V_S/2$. This means that there is no impedance mismatch between the source and the line; thus, there is no reflection from the source at $t = 2\ T_O$.

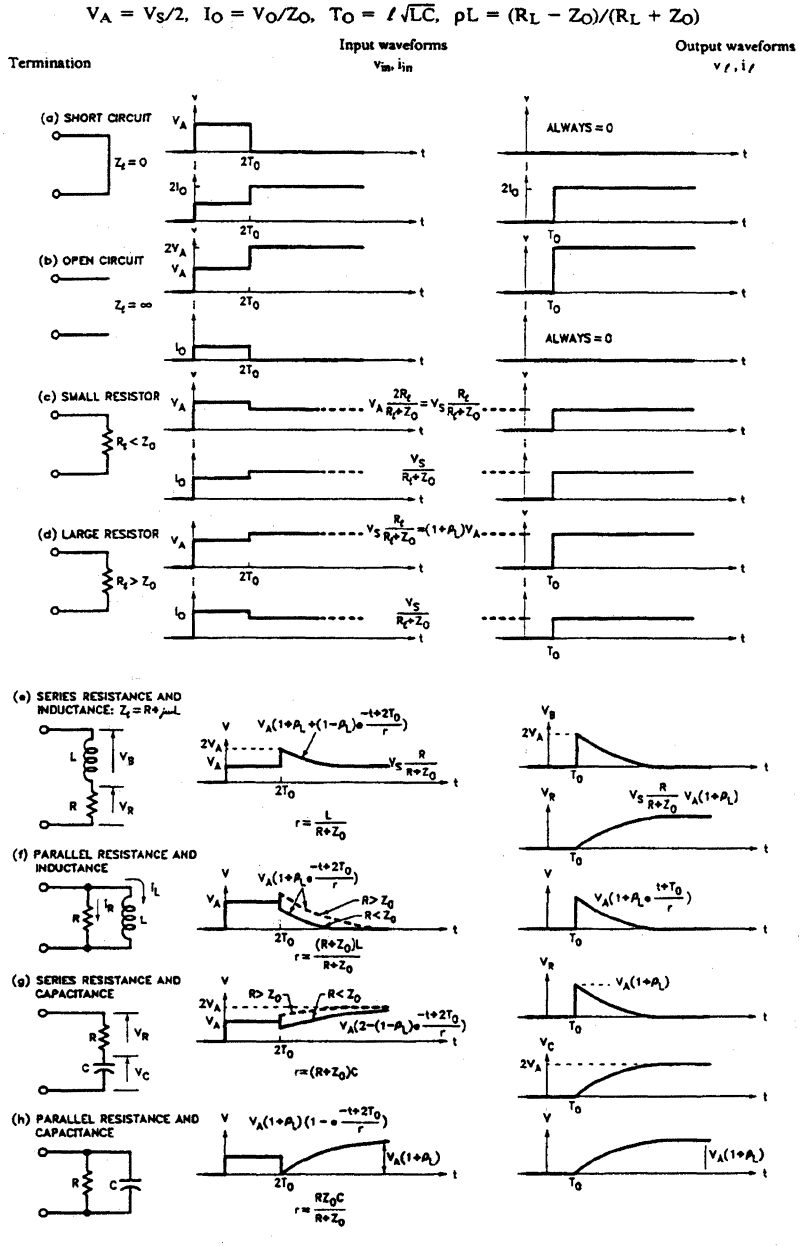


Figure 4. Step Function Response of Figure 3 for Various Terminations

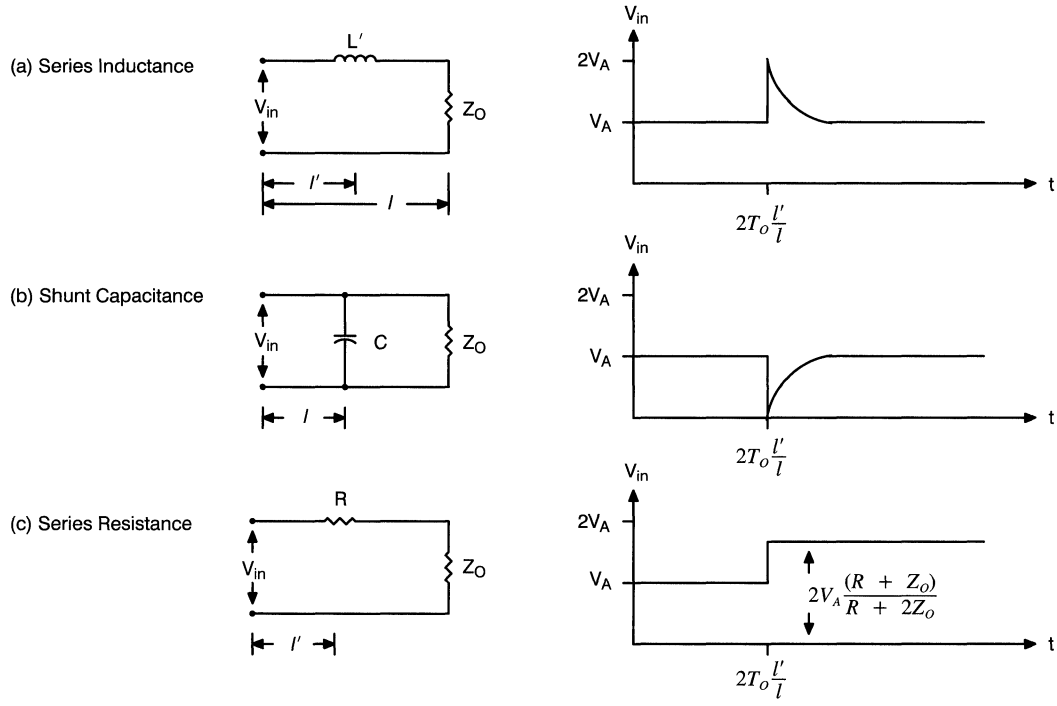


Figure 5. Reflections from Discontinuities with an Applied Step Function

Finite Rise Time Effects

Now consider the effects of step functions with finite rise times driving the ideal transmission line. During the rise time of a pulse, half the energy in the static electric field is converted into a traveling magnetic field and half remains as a static electric field to charge the line.

If the rise time is sufficiently short, the voltage at the load changes in discrete steps. The amplitude of the steps depends on the impedance mismatch, and the width of the steps depends on the line's two-way propagation delay.

As the rise time and/or the line gets shorter (smaller T_O), the result converges to the familiar RC time constant, where C is the static capacitance. All devices should be treated as transmission lines for transient analysis when an ideal step function is applied. However, as the rise time becomes longer and/or the traces shorter, the transmission line analysis reduces to conventional AC circuit analysis.

Reflections from Small Discontinuities

Figure 7 shows a pulse with a linear rise time and rounded edges driving the transmission line of *Figure 5a* and *Figure 5b*. The expressions for V_r are derived on pages 171 and 172 of Reference 1. The reflection caused by the small series inductance is useful for calculating the value of the inductor, L′, but little else.

The reflection caused by the small shunt capacitor is more interesting. If this capacitor is sufficiently large, it can cause a device connected to the transmission line to see a logic 0 instead of a logic 1.

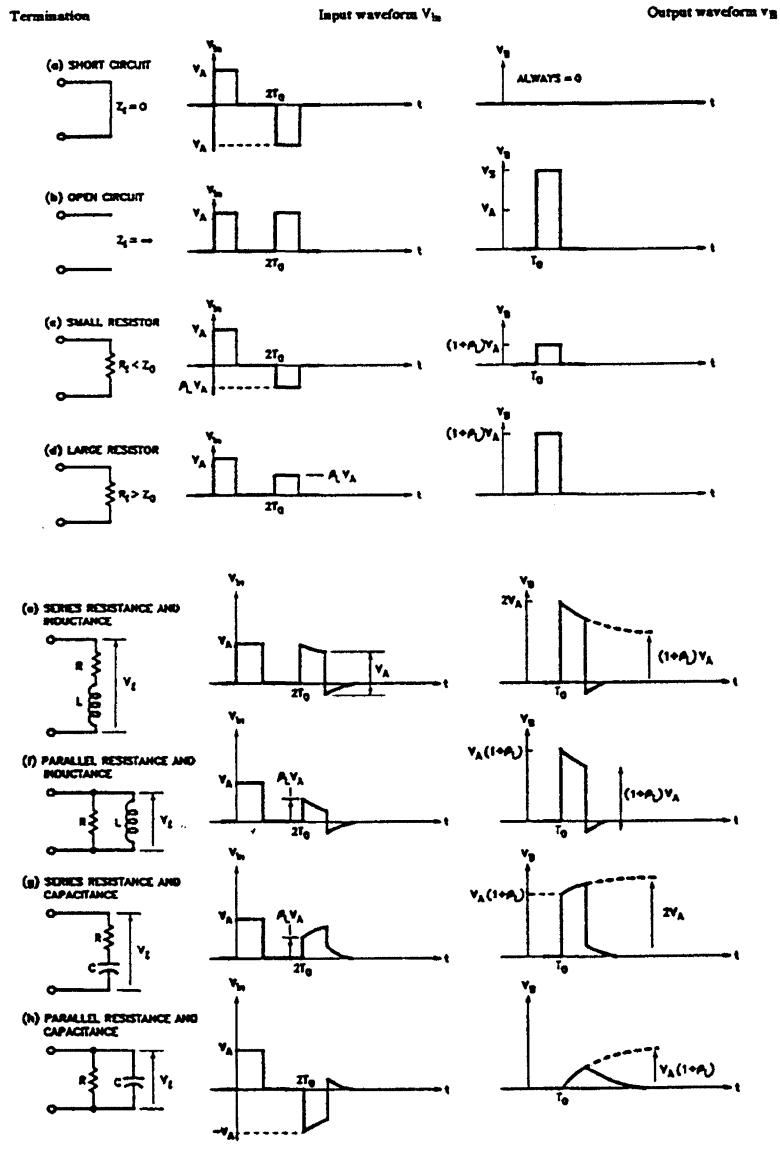


Figure 6. Pulse Response of Figure 3 for Various Terminations

$$V_A = V_S/2,\quad I_O = V_O/Z_o,\quad T_o = l\sqrt{LC},\quad \rho_L = \frac{(R_L - Z_o)}{(R_L + Z_o)}$$

The Effect of Rise Time on Waveforms

Next, consider the ideal line terminated in a resistance less than its characteristic impedance and driven by a step function with a linear rise time. The stimulus, the circuit, and the response appear in *Figure 8a*, *Figure 8b*, and *Figure 8c*, respectively. Once again, note that because the source resistance equals the line characteristic impedance, there are no reflections from the source.

The resulting waveforms are similar to those of *Figure 4c* when modified as shown in *Figure 8c*. The final value of the waveform must be the same as before (*Figure 4c*).

The resultant wave at the line input (V_{in}) is easily obtained by superposition of the applied wave and the reflected wave at the proper time. In *Figure 8*, because the step function's rise time is less than the line's two-way propagation delay, the input wave reaches its final value, $V_S/2$. At $t = 2\ T_O$, the reflected wave arrives back at the source and subtracts from the applied step function (the load reflection coefficient is negative). *Figure 9* illustrates waveforms for two relationships between the step function rise time and the propagation delay.

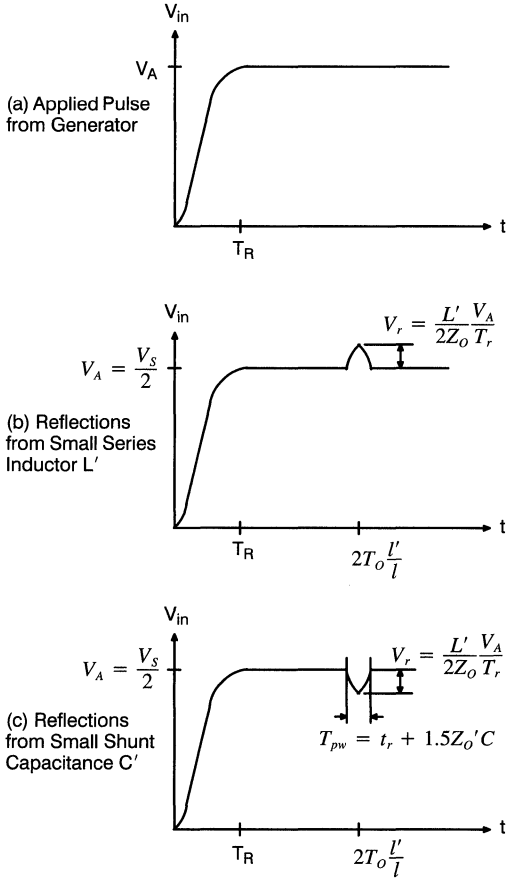


Figure 7. Reflections from Small Discontinuities with a Finite Rise Time Pulse

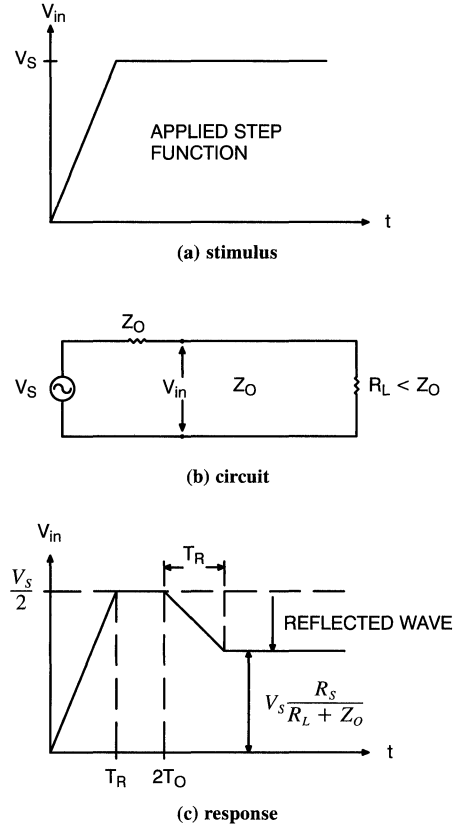


Figure 8. Effect of Rise Time on Response of Mismatched Line with $R_L < Z_O$

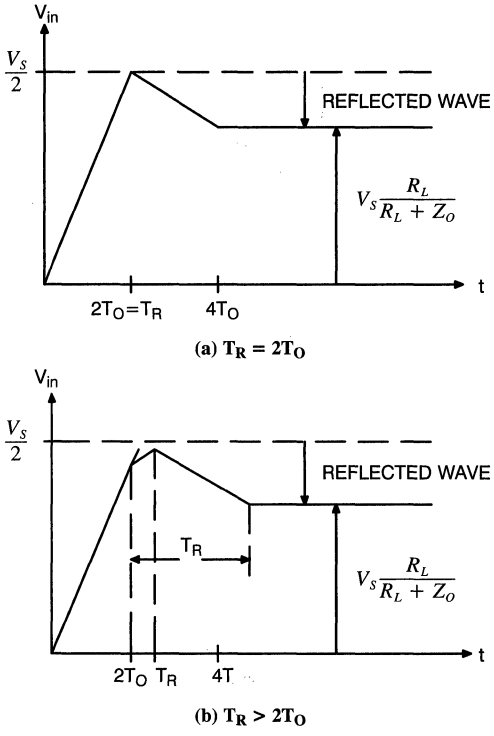


Figure 9. Effects of Rise Time on Response for $R_L < Z_O$

Multiple Reflections

Now consider the case of an ideal transmission line with multiple reflections caused by improper terminations at both ends of the line. The circuit and waveforms appear in *Figure 10*. The reflection coefficients at the source and the load are both negative—the source resistance and the load resistance are both less than the line characteristic impedance.

When the switch is initially closed, a step function of amplitude

$$V_O = V_{in} = \frac{V_S Z_O}{R_S + Z_O} \qquad \text{Eq. 30}$$

appears on the line and travels toward the load. After a one-way propagation delay time, T_O, the wave reflects back with an amplitude of $\varrho_L V_O$.

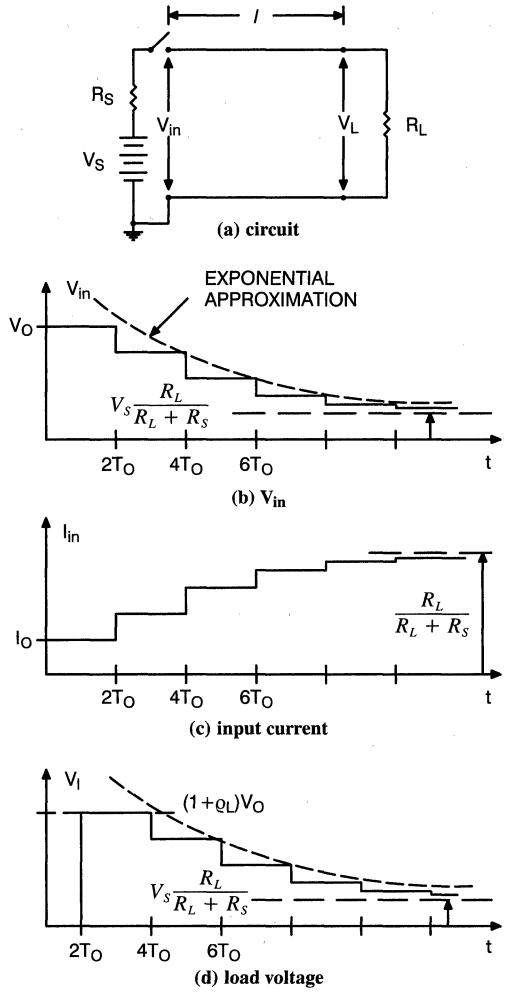


Figure 10. Step Function Applied to Line Mismatched on Both Ends; Shown for Negative Values of ϱ_S and ϱ_L

This first reflected wave than travels back to the source, and at time t = 2 T_O, the wave reaches the input end of the line. At this time, the first reflection at the source occurs, and a wave of amplitude ϱ_S (ϱ_L V_O) reflects back to the load. At time t = 3 T_O, this wave again reflects from the load back to the source with amplitude

$$\rho_L\rho_S\,(\rho_L V_O) = \rho_S\rho_L{}^2 V_O \qquad \text{Eq. 31}$$

This back and forth reflection process continues until the amplitudes of the reflections become so small that they cannot be observed. The circuit is then said to be in a quiescent state.

Effective Time Constant

Voltage reflections in small increments and of short durations approximate an exponential function, as indicated by the dashed line in *Figure 10b*. The smaller and narrower the steps become, the more closely the waveform approaches an exponential curve.

The mathematical derivation is presented on pages 178 and 179 of Reference 1. The time constant is

$$K = -\frac{2T_O}{1 - \rho_S\rho_L} \qquad \text{Eq. 32}$$

Thus, the resultant voltage waveform at the load can be approximated by

$$V(t) = V_O e\left(\frac{t}{K}\right) \qquad \text{Eq. 33}$$

For *Equation 32* to be accurate, ϱ_L and ϱ_S must be reasonably large (approaching ±1) so that the incremental steps are small. Because the product $\varrho_S\varrho_L$ is a positive number, less than one, the time constant is a negative number, which indicates that the exponential decreases with time. This is usually the case in transient circuits.

Both reflection coefficients must also have the same sign to yield a continually decreasing or increasing waveform. Opposite signs give oscillatory behavior that cannot be represented by an exponential function.

From Transmission Line to Circuit Analysis

When a transmission line is terminated in its characteristic impedance, the line behaves like a resistor. It usually does not matter if you use transmission line or circuit analysis, provided that you take the propagation delays into account.

Consider the case of a short-circuited transmission line driven by a step function with a source impedance unequal to the characteristic line impedance. The general case is shown in *Figure 10a*. For $R_L = 0$ the reflection coefficients are

$$\rho_S = \frac{Z_S - Z_O}{Z_S + Z_O} \qquad \rho_L = -1 \qquad \text{Eq. 34}$$

The approximate time constant is

$$-k = \frac{2T_O}{1 - \rho_S\rho_L} = \frac{2T_O}{1 + \rho_S} = \frac{T_O(Z_S + Z_C}{Z_S}$$

or

$$-k = T_O + \frac{T_O Z_O}{Z_S} \qquad \text{Eq. 35}$$

Recall that

$$T_O = l\sqrt{LC} \qquad \text{Eq. 36}$$

(one-way delay) and

$$Z_O = \sqrt{L/C} \qquad \text{Eq. 37}$$

where l is the physical length of the line, and L and C are the per-unit-length parameters. Substituting these variables into *Equation 35* yields

$$-k = T_O + l\frac{L}{Z_S} \qquad \text{Eq. 38}$$

It is necessary to have Z_S smaller than Z_O. Thus,the reflection coefficients have the same sign to give exponential behavior. Opposite signs give oscillatory behavior.

If $Z_S < Z_O$, the exponential approximation becomes more accurate. If Z_S is very small compared to Z_O, then T_O is negligible compared to lL/Z_O, so that Equation 35 reduces to

$$k = -l\frac{L}{Z_S} \qquad \text{Eq. 39}$$

But lL is the total loop inductance, and Z_S is the circuit's total series impedance. The time constant is then

$$k = \frac{L'}{R_S} \qquad \text{Eq. 40}$$

This is the same time constant you would obtain by a circuit analysis approach if you considered the line a series combination of L′ and R_S. By open-circuiting the line and performing a similar analysis, it can be shown that an RC time constant results.

Types of Transmission Lines

The types of transmission lines include

- Coaxial cable
- Twisted pair
- Wire over ground
- Microstrip lines
- Strip lines

Coaxial Cable

Coaxial cable offers many advantages for distributing high-frequency signals. The well-defined and uniform characteristic impedance permits easy matching. The cable's ground shield reduces crosstalk, and the low attenuation at high frequencies make the cable ideal for transmitting the fast rise-time and fall-time signals generated by Cypress CMOS ICs. However, because of high cost, coaxial cable is usually restricted to applications that permit no alternatives. These applications usually involve clock distribution systems on PCBs or backplanes.

Because coaxial cable is not easily handled by automated assembly techniques, its application requires human assemblers. This requirement further increases costs.

Coaxial cables have characteristic impedances of 50Ω, 75Ω, 93Ω, or 150Ω. These values are the most common, although special cables can be made with other impedances.

Coaxial cable's propagation delay is very low. You can compute it using the formula

$$t_{pd} = 1.017\sqrt{e_r} \ (ns/ft) \qquad \text{Eq. 41}$$

where e_r is the relative dielectric constant and depends upon the dielectric material used. For solid Teflon and polyethylene, the dielectric constant is 2.3. The propagation delay is 1.54 ns per foot. For maximum propagation velocity, you can use coaxial cables with dielectric Styrofoam or polystyrene beads in air. Many of these cables have high-characteristic impedances and are slowed considerably when capacitively loaded.

Twisted Pair

You can make twisted pairs from standard wire (AWG 24 – 28), twisted about 30 turns per foot. The typical characteristic impedance is 110Ω.

Because the propagation delay is directly proportional to the characteristic impedance (*Equation 9*), the propagation delay is approximately twice that of coaxial cable. Twisted pairs are used for backplane wiring, sometimes for driving differential receivers, and for breadboarding.

Wire Over Ground

Figure 11 shows a wire over ground. This configuration is used for breadboarding and backplane wiring. The characteristic impedance is approximately 120Ω. This value can vary as much as ±40 percent, depending upon the distance from the groundplane, the proximity of other wires, and the configuration of the ground.

Microstrip Lines

A microstrip line (*Figure 12*) is a strip conductor (signal line) on a PCB separated from a ground plane by a dielectric. If the line's thickness, width, and distance from the ground plane are controlled, the line's characteristic impedance can be predicted with a tolerance of ±5 percent.

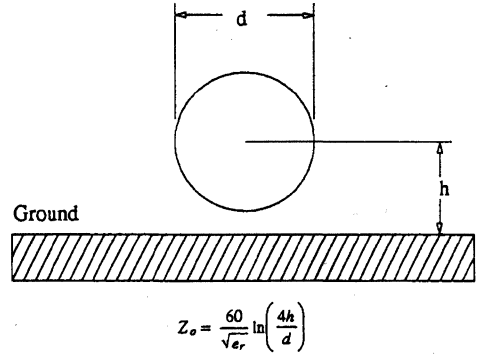


Figure 11. Wire Over Ground

The formula given in *Figure 12* has proven to be very accurate for width-to-height ratios between 0.1:1 and 3.0:1 and for dielectric constants between 1 and 15.

The inductance per foot for microstrip lines is

$$L = (Z_O)^2 C_O \qquad \text{Eq. 42}$$

where Z_O is the characteristic impedance and C_O is capacitance per foot.

The propagation delay of a microstrip line is

$$t_{pd} = 1.017\sqrt{0.45e_r + 0.67}\ (ns/ft) \qquad \text{Eq. 43}$$

Note that the propagation delay depends only upon the dielectric constant and is not a function of the line width or spacing. For G-10 fiberglass epoxy PCBs (dielectric constant of 5), the propagation delay is 1.74 ns per foot.

Strip Lines

A strip line consists of a copper strip centered in a dielectric between two conducting planes (*Figure 13*). If the line's thickness, width, dielectric constant, and distance between ground planes are all controlled, the tolerance of the characteristic impedance is within ±5 percent. The equation given in *Figure 13* is accurate for W/(b − t) < 0.35 and t/b < 0.25.

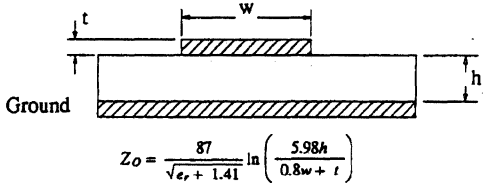


Figure 12. Microstrip Line

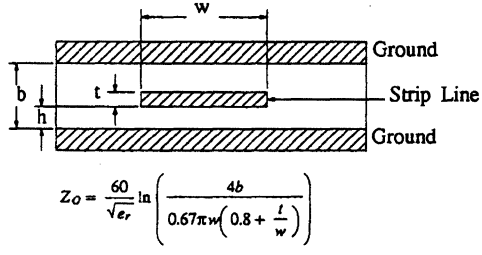


Figure 13. Strip Line Construction

The inductance per foot is given by the formula

$$L = (Z_O)^2 C_O \qquad \text{Eq. 44}$$

The propagation delay of the line is given by the formula

$$t_{pd} = 1.017\sqrt{e_r}\ (ns/ft) \qquad \text{Eq. 45}$$

For G-10 fiberglass epoxy boards, the propagation delay is 2.27 ns per foot. The propagation delay is not a function of line width or spacing.

Modern PCBs

Most PCBs employ microstrip, stripline, or some combination of the two. Microstrip construction on a double-sided board with power and ground nets can suffice for low- to medium-performance, and low-density PCBs.

For high-performance, high-density PCBs, stripline construction is preferred. Power planes isolate signal layers from each other and provide higher-quality power and grounds than those of a two-layer board. Manufacturing quality control assures that the metalization is of uniform thickness and that the layers are properly laminated, thus ensuring uniform, predictable electrical characteristics.

When to Terminate Transmission Lines

Transmission lines should be terminated when they are long. From the preceding analysis, it should be apparent that

$$Long\ Line > \frac{t_r}{2t_{pdL}} \qquad \text{Eq. 46}$$

where t_{pdL} is the loaded propagation delay of the line per unit length. For Cypress CMOS and BiCMOS products, the rise time, t_r, is typically 2 ns.

For stripline construction (multilayer PCBs), the line length at which voltage reflections might occur has been shown to vary from 4.73 inches for a 10-pF load to 3.05 inches for an 80-pF load (see *Equation 18* and *Table 1*).

Not all lines exceeding these lengths need to be terminated. Terminations are usually required on control lines (such as clock inputs, write and read strobe lines on SRAMs and FIFOs) and chip select or output-enable lines on RAMs, PROMs, and PLDs. Address lines and data lines on RAMs and PROMS usually have time to settle because they are normally not the highest-frequency lines in a system. However, if very heavily loaded, address and databus lines might require terminations.

Line Termination Strategies

There are two general strategies for transmission line termination:

1. Match the load impedance to the line impedance
2. Match the source impedance to the line impedance

In other words, if either the load reflection coefficient or the source reflection coefficient can be made to equal zero, reflections are eliminated. From a systems design viewpoint, strategy 1 is preferred. Eliminating the reflection at the load (i.e., dissipating the excess energy) before the energy travels back to the source causes less noise, electromagnetic interference (EMI), and radio frequency interference (RFI).

Multiple Loads, Buses, and Nodes

In the case where multiple loads are connected to a transmission line, only one termination circuit is required. The termination should be located at the load that is electrically the greatest distance from the source. This is usually the load that is the greatest physical distance from the source. A point-to-point or daisy chain connection of loads is preferred.

Bidirectional buses should be terminated at each end with a circuit whose impedance equals the intrinsic, characteristic line impedance. The reason is that each transmitting device sees the characteristic impedance of the line when the device is transmitting.

Consider next a line that has three bidirectional nodes: one on each end and one in the middle. The middle node, when driving the line, sees an impedance equal to $Z_O/2$, because the node is looking into two lines in parallel with each other. The end nodes, however, see an impedance of Z_O. In this case, as in a backplane, each end of the line should be terminated in an impedance equal to $Z_O/2$. When heavily loaded, *Equation 12* must be used to calculate the loaded characteristic impedance, and this must be used instead of Z_O.

Types of Terminations

There are three basic types of terminations: series damping, pull-up/pull-down, and parallel AC terminations. Each has its advantages and disadvantages.

Except for series damping, the termination network should be attached to the input (load) that is electrically the greatest distance from the source. Component leads should be as short as possible to prevent reflections due to lead inductance.

Series Damping

Series damping is accomplished by inserting a small resistor (typically 10Ω to 75Ω) in series with the transmission line, as close to the source as possible (*Figure 14*). Series damping is a special case of damping in which the series resistor value plus the circuit output impedance equals the transmission line impedance. The strategy is to prevent the wave reflected back from the load from reflecting back from the source. This is done by making the source reflection coefficient equal to zero.

The channel resistance (on resistance) of the pull-down device for Cypress ICs is 10Ω to 20Ω, depend-

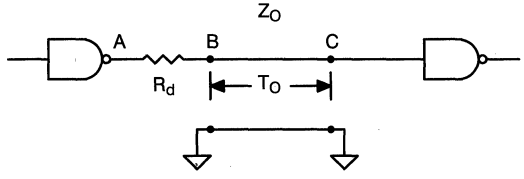


Figure 14. Series Damping Termination

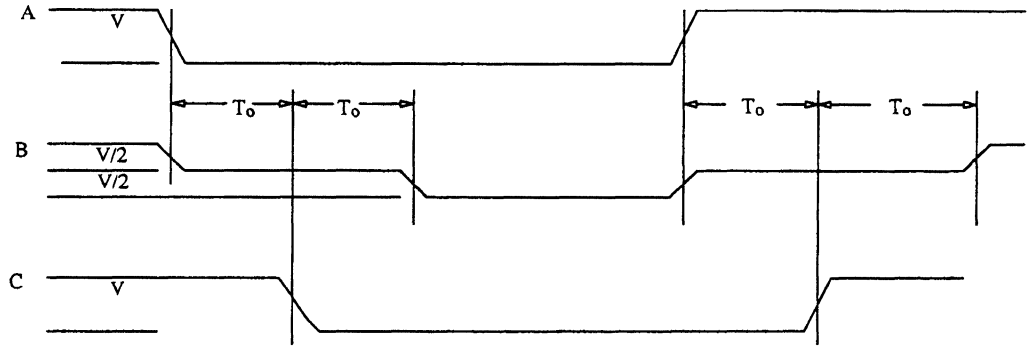


Figure 15. Series Damping Timing

ing upon the current-sinking requirements. Thus, subtract this value from the series-damping resistor, R_d.

$$Z_O = R_S + R_d \qquad \text{Eq. 47}$$

A disadvantage of the series-damping technique, as illustrated in *Figure 15*, is that during the two-way propagation delay time of the signal edges, the voltage at the input to the line is halfway between the logic levels, due to the voltage divider action of R_S. The "half voltage" propagates down the line to the load and then back from the load to the source. This means that no inputs can be attached along the line, because they would respond incorrectly during this time. However, you can attach any number of devices to the load end of the line because all the reflections are absorbed at the source. If two or more transmission lines must be driven in parallel, the value of the series-damping resistor does not change.

The advantages of series termination are:

- Requires only one resistor per line
- Consumes little power
- Permits incident wave switching at the load after a T_O propagation delay
- Provides current limiting when driving highly capacitive loads; the current limiting also helps reduce groundbounce

The disadvantages of series termination are:

- Degrades rise time at the load due to increased RC time constant
- Should not be used with distributed loads

The low input current required by Cypress CMOS ICs results in essentially no DC power dissipation. The only AC power required is to charge and discharge the parasitic capacitances.

Pull-Up/Pull-Down Termination

The pull-up/pull-down resistor termination shown in *Figure 16* is included for historical reasons and for the sake of completeness. For TTL driving long cables, such as ribbon cables, the values $R_1 = 220\Omega$ and $R_2 = 330\Omega$ are recommended by several bus interface standards. If the cable is disconnected, the voltage at point B is 3V, which is well above the 2V minimum high TTL specification. Because most

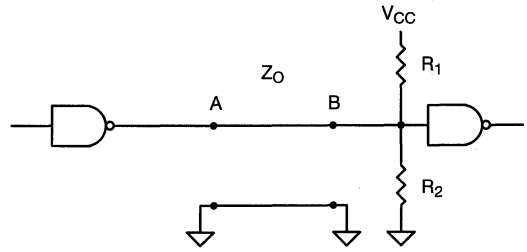


Figure 16. Pull-Up/Pull-Down

control signals are active LOW, a disconnected cable results in the unasserted state.

The maximum value of R_1 is determined by the maximum acceptable signal rise time, which is a function of the charging RC time constant. The minimum value of R_1 is determined by the amount of current the driver can sink. The value of R_2 is chosen such that a logic HIGH is maintained when the cable is disconnected. The equivalent Thévenin resistance is

$$R_T = \frac{R_1 R_2}{R_1 + R_2} \quad \text{Eq. 48}$$

The value of R_1 and R_2 in parallel is slightly less than the cable's characteristic impedance. Ribbon cables with characteristic impedances of 150Ω are typical.

If both resistors are used, DC power is dissipated all the time. If only a pull-down resistor (R_2) is used, DC power is dissipated when the input is in the logic HIGH state. Conversely, if only a pull-up resistor (R_1) is used, power is dissipated when the input is in the LOW state. Due to these power dissipations, this termination is not recommended.

If an unterminated control signal on a PCB is suspected of causing a problem, a resistor whose value is slightly less than the characteristic impedance of the line (e.g., 47Ω) can be connected between the input pin and ground. Be sure that the driver can source sufficient current to develop a TTL high voltage level (2.0V) across the resistor.

In special cases where inputs should be either pulled up (HIGH) for logic reasons or because of very slow rise and fall times, you can use a pull-up resistor to V_{CC} in conjunction with the terminating network shown in *Figure 17*. DC power is dissipated when the source is LOW.

Parallel AC Termination

Figure 17 illustrates the recommended general-purpose termination. It does not have the disadvantage of the half-voltage levels of series damping terminations, and it causes no DC power dissipation. You can attach loads anywhere along the line, and they see a full voltage swing.

The disadvantage is that a parallel AC termination requires two components, versus the one-component series-damping termination.

Commercially Available RC Networks

A variety of combinations of R and C values are available as series RC networks in SIP packages from at least two sources.

Bourns calls these networks the Series 701 and 702 RC Termination Networks. You can obtain datasheets by calling the factory in Logan, Utah (801-750-7200) or a local sales office.

Thin Film Technology also refers to the networks as RC Termination Networks. You can obtain datasheets by calling the factory in North Mankato, Minnesota at 507-635-8445.

Dale Electronics calls their product Resistor/Capacitor Networks. Call 915–595–8139 for information.

California Micro Devices calls their product R–C Networks. Call 408–263–3214 for information.

Low-Pass Filter Analysis

The parallel AC termination has another advantage: it acts as a low-pass filter for short pulses. You can verify this by analyzing the response of the circuit illustrated in *Figure 18* to a positive and a negative step function. The positive step function is generated by moving the switch from position 2 to position 1. The negative step function is generated by moving the switch from position 1 to position 2. The response of the circuit to a pulse is the superposition of the two separate responses. The input impedance of the Cypress circuits connected to the

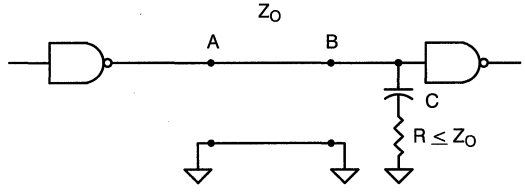


Figure 17. Parallel AC Termination

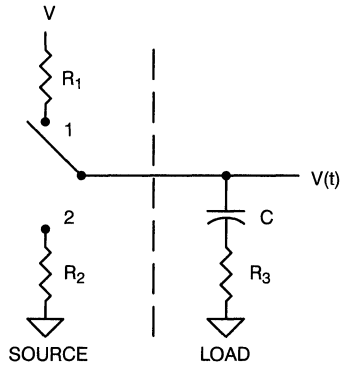


Figure 18. Lumped Load; AC Termination

termination network are so large that they can be ignored for this analysis.

Classic circuit analysis usually assumes an ideal source ($R_1 = R_2 = 0$). In real-world digital circuits, the source output impedance is not only non-zero, but also varies depending upon whether the output is changing from LOW to HIGH or vice versa.

For Cypress ICs, $100\Omega > R_1 > 50\Omega$ and $20\Omega > R_2 > 10\Omega$, depending upon speed and output current-sinking requirements.

Positive Step Function Response

The initial voltage on the capacitor is zero. At t = 0, the switch is moved from position 2 to position 1. At t = 0+, the capacitor appears as a short circuit, and the voltage V is applied through R_1 to charge the load (R_3C). The voltage across the capacitor $V_C(t)$, is

$$V_C(t) = V\left(1 - e^{\left[\frac{-t}{(R_1 + R_3)C}\right]}\right) \quad \text{Eq. 49}$$

In theory, the voltage across the capacitor reaches V when t equals infinity. In practice, the voltage reaches 98 percent of V after 3.9 RC time constants. You can verify this by setting $V_C(t)/V = 0.98$ in *Equation 49* and solving for t.

Negative Step Function Response

The capacitor is charged to approximately V. At t = 0, the switch is moved from position 1 to position 2, and the capacitor is discharged. The voltage across the capacitor, $V_C(t)$ is

$$V_C(t) = Ve^{\left[\frac{-t}{(R_2 + R_3)C}\right]} \quad \text{Eq. 50}$$

The voltage decays to 2 percent of its original value in 3.9 RC time constants. You can verify this by setting $V_C(t)/V = 0.02$ in *Equation 50* and solving for t.

The Ideal Case

Consider the ideal case where $R_1 = R_2 = 0$. Let $R_3 = R$ in *Equations 49* and *50*. If a positive pulse of width T is applied to the modified circuit of *Figure 18*, the pulse disappears if 4RC > T.

Because the discharging time constant is the same as the charging time constant for the ideal case, a negative-going pulse of width T also disappears if 4RC > T. That is, if the applied signal is normally HIGH and goes LOW, as does the write strobe on an SRAM, the termination filters out all negative glitches less than 4 RC time constants in width.

The maximum frequency that the circuit passes is

$$F(max.) = \frac{1}{2T} \quad \text{Eq. 51}$$

This is true because the charging and discharging time constants are equal for the ideal case.

Capacitance for the Ideal Case

The value of the capacitor, C, must be chosen to satisfy two conflicting requirements. First, the capacitor should be large enough to either absorb or supply the energy contained or removed when positive-going or negative-going glitches occur. Second, the capacitor should be small enough to avoid either delaying the signal beyond some design limit or slowing the signal rise and fall times to more than 5 ns.

A third consideration is the impedance caused by the capacitor's capacitive reactance, X_C. The digital waveforms applied to the AC termination can be ex-

pressed as a Fourier Series so that they can be manipulated mathematically. However, because these signals are not periodic in the classical meaning of the word, it is not clear that the AC steady-state analysis model of X_C applies here.

In most applications, the degradation of the signal's rise and fall times beyond 5 ns determines the maximum value of the capacitor. The procedure is to calculate the rise time between the 10- and 90-percent amplitude levels, equate this rise time to 5 ns, and solve for C in terms of R:

$$V(t) = V(1 - e^{\left[\frac{-t}{RC}\right]}) \qquad \text{Eq. 52}$$

for t yields

$$t = RC \ln\left[\frac{1}{1 - \frac{V(t)}{V}}\right] \qquad \text{Eq. 53}$$

For $\frac{V(t)}{V} = 0.1$, $t = 0.10\ RC$.

For $\frac{V(t)}{V} = 0.9$, $t = 2.3\ RC$.

The time for the signal to transition from 10 to 90 percent of its final value is then T = 2.2 RC. Solving for C yields

$$C = \frac{T}{2.2R} \qquad \text{Eq. 54}$$

For T = 5 ns, *Table 2* can be constructed. This table indicates that 50Ω transmission lines on PCBs that are terminated with RC networks should use a 47Ω resistor and a capacitor of 48 pF max; 47 pF is a standard value. This network eliminates glitches of 9 ns or less. The table's second column applies to wirewrapping construction, which is not recommended for systems operating at frequencies over 10 MHz. An exception is if the system consists of less than six MSI or SSI ICs.

Table 2. Termination Value for an Ideal Case

	PCB	Wirewrapped
Z_O (Ω)	50	120
R (Ω)	47	110
C (max., pF)	48	20
RC (ns)	2.25	2.2
4RC (ns)	9	8.8

The Real World

To go from the ideal to the real world, calculate the values of R_1 and R_2 from the curves on the datasheet of the device driving the line. R_1 is the slope of the output source current vs. output voltage between 2 and 4V. R_2 is the slope of the output sink current vs output voltage between 0 and 0.8V.

Add the value of R_1 to 47Ω and calculate C, using *Equation 54*. Then check to see that the RC charging time constant does not violate some minimum positive pulse-width specification for the line. If so, reduce C.

Add the value of R_2 to 47Ω and calculate C. Then check to see if the discharging RC time constant violates some minimum pulse-width specification for the line. If so, reduce C.

If the line is heavily loaded, *Equation 12* must be used to calculate the loaded characteristic impedance, which determines the maximum value of R. The Maximum value of C is then calculated using *Equation 54*.

Schottky Diode Termination

In some cases it can be expedient to use Schottky diodes or fast-switching silicon diodes to terminate lines. The diode switching time must be at least four times as fast as the signal rise time. Where line impedances are not well defined, as in breadboards and backplanes, the use of diode terminations is convenient and can save time.

A typical diode termination appears in *Figure 19*. The Schottky diode's low forward voltage, V_f (typically 0.3 to 0.45V), clamps the input signal to a V_f below ground (lower diode) and $V_{CC} + V_f$ (upper diode). This significantly reduces signal undershoot

and overshoot. Some applications may not require both diodes.

The advantages of diode terminations are:

- Impedance matched lines are not required
- The diodes replace terminating resistors or RC terminations
- The diodes' clamping action reduces overshoot and undershoot
- Although diodes cost more than resistors, the total cost of layout might be less because a precise, controlled transmission-line environment is not required
- If ringing is discovered to be a problem during system debug, the diodes can be easily added

As with resistor or RC terminations, the leads should be as short as possible to avoid ringing due to lead inductance.

A few of the types of Schottky diodes commercially available are

- HSMS–2822 (Hewlet–Packard)
- 1N5711
- MBD101, MBD102 (Motorola)
- SN74S1050/52/56 (TI, single-diode arrays)
- SN74S1051/53 (TI, double-diode arrays)

Unterminated Line Example

The following example illustrates the procedure for calculating the waveforms when a Cypress PLD generates the write strobe for four Cypress FIFOs. The PLD is a PALC16L8 device and the FIFOs are CY7C429s.

The equivalent circuit appears in *Figure 20* and the unmodified driving waveform in *Figure 21*. The rise and fall times are 2 ns. The length of the stripline trace on the PCB is 8 inches and the intrinsic characteristic line impedance is 50Ω. The voltage waveforms at the source (point A) and the load (point B) must be calculated as functions of time. Stripline construction is used for this example because in most modern high-performance digital systems, the PCBs have multiple layers.

The equivalent ON channel resistance of the PLD pull-up device, 62Ω, is calculated using the output

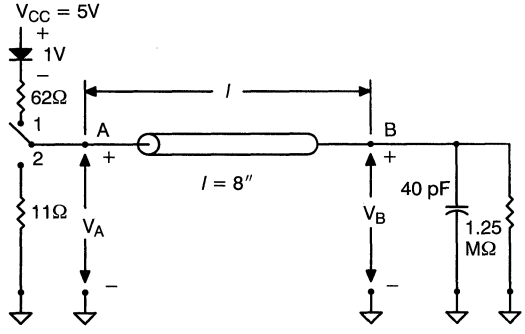


Figure 20. Equivalent Circuit for Cypress PAL Driving

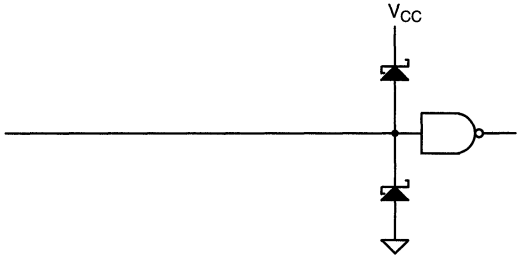


Figure 19. Schottky Diode Termination

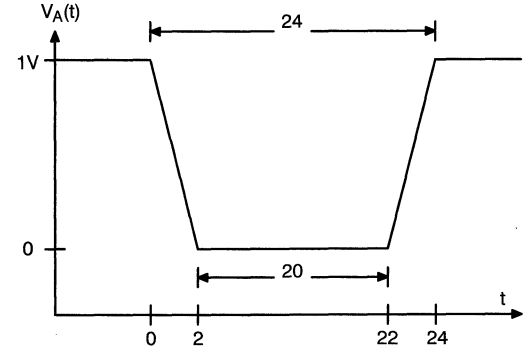


Figure 21. $V_A(t)$, Unmodified

source current versus voltage graph, over the region of interest (2 to 4V), from the PALC20 series datasheet. The equivalent resistance of the pull-down device, 11Ω, is calculated in a similar manner, using the output sink current versus output voltage graph, over the region of interest (0.4 to 2V), also on the datasheet.

The equivalent input circuit for the FIFO is constructed by approximating the input and stray capacitance with a 10-pF capacitor and the input resistance with a 5-MΩ resistor. The input leakage current for all Cypress products is specified as a maximum of ±10 μA, which guarantees a minimum of 500 KΩ at V_{in} = 5V. Typical leakage current is 10 pA.

Because the PLD is driving four FIFOs in parallel, the equivalent lumped capacitance is 4 × 10 pF = 40 pF, and the equivalent lumped resistance is 5,000,000/4 = 1.25 MΩ

The next step is to calculate the propagation delay and the loaded characteristic impedance of the line. The unloaded propagation delay of the line is calculated using *Equation 45* with a dielectric constant of 5:

$$t_{pd} = 2.27\ (ns/ft) \qquad \text{Eq. 55}$$

To calculate the loaded line propagation delay, the intrinsic capacitance must first be calculated using *Equation 9.*

$$t_{pd} = Z_O C_O \qquad \text{Eq. 56}$$

where Z_O is the intrinsic characteristic impedance, and C_O is the intrinsic capacitance.

$$C_O = \frac{t_{pd}}{Z_O} = \frac{2.27\ ns/ft}{50} = 45.4\ pF/ft. \qquad \text{Eq. 57}$$

Because the line is loaded with 40 pF, *Equation 11* is used to compute the loaded propagation delay of the line.

$$t_{pdL} = t_{pd}\sqrt{1 + C_D/C_O}$$

$$t_{pdL} = 2.27\ ns/ft \sqrt{1 + \frac{40\ pF}{45.4\ pF/ft \times \frac{8\ in.}{12\ in./ft}}}$$

$$t_{pdL} = 3.46\ ns/ft \qquad \text{Eq. 58}$$

Note that the capacitance per unit length must be multiplied by the line length to arrive at an equivalent lumped capacitance.

The intrinsic line impedance is reduced by the same factor by which the propagation delay is increased (1.524; see *Equation 12*):

$$Z_O' = \frac{50\Omega}{1.524} = 32.8\Omega \qquad \text{Eq. 59}$$

Initial Conditions

At time t = 0, the circuit shown in *Figure 20* is in a quiescent state. The voltage at points A and B must be the same. By inspection:

$$V_A = V_B = (V_{CC} - Vf)\left(\frac{R_L}{R_S + R_L}\right)$$

$$= (5 - 1)\left(\frac{1.25 \times 10^6}{28 + 1.25 \times 10^6}\right) = 4V \qquad \text{Eq. 60}$$

At t = 0, the driving waveform changes from 4V to approximately 0V with a fall time of 2 ns. This is shown in *Figure 20* by the switch arm moving from position 1 to position 2.

The wave propagates to the load at the rate of 3.46 ns per foot and arrives there

$$T_O = 3.46\ ns/ft \times \frac{8\ in.}{12\ in./ft} = 2.3\ ns \qquad \text{Eq. 61}$$

later, as illustrated in *Figure 22b.*

Because the reflection coefficient at the load is ϱL = 1, an early equal and opposite polarity waveform is propagated back to the source from the load. The reflection arrives at t = $2T_O$ = 4.6 ns (*Figure 22a*). Note that the fall time is preserved.

The reflection coefficient at the source is

$$\rho_S = \frac{R_S - Z_O'}{R_S + Z_O'} = \frac{11 - 32.8}{11 + 32.8} = -0.498 \qquad \text{Eq. 62}$$

To simplify the calculations that follow, consider −0.5 to be the low-level source reflection coefficient. The magnitude of the reflected voltage at the source is then

$$V_{S1} = -4V \times (-0.5) = 2V \qquad \text{Eq. 63}$$

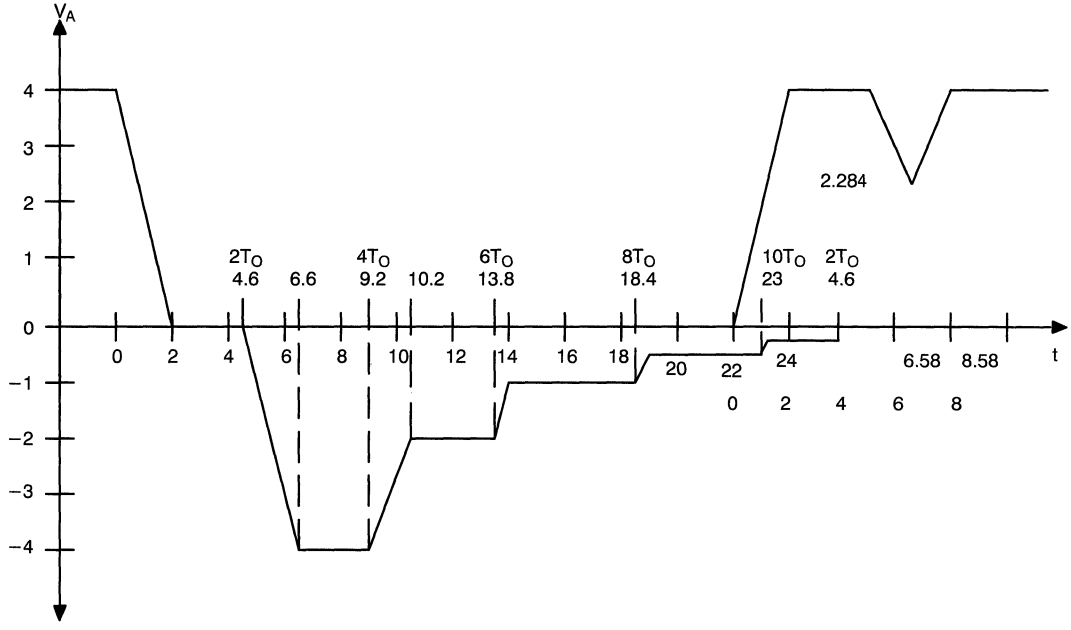


Figure 22a. Unterminated Line Example; $V_A(t)$

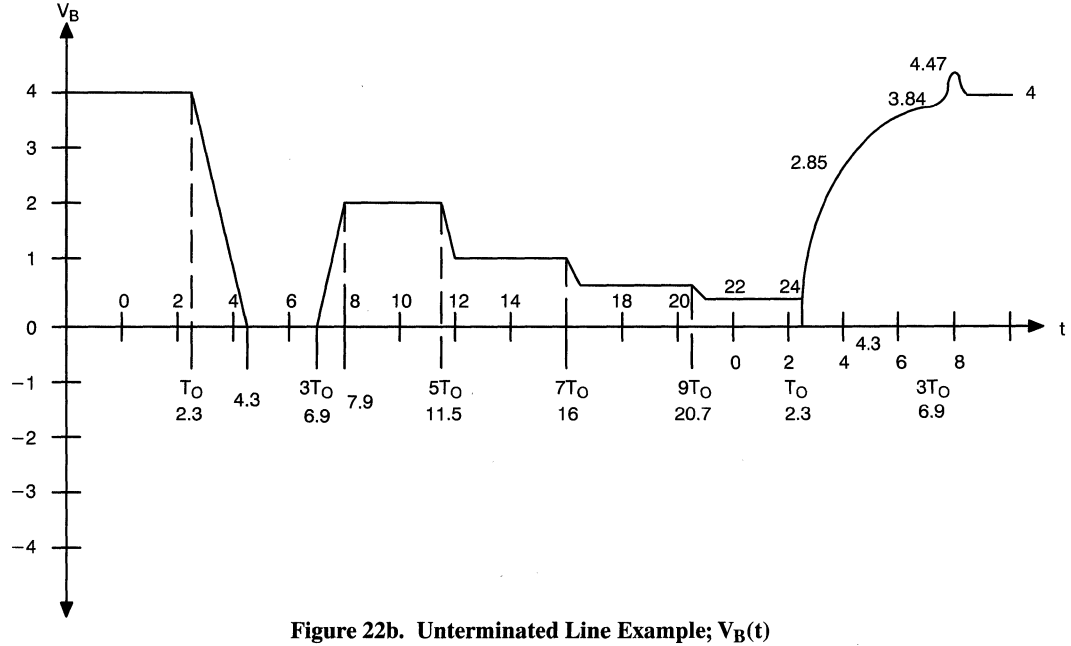


Figure 22b. Unterminated Line Example; $V_B(t)$

This wave propagates from the source to the load and arrives at t = 3 T_O. The wave adds to the 0V signal. The rise time is preserved, and thus the time required for the signal to go from 0 to 2V is

$$t_r = \frac{2V \times 2\,ns}{4V} = 1\,ns \qquad \text{Eq. 64}$$

The signal at the load thus reaches the 2V level at time

$$t = 3T_O + 1\,ns = 7.9\,ns \qquad \text{Eq. 65}$$

and remains at that level until the next reflection occurs at

$$t = 5T_O \qquad \text{Eq. 66}$$

The wave that arrives at the load at 3 T_O reflects back to the source and arrives at

$$t = 4T_O = 9.2\,ns \qquad \text{Eq. 67}$$

The 2V level adds to the −4V level, for a total of −2V. The rise time is preserved, so that this level is reached at

$$t = 4T_O + 1\,ns = 10.2\,ns \qquad \text{Eq. 68}$$

and maintained until the next reflection occurs at

$$t = 6T_O \qquad \text{Eq. 69}$$

The 2V wave that arrives at the source at t = 4T_O reflects back to the load and arrives at t = 5T_O. The portion that is reflected back to the load is

$$V_{S2} = 2 \times (-0.5) = -1V \qquad \text{Eq. 70}$$

This value subtracts from the 2V level to give 2 − 1 = 1V. Because the fall time is preserved, the time required for the signal to go from 2 to 1V is

$$t_f = \frac{1V \times 2\,ns}{4V} = 0.5\,ns \qquad \text{Eq. 71}$$

The 1V level is thus reached at time

$$t = 5T_O + 0.5\,ns = 12\,ns \qquad \text{Eq. 72}$$

At t = 6T_O, the 1V wave arrives back at the source, where it subtracts from the −2V level to give −1V. The rise time is

$$t_r = 1 \times 0.5\,ns/V = 0.5\,ns \qquad \text{Eq. 73}$$

The signal at the source reaches the −1V level at

$$t = 6T_O + 0.5 = 14.3\,ns \qquad \text{Eq. 74}$$

The 1V wave that arrives at the source at t = 6T_O is reflected back to the load and arrives at t = 7T_O. The portion that is reflected back is

$$V_{S3} = 1 \times (-0.5) = -0.5V \qquad \text{Eq. 75}$$

This value subtracts from the 1V level to give 0.5V. The fall time is 0.25 ns. The 0.5V level remains until the next reflection reaches the load at

$$t = 9T_O \qquad \text{Eq. 76}$$

At t = 8T_O the 0.5V wave that reflects from the load at t= 7T_O arrives back at the source, where it subtracts from the −1V level to give −0.5V. The rise time is 0.25 ns. The portion that reflects back to the load is

$$V_{S4} = 0.5 \times (-0.5) = -0.25V \qquad \text{Eq. 77}$$

The −0.25V signal arrives at the load at t = 10T_O = 23 ns and subtracts from the 0.5V signal to give 0.25V.

This process continues until the voltages at points A and B decay to approximately 0V.

Observations

The positive reflection coefficient at the load and the negative reflection coefficient at the source result in an oscillatory behavior that eventually decays to acceptable levels. The voltage at point A reaches −1V after 6T_O delays and the voltage at point B reaches 0.5V after 7T_O delays.

The reflection at the load that causes the voltage to equal the TTL minimum one level (2V) at T = 3T_O causes a problem. The actual input voltage threshold level is 1.5V for TTL-compatible devices that do not exhibit hysteresis.

The voltage at the load falls from 4V to 0V in 2 ns, beginning at t = T_O. Because T_O = 2.3 ns, the voltage reaches zero at

$$2.3\,ns + 2\,ns = 4.3\,ns \qquad \text{Eq. 78}$$

The 1.5V level occurs at

$$4.3\,ns - \frac{2\,ns}{4V} \times 1.5V = 3.55\,ns \qquad \text{Eq. 79}$$

The rising edge begins at

$$t = 3T_O = 6.9\ ns \qquad \text{Eq. 80}$$

The 1.5V level occurs at

$$6.9\ ns + \frac{2\ ns}{4V} \times 1.5 = 7.65\ ns \qquad \text{Eq. 81}$$

The time difference (7.65 − 3.55 = 4.1 ns) is long enough for the FIFO to interpret the signal as a LOW.

Next, consider the width of the positive pulse that begins at the load at t = $3T_O$. Because the rise time is preserved, the signal takes 1 ns to reach 2V, or 0.75 ns to reach 1.5V. The signal begins to fall at t = $5T_O$, reaching 1.5V at

$$t = 5T_O + 0.25\ ns = 11.75\ ns \qquad \text{Eq. 82}$$

The difference (11.75 − 7.65) is 4.1 ns, which is wide enough for the FIFO to interpret as a second clock. To eliminate this pulse, the line must be terminated.

Strobe Shortening Considerations

In this example the width of the negative strobe is 22 to 24 ns. If a CY7C429-20 FIFO is used, the write (or read) strobe must not be shorter than 20 ns. Even if the FIFO does not recognize the 4.5-ns negative pulse, the shortening of the write strobe by $5T_O$ = 11.5 ns is sufficient to violate the minimum negative-pulse-width specification.

This strobe-shortening phenomenon might also occur on other active-LOW control lines such as output enables and chip selects. Clock lines must also be analyzed for this problem; in general, these lines should be terminated.

Now consider an analysis of the write strobe's rising edge to assure that the reflections associated with this edge do not cause multiple clocks or false triggering of the FIFO. At t = 22 ns, the rising edge of the write strobe begins, which is the equivalent of closing the switch in *Figure 20* in the 1 position. For this analysis, it is convenient to start the timescale over at zero, as appears in *Figure 22a* and *b*.

If the forcing function were a step function, the equations of *Figure 4h* would apply. The time constant in the equation is

$$T = \frac{RZ_O'C_e}{R + Z_O'} \qquad \text{Eq. 83}$$

Because

$$R > Z_O',\ T = Z_O'C_e \qquad \text{Eq. 84}$$

where $Z_O' = 32.8\Omega$ and $C_e = 45.4$ pF.

This is the equivalent of saying that you can ignore the1.25-MΩ device input resistance for transient circuit analysis. Substituting Z_O' and C_e into the preceding equation yields a time constant of T = 1.489 ns.

Writing the equation for the voltages for the circuit of *Figure 20* yields

$$V_B(t) = iZ_O' + \frac{1}{C_e}\int_o^t i\,dt \qquad \text{Eq. 85}$$

Also,

$$V_B(t) = K_tU(t) - K(t - T1)\,U(t - T1) \qquad \text{Eq. 86}$$

where K_t is the rising edge of the write strobe (K = 2V/ns) applied at t = 0 using a unit step function, U(t); and −K(t − T1) represents an equal but opposite waveform applied at t = T1 (after the rise time) using a unit step function, U(t − T1).

Equating the expressions and taking the LaPlace transforms of both sides yields

$$\frac{K}{s^2} - \frac{Ke^{-T1s}}{s^2} = Z_O'I(s) + \frac{I(s)}{C_es} = \left(Z_O' + \frac{1}{C_es}\right)I(s) \qquad \text{Eq. 87}$$

However,

$$V_B(t) = \frac{1}{C_e}\int_0^t i\,dt, \quad \text{or,} \quad V_B(s) = \frac{I(s)}{C_es} \qquad \text{Eq. 88}$$

Therefore,

$$\frac{K}{s^2} - \frac{Ke^{-T1s}}{s^2} = \left(Z_O' + \frac{1}{C_es}\right)C_esV_B(s) \qquad \text{Eq. 89}$$

Solving for $V_B(s)$ yields

$$V_B(s) = \frac{\frac{K}{s^2}(1 - e^{-T1s})}{C_es\left(Z_O' + \frac{1}{C_es}\right)} \qquad \text{Eq. 90}$$

which is equivalent to

$$\frac{\frac{K}{Z_O'C_e}(1 - e^{-T1s})}{s^2\left(s + \frac{1}{Z_O'C_e}\right)} \quad \text{Eq. 91}$$

Taking the inverse LaPlace transform yields

$$V_B(t) = \left[KZ_O'C_e\left(e^{\frac{-t}{Z_O'C_e}} - 1\right) + K_t\right]U(t) -$$

$$\left[KZ_O'C_e\left[e^{\left[\frac{-(t-T1)}{Z_O'C_e}\right]} - 1\right] + K(t - T1)\right]U(t - T1) \quad \text{Eq. 92}$$

The first term in *Equation 92* applies from time zero up to and including T1, and the second term applies after T1:

$$V_B(t) = \frac{KZ_O'C_e}{T1}\left(e^{\left[\frac{-t}{Z_O'C_e}\right]} - 1\right) + \frac{K}{T1}(t) \quad \text{Eq. 93}$$

for t ≤ T1.

$$V_B(t) = \frac{KZ_O'C_e}{T1}\left(1 - e^{\left[\frac{T1}{Z_O'C_e}\right]}\right)e^{\left[\frac{-t}{Z_O'C_e}\right]} + K1 \quad \text{Eq. 94}$$

for t > T1.

where K1 is the final value, which is 4V.

Substituting the correct values for t = T1 = 2 ns yields

$$V_B(t = T1) = \frac{2 \times 32.8 \times 45.4 \times 10^{-12}}{2 \times 10^{-9}}(\epsilon^{-1.489} - 1) + \frac{2V}{ns} \times 2\,ns$$

$$= -1.15 + 4 = 2.85V \quad \text{Eq. 95}$$

If the forcing function is a step function, the equation is

$$V_B(t) = 4V\left(1 - e^{\left[\frac{-t}{Z_O'C_e}\right]}\right) \quad \text{Eq. 96}$$

at t = 2 ns, V_B = 3V, which is more than the 2.85V calculated using *Equation 93*.

At t = 22 ns + T_O, the voltage waveform begins to build up at the load and continues to build until the first reflection from the source occurs at t = $3T_O$.

Equation 94 is used to calculate the voltage at the load at t = $2T_O$, because $1T_O$ is used for propagation delay time:

$$V_B(t = 2T_O) = \frac{-2V \times 32.8 \times 45.4 \times 10^{-12}}{2 \times 10^{-9}}(1 - e^{-1.489})(e^{-2}) + 4$$

$$= -1.489\,(0.774)\,(0.1353) + 4$$

$$= -1.559 + 4 = 3.84V \quad \text{Eq. 97}$$

The voltage at the load remains at this value until the first reflection from the source reaches the load at t = $3T_O$.

Meanwhile, at t = T_O, the wave at the load reflects back to the source and arrives at t = $2T_O$. The wave subtracts from the 4V level at the source, as illustrated in *Figure 6c*. The amplitude of the droop is given by

$$V_r = \frac{C'Z_O'}{2}\frac{V_O}{T_r} \quad \text{Eq. 98}$$

for $R_S = Z_O$.

If R_S does not equal Z_O', *Equation 98* must be modified. Instead of $V_O/2$, the voltage is

$$V_O\left(\frac{R_S}{R_S + Z_O'}\right) \quad \text{Eq. 99}$$

so that *Equation 98* becomes

$$V_r = \frac{C'Z_O'V_O}{T_r}\left(\frac{R_S}{R_S + Z_O'}\right) \quad \text{Eq. 100}$$

where C′ = 40 pF, Z_O' =32.8Ω, R_S = 62Ω, T_r = 2 ns, and V_O = 4V. Substituting these values into *Equation 100* yields

$$V_r = 1.716V \quad \text{Eq. 101}$$

Because 4V − 1.716 = 2.284, the voltage does not drop below the minimum TTL V_{IH} level of 2V, but it does come close.

The reflection coefficient at the source is

$$\rho_S = \frac{R_S - Z_O'}{R_S + Z_O'} \quad \text{Eq. 102}$$

where, R_S = 62 ohms, Z_O' = 32.8 ohms, ϱ_s = 0.308.

The amount of voltage reflected from the source back to the load is then

$$V_{S1} = 1.716 \times 0.308 = 0.53V \quad \text{Eq. 103}$$

The 40-pF capacitor reduces the rise time of the waveform at the load. The reflection at the source caused by the load capacitor is insufficient to reduce the 4V level to less than the TTL one level (2V).

The reflection coefficient at the source is small enough so that the energy reflected back to the load is insufficient to cause a problem.

References

1. Matick, Richard E. *Transmission Lines for Digital and Communications Networks*. McGraw Hill, 1969.
2. Blood, Jr., William R. *MECL System Design Handbook*. Motorola Inc., 1983.

Protection, Decoupling, and Filtering of Cypress CMOS Circuits

This application note explains how to protect your ICs with a low-cost zener diode and why it is good insurance against inadvertent voltage transients. Also explained is the reason why decoupling and high-frequency-filtering capacitors are required. A method is provided for determining the capacitors' values.

Zener Diode Protection

Linear power supplies can cause large voltage transients. The transient is negative when it is caused by the collapse of a magnetic field and is positive when the supply is turned on.

Some commercially available laboratory bench supplies behave the same way. When they turn on, they can overshoot several volts. When they turn off, lead inductance can cause a negative transient voltage at the V_{CC} pin. If there is enough energy, this inductance can break down internal gate oxides, destroying or weakening the IC to the extent that it might fail later.

You can avoid this problem by adding a 20¢ zener diode (also called a voltage-regulator diode) between V_{CC} and ground. Connect the diode's cathode to V_{CC} and the anode to ground (see *Figure 1*). A 400-mW, 6.2V 1N525 or equivalent is recommended. You can also use the 1N753, a 500-mW, 6.2V zener diode.

If a voltage greater than the zener voltage (6.2V) occurs on V_{CC}, the diode breaks down, clamping the voltage to 6.2V and shunting the current to ground (see *Figure 2*). The diode can be destroyed if the current multiplied by the zener voltage exceeds the diode's power rating. Because zener diodes *always* fail shorted, they cause the power supply to "crowbar" and thus protect the ICs.

A negative voltage on the V_{CC} line puts a forward bias on the diode. This turns on the diode, which clamps the voltage to approximately −0.8V. If the negative voltage multiplied by the current exceeds the diode's power rating, the diode fails shorted, as in the reversed-bias case, and protects the ICs.

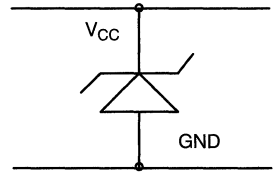


Figure 1. Zener Diode Connection

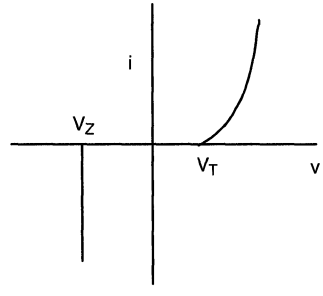


Figure 2. Zener Diode Characteristic

High-Frequency Filtering

In addition to the protection offered by zener diodes, decoupling and high-frequency filter capacitors are required on high-performance CMOS circuits. To use these capacitors effectively, you must understand why they are required.

To realize the fast rise and fall times that Cypress CMOS integrated circuits are capable of achieving, the power-distribution system must be able to supply the instantaneous current required when the device outputs switch from LOW to HIGH. The energy converted to current is stored as charge on the local decoupling capacitors. They decouple or isolate the circuit from the power-distribution system. It is standard practice to use one decoupling capacitor for each IC that drives a transmission line and one capacitor for every three devices that do not.

The PCB trace inductance plus the IC lead inductance can "current-starve" the output circuits, causing rise-time degradation. Remember that the current through an inductor cannot change instantaneously. Therefore, you must minimize any series inductance, including the lead inductance of the decoupling capacitors.

Decoupling-Capacitor Calculations

To determine the value of the decoupling capacitor, you must estimate the instantaneous current required when all the outputs of an IC switch from LOW to HIGH, assuming a reasonable droop of the voltage on the capacitor. The charge stored on the local decoupling capacitor is

$$Q = CV$$

Differentiating yields

$$i(t) = \frac{dQ}{dt} = C\frac{dV}{dt} \qquad \text{Eq. 1}$$

The characteristic impedance of a typical transmission line is 50Ω. Lines with a heavy capacitive load have lower characteristic impedances.

Next, assume that the IC is a nine-output FIFO, such as the CY7C429. The outputs reach

$$V_{CC} - V_t = 5V - 1V = 4V$$

Each output requires 4V/50Ω = 80 mA. Because the FIFO has nine outputs, it requires a total of 720 mA during the rise times of the outputs.

Solving *Equation 1* for C yields

$$C = i\frac{dt}{dv} \qquad \text{Eq. 2}$$

The last step is to assume a reasonable, tolerable droop in the capacitor voltage. Assume dV = 100 mV. Additionally, the signal rise and fall times are 2 ns. Substituting these values in *Equation 2* yields

$$C = \frac{720 \times 10^{-3} \times 2 \times 10^{-9}}{100 \times 10^{-3}}$$

$$= 14.4 \times 10^{-9}$$

$$= 0.0144\,\mu F$$

It is standard practice to use 0.01 to 0.1-μF decoupling capacitors. A 0.1-μF capacitor can supply 5A under the conditions assumed in the preceding calculations. Another way to look at the situation is that a 0.1-μF capacitor supplies 720 mA of instantaneous current in 2 ns with only 14.4 mV of voltage droop across the capacitor.

Decoupling capacitors for high-speed Cypress CMOS circuits should be of the high-K ceramic type with a low Effective Series Resistance (ESR). Capacitors using Z5U dielectric are a good choice.

High-Frequency Filter Capacitors

The 0.1 to 0.01-μF decoupling capacitors usually do not provide high-frequency decoupling or filtering. These capacitors do not behave like capacitors at high frequencies because their series resonance frequency is not high enough. This is primarily because of lead inductance in their construction, which is a result of the capacitor's relatively large value.

For high-frequency filter analysis, you can use the simplified capacitor equivalent circuit shown in *Figure 3*. R_S is the ESR, L is the Effective Series Inductance (ESL), and C is the capacitance.

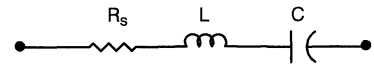


Figure 3. Simplified Capacitor Equivalent Circuit

The impedance of the simplified equivalent circuit is:

$$Z_c = R_s + j\omega L + \frac{1}{j\omega C} \quad \text{Eq. 3}$$

$$Z_c = R_s + j\left[\omega L - \frac{1}{\omega C}\right] \quad \text{Eq. 4}$$

The magnitude of the impedance is

$$Z_c = \sqrt{R_s^2 + \left[\omega L - \frac{1}{\omega C}\right]^2} \quad \text{Eq. 5}$$

At the series resonant frequency:

$$\omega L = \frac{1}{\omega C}$$

or,

$$\omega = \frac{1}{\sqrt{LC}}$$

At the resonant frequency, $Z_C = R_S$, which is the minimum impedance.

Figure 4 shows how the impedance varies with frequency. The series resistance usually increases as the capacitance decreases. Also, as the capacitance decreases, the inductance typically decreases, which means that the resonant frequency increases. This is usually due to the capacitor's physical construction. Note that a surface-mounted capacitor's lead inductance is at least an order of magnitude less than that of an axial-lead capacitor.

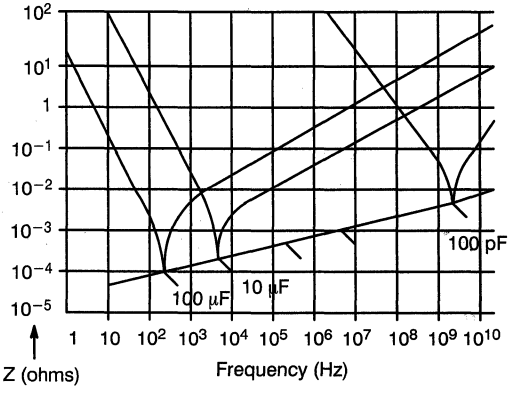


Figure 4. Capacitor Impedance Versus Frequency

The next step in high-frequency filter analysis is to determine a typical system's expected high-frequency components. Begin by assuming that the circuit is driven by a series of digital pulses with finite rise and fall times, then perform a Fourier transform on the series to determine their frequency components.

Fourier Transform of a Periodic Pulse

Figure 5 illustrates a periodic pulse of amplitude A, period T, rise and fall times of t_r, and pulse width of T_p, as measured between the 50-percent-amplitude points.

The approximate frequency-domain transform appears in *Figure 6*. The amplitude of the frequency-domain voltage is a function of the signal's amplitude and duty cycle in the time domain. The fundamental frequency, F_0, is related to the pulse train's period. The first harmonic, F_1, is of equal energy and is a function of the pulse width. The second

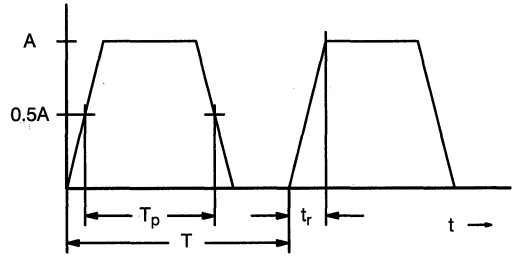


Figure 5. Periodic Pulse Waveform

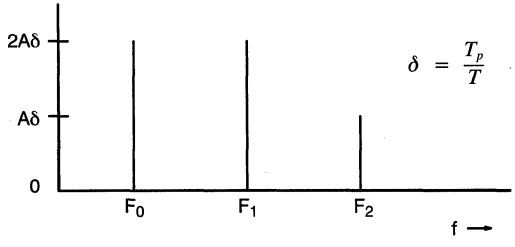


Figure 6. Fourier Transform of Periodic Pulse

harmonic, F_2, contains half the energy of F_0 and is a function of the pulse rise time.

The rise and fall times of Cypress's CMOS and BiCMOS circuits are 2 ns, by design. If a Cypress PLD is driving the write- or read-strobe inputs of a CY7C429–20 FIFO at the maximum frequency of 33.3 MHz (T = 30 ns) with a 10-ns/30-ns duty cycle signal (T_p = 10 ns), the following signal frequencies are generated:

$$F_0 = \frac{1}{\pi T} = \frac{1}{3.1416 \times 30 \times 10^{-9}} = 10.61\ MHz$$

$$F_1 = \frac{1}{\pi T_p} = \frac{1}{3.1416 \times 10 \times 10^{-9}} = 31.83\ MHz$$

$$F_2 = \frac{1}{\pi t_r} = \frac{1}{3.1416 \times 2 \times 10^{-9}} = 159.15\ MHz$$

Within the IC, signal rise and fall times can be as fast as 300 ps (picoseconds), which means that F2 = 1.061 GHz (1,061 MHz). In some ICs short timing pulses are generated internally, but they are usually longer than the 300-ps rise time, so the preceding F2 is the highest harmonic present.

Because the IC's data outputs can normally change no faster than those of the inputs, the outputs do not generate additional higher-frequency harmonics.

Parallel the Filter Capacitors

It will not be possible to find a capacitor with three series resonant frequencies that correspond to F0, F1, and F2. Instead, select one capacitor with a resonant frequency greater than 160 MHz and connect it in parallel with the decoupling capacitor, between V_{CC} and ground, as close to the IC as possible. It will act like a bandpass filter, shunting the unwanted, high frequency signals to ground. The sum of the values of the capacitors should be greater than or equal to the value of capacitance given by *Equation 2.*

The AVX Corporation, Myrtle Beach, South Carolina (803–448–9411), makes a series of "RF/Microwave NPO Capacitors." Their "Ultra Low ESR, 'U' Series" have an ESR of 0.06 Ohm at 500 MHz. A value of 470 pF in the EIA standard size 1210 "chipcap" is recommended. Its series resonant frequency is approximately 180 MHz.

Low-Frequency Filter Capacitors

A solid tantalum capacitor of 10 µF is recommended for every 50 to 100 ICs to reduce power-supply ripple. Place this capacitor as close as physically possible to where the V_{CC} and ground enter the PCB or module.

Using Decoupling Capacitors

Introduction

This application note describes some revised recommendations regarding the use of decoupling capacitors. The "conventional" recommendation of using two different values and two different types can, in many circumstances, cause less than ideal operation. Simpler, more reliable designs will often result from following the design guidelines of this note.

The Problem

Faster edges, more sensitive devices, higher clock rates all demand "good" decoupling of the power supplies.

Decoupling:
The art and practice of breaking coupling between portions of systems and circuits to ensure proper operation.

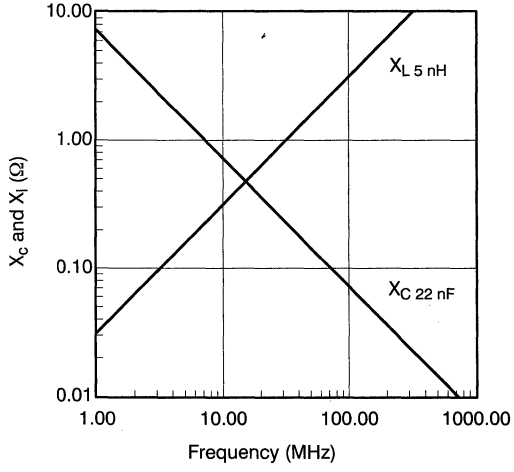


Figure 1. Z vs. f for Parts of a Real Capacitor

Bypassing:
The practice of adding a low-impedance path to shunt transient energy to ground at the source. Required for proper decoupling.

What used to work for lower system speeds and slower logic may not work well when the system speed increases. The common practice of using two different values for decoupling can:

- Increase the RFI/EMI problems
- Reduce the reliability of operation
- Reduce the noise tolerance

Each physical component shown on the schematic brings with it additional electrical components determined by the design and mounting of that component into the system.

Look in *Figure 1* at the behavior of two ideal components, a capacitor and an inductor representing parts of the capacitor shown in *Figure 2*. Note that without any lead inductance or resistance, the resulting capacitive reactance approaches 0Ω with increasing frequency. Note also that the inductive reactance of the ideal inductor, without any stray capacitance, approaches infinity.

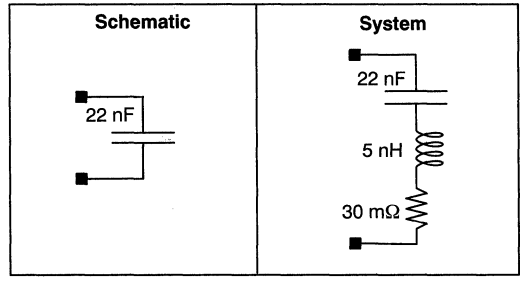


Figure 2. The "Real" Schematic

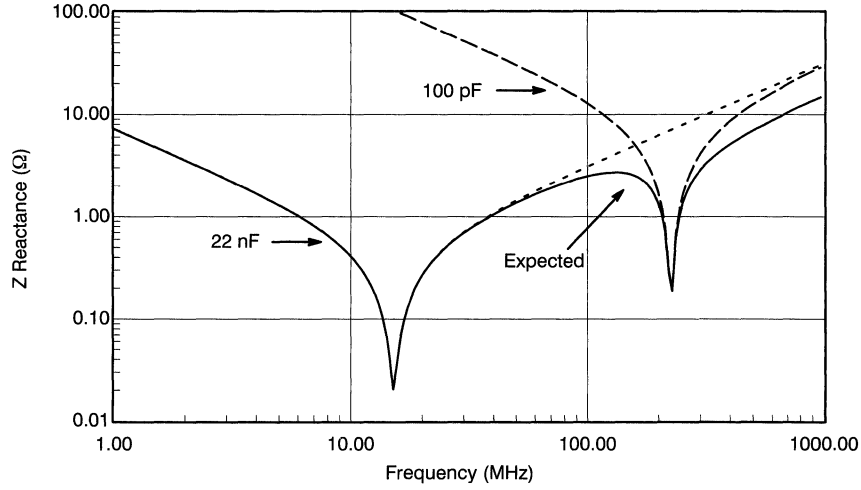


Figure 3. Expected Impedance of "Real" Capacitors

A real capacitor includes an inductor and resistor in the form of leads, traces, and even ground planes in series with it (*Figure 2*).

Multi-layer capacitors have approximately 5 nH of parasitic inductance when mounted on a printed circuit board. While the component drawn on the schematic (*Figure 2*) shows a 22-nF capacitor, the system sees the 22-nF capacitor in series with a 5-nH inductor and a 30-mΩ resistor.

The impedance curve of "Real" capacitors resembles the traces marked 22 nF and 100 pF of *Figure 3*. The shape of these calculated curves match the curves given in capacitor manufacturers' datasheets. This means that in a circuit, a capacitor acts as a low-impedance element only over a limited range of frequencies. A solution, proposed in many works, added a second capacitor to bypass frequencies outside the limited range of the single capacitor. This approach expected that the resulting impedance curve would look like the solid line marked "Expected" in *Figure 3*. This solution, however, has a significant problem at "intermediate" frequencies.

These intermediate frequency problems come from the circuit shown in *Figure 4*. The circuit on the left represents the schematic form of a typical decoupling arrangement, a 22-nF and a 100-pF capacitor in parallel.

Conventional wisdom suggests that the 100-pF should decouple the high frequencies, and the 22-nF should decouple the low frequencies. However, the combination results in some unexpected interactions. The circuit on the right in *Figure 4* shows a clearer representation of the system, including the parasitic inductances and resistances. This picture shows all the components necessary to create a resonant tank circuit.

Figure 5 shows a combined plot of Z vs. frequency of this circuit. The values given for effective series resistance (ESR; 30 mΩ) and effective series induc-

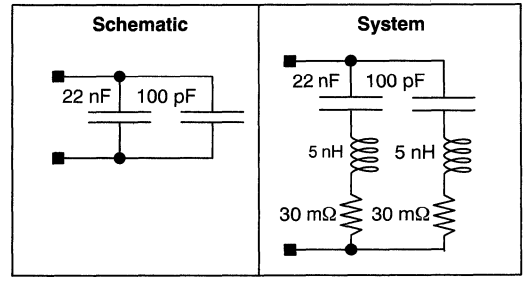


Figure 4. The "Real" Schematic

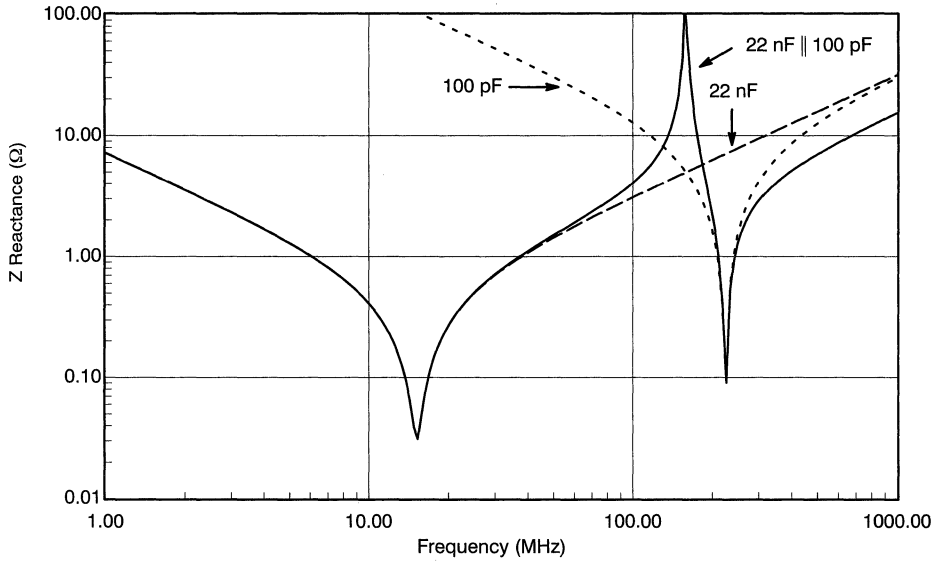


Figure 5. Real Z vs. f for Parallel 22-nF and 100-pF Capacitors

tance (ESL; 5 nH) are achievable on real PCBs using "good" layouts and surface-mounted capacitors.

The graph of *Figure 5* shows a range of frequencies where this combination of two capacitors results in a higher impedance than that of the larger capacitor alone. For the combination shown, this range includes approximately 15 MHz through 175 MHz. Notice the large peak in reactance at 150 MHz due to resonance of the two capacitors. Any energy from the rest of the system (ICs, clocks, and harmonics), over this intermediate range of frequencies, will see a higher impedance than that of a single 22-nF capacitor alone. Over this range of frequencies, the parallel combination will bypass less of the energy to ground.

The height of the peak shown in *Figure 5* varies inversely with the ESR of the capacitors. As board designs and components improve, the height of the resulting peak will actually increase due to a reduction of the system ESR. The exact shape and location of the parallel resonant peak will vary for each system depending on the design of the printed circuit board (PCB) and choice of capacitors.

Recommendations

The following recommendations can improve the resulting designs:

- Use only one value of capacitor.
- Choose the capacitor based on the self-resonant characteristics from the manufacturers' datasheet to match the clock rate or expected noise frequency of the design.
- Add as many capacitors as needed for your range of frequencies. As an example, the capacitor shown (22 nF) has a self resonant frequency of approximately 11 MHz, and a useful (less than 1Ω) impedance range of 6 to 40 MHz. Use as many of these as needed to achieve the desired level of decoupling.
- A minimum of one capacitor per power pin placed as physically close to the to the power pins of the IC as possible to reduce the parasitic impedances.
- Keep lead lengths on the capacitors below 1/4″ between the capacitor endcaps and the ground or power pins.

- Place the bypass capacitors on the same side of the PCB as the ICs. *Figure 6* shows an example of a recommended layout for a HOTLink™ Transmitter and Receiver.

A special note about *Figure 6*: in both of the layouts, only one connection is made to the V_{CC} plane. This is done so that the noise, generated both inside the IC and external to this portion of the circuit, must go through the single via to the power plane. The additional reactance of the via helps to keep the noise from spreading throughout the rest of the system.

HOTLink parts tolerate a fairly large amount of V_{CC} noise. However, to achieve the absolute "best" performance, use these recommendations.

What About Multiple Clocks?

When the design calls for multiple clock frequencies, split the power plane as shown in *Figure 6* and use the correct value of capacitor for each section, maintaining only one value per section. An example of this technique may be found in "HOTLink Design Considerations, Power Distribution Requirements for Optical Drivers." The isolation provided by the slotted power plane keeps the noise of one section away from the sensitive parts of the other sections, and allows the separation of the capacitor values.

What About Variable Clock Frequencies?

Bypassing ICs when the clock rate changes over a wide range of frequencies presents the most difficult situation covered here. Fortunately, most data communications applications use only a single clock rate.

When the range of operation of a single part covers a large range of frequencies, placing two capacitors that are within approximately 2:1 of each other in capacitance results in a wider low-impedance zone and allows a broad range of bypass frequencies. In *Figure 7* notice that the peak in the reactance still occurs, but that the maximum impedance stays well below 1.5Ω and that the usable range (less than 1.5Ω) now extends from approximately 3.25 MHz to 100 MHz. Use this multiple decoupling capacitor method only when a wide range of frequencies must be bypassed around a *single* integrated circuit and adequate range cannot be achieved by a single capacitor. Again, the capacitors must remain within a 2:1 range to prevent the reactance peak from exceeding useful limits.

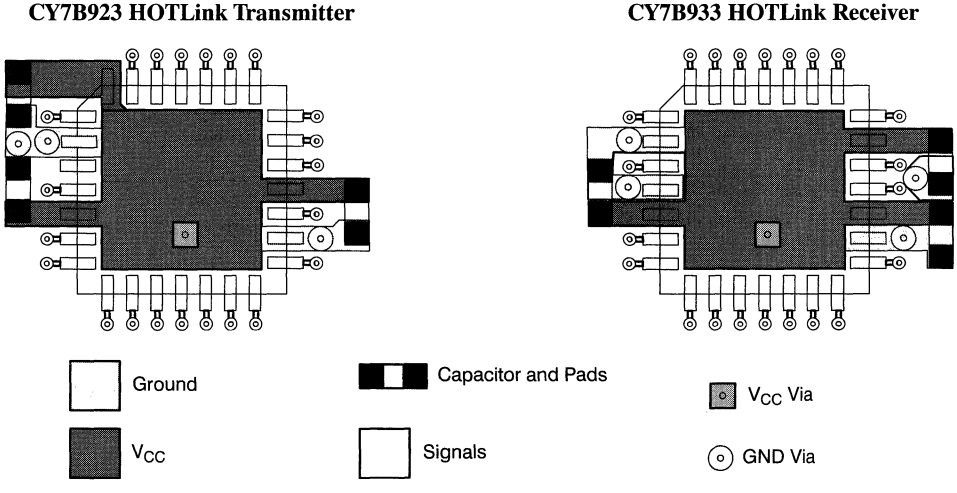


Figure 6. Sample Layouts

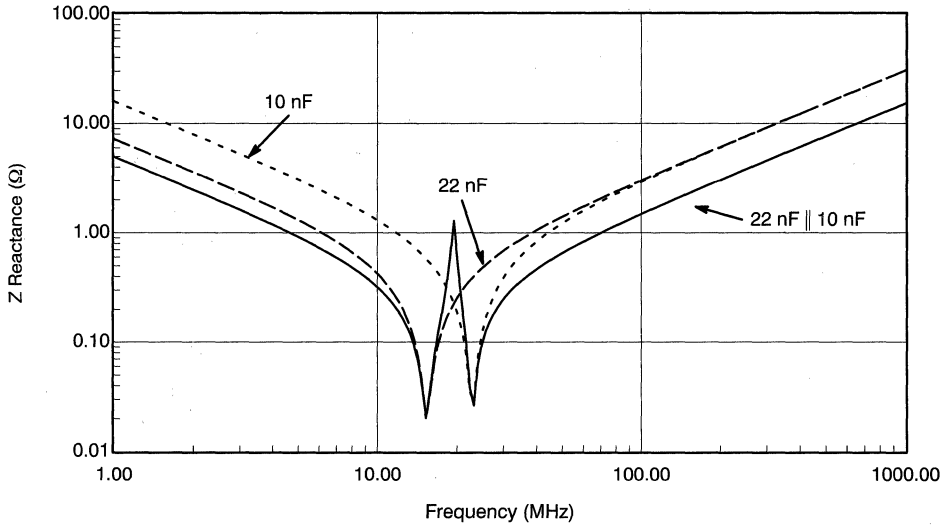


Figure 7. Real Z vs. f for Parallel 22-nF and 10-nF Capacitors

Conclusions

Application of these techniques resulted in improving the measured optical margin of a HOTLink-based OLC (optical link card) by about 1 dB. It simplifies the Bill of Material because only one value is used instead of two. Finally, using only one value of capacitor gave the best jitter measurements of the HOTLink Transmitter.

HOTLink is a trademark of Cypress Semiconductor.

SRAMs – 2

SRAMs Section Contents and Abstracts

This application note works through the design decisions that occur when an L2 cache is designed into an Intel 486–based system built with the Contaq PCI chipset. Then a design example shows how to use the CYM9246 family of L2 cache modules with the Contaq PCI chipset.

Using an L2 Cache Module with the Contaq 82C599 PCI Chipset for the Intel 486 CPU

Overview

Cypress Semiconductor markets the Contaq 82C599 PCI Chipset for Intel® 486-based systems. The Intel 486 CPU has an on-chip 8-Kbyte first level (L1) cache that significantly improves system performance. The Contaq PCI chipset includes an integrated high-performance cache controller for an external second-level (L2) cache.

This application note works through the design decisions that occur when an L2 cache is designed into an Intel 486-based system built with the Contaq PCI chipset. The questions that are addressed are:

- What are the cache requirements?
- Why use a cache module?
  — discrete vs. modular designs
- Which cache module(s)?
  — selecting an L2 cache module

L2 Cache Requirements

The L2 cache will be defined by size, speed, and type. There is also the matter of buffering the input address bits and providing chip select inputs to the data RAMs.

Cache Size

The current market requirement for L2 cache in 486-based systems is largely 128 Kbytes with an expansion option to 256 Kbytes. A small percentage of customers request 512 Kbytes. The larger 512-Kbyte cache size is considered useful in high-performance multiprocessing applications. The Contaq PCI chipset supports cache sizes from 32 Kbytes to 1 Mbyte.

Assume a nominal cache size of 128 Kbytes with an expansion option to 256 Kbytes.

In that case, the data RAMs can be a standard 32Kx8 device (e.g., CY7C199). The 128-Kbyte cache can be built with one bank of four 32Kx8 RAMs. The 256-Kbyte expansion option can be a second bank of four more 32Kx8 RAMs. With the Contaq PCI chipset, the 256-Kbyte cache can be configured as two interleaved banks.

Cache Speed

The cache should support zero-wait-state operation at a bus frequency of 33 MHz. That requires a tag RAM with an access time (t_{AA}) of 15 ns. The access time of the data RAMs depends on the organization. A single-bank array (128-Kbyte) should have t_{AA} = 20 ns. An interleaved two bank array (256-Kbyte) can use slower data RAMs with t_{AA} = 25 ns.

The Contaq PCI chipset also supports a 50-MHz clock option with one wait state (3222). In this mode, the tag RAMs can be slower with an access time of 20 ns. The data RAM access times are the same as noted above.

Assume two cache configurations at 33 MHz: a single-bank 128-Kbyte cache and a two-way interleaved 256-Kbyte cache. The tag RAM will have t_{AA} = 15 ns in either configuration. The 128-Kbyte cache will use 20-ns data RAMs and the 256-Kbyte cache can use lower cost 25-ns data RAMs.

Cache Type

The cache type can be either write-through or write-back. The Contaq PCI chipset supports both types of cache with an on-chip 8-bit address comparator and logic to process an optional dirty bit.

The Contaq PCI chipset has two write-back modes: 7-bit tag with one dirty bit or 8-bit tag without a dirty bit. In write-through mode, the chipset supports an 8-bit tag.

The type of cache and cache size affect the cacheable address range. With a 7-bit tag, one dirty bit, and 128 Kbytes of cache, the cacheable address range is 16 Mbytes. Increasing the cache size to 256 Kbytes doubles the cacheable address range to 32 Mbytes. With an 8-bit tag, no dirty bit, and 128 Kbytes of cache, the cacheable address range is 32 Mbytes. With 256 Kbytes of cache, the cacheable address range is 64 Mbytes.

Please note that although the system behavior is different for all three modes of operation, the external support hardware (8-bit tag RAM) is exactly the same. The tag RAM size is 8Kx8 for 128 Kbytes of cache and 16Kx8 for 256 Kbytes of cache.

Address Buffers for 128-Kbyte Cache

The single bank 128-Kbyte cache will require 15 bits of address ($A_{16:2}$). The upper 13 bits ($A_{16:4}$) from the 486 address bus are buffered through a pair of transparent latches (74FCT373C) to minimize the loading on the 486 address bus. The address latches are gated by the ALE signal from the CPU.

The lower two bits ($A_{3:2}$) are time critical for burst accesses and require special handling. To support the different memory configurations, these address inputs are driven by the Contaq PCI chipset. $TOGA_2$ from the chipset drives cache address A_2. $TOGA_3$ from the chipset drives cache address A_3.

The write enable ($\overline{CWE}_0$) and output enable ($\overline{CRD}_0$) signals for bank 0 from the Contaq PCI chipset are used to drive the write enable and output enable inputs to the data RAMs.

$TOGA_2$ drives RAM address bit A_0 and $TOGA_3$ drives RAM address bit A_1. The upper 13 bits of latched address ($LA_{16:4}$) are applied directly to the tag RAM address bits $A_{14:2}$.

The loading on the CPU address bus ($A_{16:4}$) is therefore limited to two loads (latch and tag RAM). The loading on the $TOGA_{3:2}$ outputs from the chipset is four loads (data RAMs). The ALE input from the 486 has two loads (latches).

Address Buffers for 256-Kbyte Cache

The address requirements for the interleaved two-bank 256-Kbyte cache are somewhat different. The upper 14 bits ($A_{17:4}$) from the 486 address bus are buffered through a pair of transparent latches (74FCT373C) to minimize the loading on the 486 address bus. The address latches are gated by the ALE signal from the CPU.

The lower two address bits ($A_{3:2}$) are provided by the chipset as $TOGA_2$ (address bit 3 for bank 0) and $TOGA_3$ (address bit 3 for bank 1). To support the two-way interleave, the Contaq PCI chipset provides separate write enables ($\overline{CWE}_0$ and $\overline{CWE}_1$) and output enables ($\overline{CRD}_0$ and $\overline{CRD}_1$) for each bank.

The address to bank 0 of the data RAMs is thus formed by $TOGA_2$ driving RAM address bit A_0 and latched address $LA_{17:4}$ driving RAM address bits $A_{14:1}$. The address to bank 1 of the data RAMs is formed by $TOGA_3$ driving RAM address bit A_0 and latched address $LA_{17:4}$ driving RAM address bits $A_{14:1}$.

The upper 14 bits of address ($A_{17:4}$) are applied directly to the tag RAM address bits $A_{13:0}$. The tag RAM is implemented as a 32Kx8 part, so the upper address bit A_{14} of the tag RAM is either grounded or tied to V_{CC}.

The loading on the CPU address bus ($A_{17:4}$) is two loads (latch and tag RAM). The loading on the $TOGA_{3:2}$ outputs from the chipset is four loads (data RAMs). The ALE input from the 486 has two loads (latches).

Generating Chip Selects $\overline{CS}_{3:0}$

The Contaq PCI chipset requires logic to combine the read/write signal (W/$\overline{R}$) and byte enables ($\overline{BE}_{3:0}$) from the Intel 486 to form the chip select

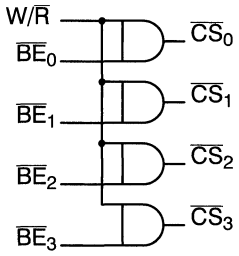


Figure 1. Chip Select Logic

($\overline{CS}_{3:0}$) inputs to the cache data RAMs as shown in *Figure 1*. A write cycle (W/$\overline{R}$=1) selects which byte(s) are written based on the byte enables ($\overline{BE}_{3:0}$). A read cycle (W/$\overline{R}$=0) selects all bytes for read independent of the byte enables.

This logic is typically implemented in a PLD (e.g., P16L8) to minimize the loading on the read/write line from the processor.

For a 128-Kbyte cache, each chip select input will go to one data RAM (one load). For a 256-Kbyte cache, each chip select will go to one data RAM per bank (two loads).

Discrete vs. Modular Designs

The L2 cache design that results from the discussion so far is shown in *Figure 2*. The questions now are how much (if any) of the L2 cache will be included on the motherboard and how much (if any) of the logic will be on a module.

Cypress Semiconductor supports either discrete or module-based designs:

- A wide range of 486 L2 cache modules for most popular chipsets
- High-speed SRAMs for tag and data RAMs
- FCT logic for the address buffers
- Fast PLDs for the chip select logic

The decision of a discrete vs. module-based design is usually based on flexibility, board space, and cost.

Flexibility

Implementing the L2 cache described in this paper as a module allows the customer to choose one of four configurations:

- No cache for lowest possible cost
- Low-cost 128-Kbyte cache
- Higher-performance 256-Kbyte cache
- Custom configuration (e.g., 512 Kbytes cache)

The modules under consideration for this application require a 112-position Burndy socket (part number CELP2X56SC3Z48). This socket is a high-quality, reliable socket that is a standard in the industry.

For contrast, a discrete implementation with the flexibility to support three of these configurations (no cache, 128 Kbytes, 256 Kbytes) would require sockets for the 9 RAMs in the cache design. These sockets would tend to reduce the reliability of the design. The FCT latches and PLD would usually not be socketed to improve the reliability for minimal cost.

In other words, a module-based design is much more flexible than an equivalent discrete design. Cache modules allow customers to tailor the cache to balance cost vs. performance tradeoffs to meet their requirements.

Board Space

The amount of board space required by a module-based design depends on how much of the required logic is on the module and how much is on the motherboard.

The minimum space occurs when all of the logic is on the module and the motherboard only has a 112-position socket with normal clearance around the socket (usually 0.1 inch). The section on "Selecting an L2 Cache Module" shows that this will not be the case. The chip select logic (one PLD—P16L8) will also be on the motherboard.

A discrete implementation will have nine 28-pin RAMs, two 20-pin latches, and one 20-pin PLD. It may also have sockets for at least the nine RAMs.

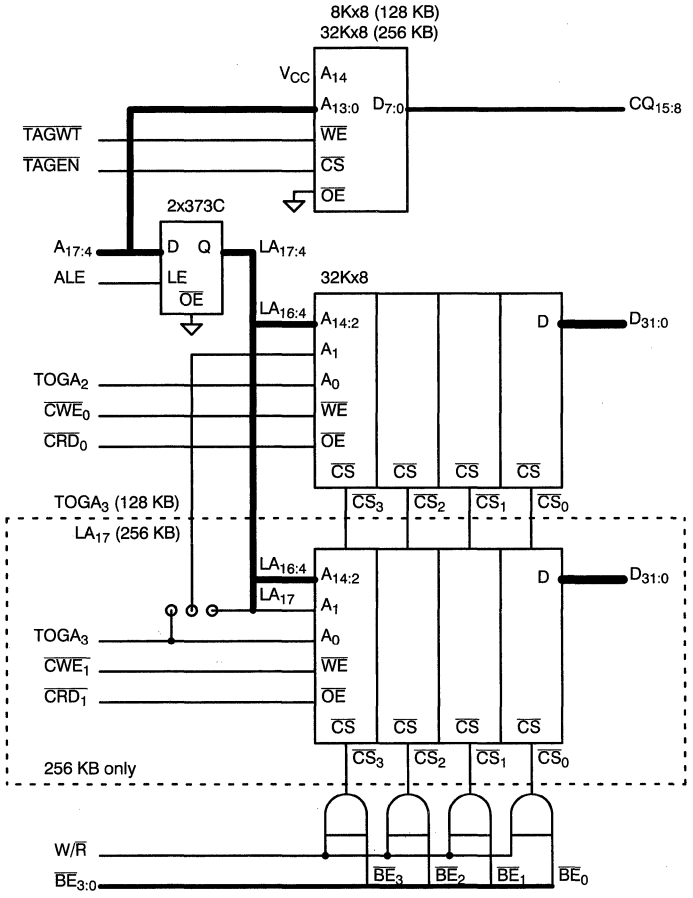


Figure 2. L2 Cache Design

The amount of board space required for a discrete design is significantly larger than the amount of space required for a module connector and a PLD. Therefore, a cache module design minimizes the amount of board space required on the motherboard.

Cost

The lowest-cost module option (no cache) requires one 112-pin socket and one 16L8 PLD. This should cost less than two 373 latches, one PLD, and nine 28-pin sockets.

A discrete 128-Kbyte cache will consist of two 373 latches, one PLD, one 8Kx8 RAM, four 32Kx8 RAMs and four 28-pin sockets. The 128-Kbyte cache module will be the same with a 112-pin socket plus a printed circuit board (substrate) minus the four 28-pin sockets. Module vendors will also add a profit margin to the cost of the module. As a result, a 128-Kbyte cache module will usually cost more than an equivalent discrete design.

For a 256-Kbyte cache, the cache module has the same components as the discrete design with the addition of a 112-pin connector, substrate, and ven-

dor margin. The 256-Kbyte module usually will cost more than an equivalent discrete design.

Selecting an L2 Cache Module

Cypress Semiconductor currently builds 8 different 486 compatible L2 cache modules in a total of 17 configurations. The question is which module is closest to the cache design described in this paper for the Contaq PCI chipset. The criteria are:

- 128/256 Kbytes data RAM
- 8-bit tag RAM
- No dirty RAM
- Address latches gated by ALE as opposed to address buffers
- Bank write enables as opposed to write enables for each chip
- Four chip selects as opposed to bank selects

The winner is the CYM9246/CYM9247/CYM9248 family of cache modules! These modules are very close to the requirements outlined in this paper with the following design considerations:

- The chip select logic resides on the motherboard, instead of the module.
- The Contaq PCI chipset does not require a dirty RAM separate from the tag RAM.
- The $TOGA_{3:2}$ address outputs to the module will require a strap on the motherboard.
- The $\overline{TAGOE}$ input to the module should be grounded on the motherboard.
- The $\overline{DIRTYCS}$ and $\overline{DIRTYWE}$ module inputs should be connected to V_{CC} on the motherboard.
- The signal naming conventions are different.

With regards to the dirty RAM, the customer has two choices:

- Tie the dirty RAM control signals inactive (V_{CC}) on the motherboard and ignore the dirty RAM.
- Ask Cypress to ship the module without the dirty RAM at a reduced cost.

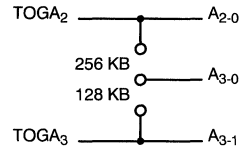


Figure 3. Address Straps

The $TOGA_{3:2}$ address outputs from the Contaq PCI chipset do not quite match the address inputs to the module and will require the strap logic shown in *Figure 3* on the motherboard.

Please refer to *Table 1* for a signal name cross reference between the Contaq PCI chipset and the CYM9246 cache module family.

Table 1. Signal Name Cross Reference

Contaq PCI Chipset	924X Module Family
$\overline{TAGWT}$	$\overline{TAGWE}$
$\overline{TAGEN}$	$\overline{TAGCS}$
$\overline{CWE}_{1:0}$	$\overline{WE}_{1:0}$
$\overline{CRD}_{1:0}$	$\overline{OE}_{1:0}$
$CQ_{15:8}$	$TAG_{7:0}$
$TOGA_2$	A_{2-0} (128 KB only) A_{3-0} (256 KB only)
$TOGA_3$	A_{3-0} (128 KB only) A_{3-1} (256 KB only)

Summary

The CYM9246 family of L2 cache modules can be designed into an Intel 486 system based on the Contaq PCI chipset. By adding a 112-pin DIMM connector, a P16L8, and a two-position jumper strap to the motherboard design, the customer can offer:

- A lowest-possible-cost option with no cache
- A low-cost performance upgrade with a single bank 128-Kbyte cache module (CYM9246)
- A higher-performance upgrade with a two-way interleaved 256-Kbyte cache module (CYM9247)
- Upgrades to larger cache modules such as the CYM9248 (512-Kbyte)

PROMs/EPROMs – 3

PROMs/EPROMs Section Contents and Abstracts

This application note introduces PROMs to the user and then explains the methods of generating PROM programming files. A brief description of PROM usage in systems is presented followed by a discussion of various PROM programming file formats, including the Intel, Motorola, DEC, and Tektronix formats. Finally, the application note discusses various methods of using high-level languages (ABEL HDL, ISDATA, LOG/iC HDL, BASIC, C) to generate PROM programming files.

This application note discusses how to use the CY7C276 PROM as program memory for various DSPs. It will cover the topic of interfacing the CY7C276 high-speed PROM to some of today's most popular DSPs for program memory only. Data memory storage is typically done with SRAM and its interface is not included in this application note. The AT&T DSP1616, Analog Devices ADSP 2100A, Motorola DSP56000 and TI TMS320C5x family of devices are discussed. Also included is a detailed description of the CY7C276 (including architecture, programming options, and signal descriptions) and brief descriptions of the DSPs (architectures, signals and timing requirements). For ease of explanation, only one example from each product family is included. The other devices in each product family are similar and are left as an exercise for the reader. Detailed timing calculations that show code sizes up to 16K words in depth are included in the examples. Finally, a table is provided to help summarize the analysis.

This application note describes how to use a Cypress CY27H010 1-Megabit PROM with the Rockwell V.FAST chipset to create a high performance fax/modem running with 0 wait states.

This application note describes a method for interfacing a high-speed 5V Cypress PROM to a 3.3V system. The I/O level translation is achieved using a CYBUS3384 Bus Switch.

Generating PROM Programming Files

PROMs are nonvolatile memory devices that were first conceived as instruction and data storage devices for microprocessor systems. Since their introduction, PROMs have benefited from improvements in processing and manufacturing technology. The evolution of PROMs has included a tremendous increase in their density and speed and has added new features such as built-in registers and reprogrammability. Now these devices can be used in a wide variety of applications other than instruction storage. PROMs are commonly found in state machines, decoders, encoders, complex counters, controllers, sequencers, and look-up tables as well as in their traditional role of instruction or microcode storage.

PROMs are simply an array of data coupled with an input address decoder. The address presented to the device drives a simple 1-of-n decoder. The decoder selects one preprogrammed memory location whose data flows to the output pins of the device. PLAs (Programmable Logic Array) and PALs (Programmable Array Logic) are also programmable devices and, along with PROMs, make up the majority of devices that are considered to be programmable logic elements. The difference between the three types of programmable logic elements can be seen by observing the internal structure of the programmable array of each of the devices. PLAs have both a programmable "AND" array and a programmable "OR" array. PALs have a similar AND-OR structure, but the number of inputs to the OR function is fixed, so only the AND array is programmable. Both the PLA and PAL have a fixed number of AND-OR terms dedicated to each output. Therefore, the number of functions controlling each output is significantly reduced. PROMs, on the other hand, can realize every possible combination or function of n input lines for a given output. There are 2^n product terms (where n = number of address lines) per PROM output. This makes PROMs useful in very complex functions that exhaust the sum-of-product resources of a traditional PAL or PLA architecture. Some PROMs have additional features, such as output registers, that enable them to operate synchronously, which is required for state machines. The Cypress CY7C245A is one of these PROMs. Presets, clears, and initialization words are also available for dealing with power-on and reset conditions.

After understanding the basic function of a PROM, the designer must now create the PROM data in the form of a programming file. Creating the PROM data can be intimidating to engineers who are not familiar with the process. Looking back, we can see that PROMs were mainly used for instruction or microcode storage in a microprocessor or bit-slice-based system. Therefore, the PROM data for such systems is generated by the compilers, assemblers, and linkers that are resident on the CPU development station or emulator. Generating the PROM files for such systems is almost trivial because the programming data file is simply a listing of the CPU's executable instructions generated by the compiler. But creating the programming file for a complex decoder, look-up table, sequencer, or state machine can be pretty complicated and overwhelming. In fact, just figuring out where to start or what tools to use can become very time consuming. In this brief application note we will discuss the structure of PROM data files and show several ways to create them. Examples using simple languages such

as C and BASIC, as well as PLD development tools such as ABEL and LOG/iC, will be discussed.

In order to understand how to create programming files, you must first be familiar with the actual structure or format of such a file. Again, a PROM is simply an array of programmable memory locations. The data file that is transmitted to the PROM programmer must therefore contain data for each of the locations to be programmed. There are many standard formats for PROM data files.

Generic PROM programmers, such as those manufactured by Data I/O, Stag, Logical Devices, Digelic, SMS, and Kontron, are generally compatible with the following formats:

- ASCII—HEX (Space)
- Binary
- DEC Binary
- Motorola Exorciser
- Motorola Exormax
- Intel "Intellec" 8/MDS
- Intel MCS86 "intellec 86"
- Tektronix "HEX"
- Extended Tektronix "HEX"

The following section describes each format in detail. Each format has its own set of required fields, delimiters, and special characters. When writing code in C or BASIC, you must know exactly where to place each field and special character so that a programmer will interpret your data correctly.

ASCII–HEX (Space)

One of the simplest and probably the most universal file formats is HEX or HEX-Space ASCII. This format does not support checksum or address field conventions. Therefore, the data in the file must be in order incrementing from address 0. However, many times the program that reads the file into programmer memory can manipulate the data to start at any address location.

Three hidden instructions are used in this format:

1. ASCII STX Character (ASCII 02) marks the beginning of the file.
2. ASCII ETX Character (ASCII 03) marks the end of the file.
3. ASCII Space (ASCII 20) is between each data byte.

Figure 1 shows a data file for a 64-byte PROM implemented in ASCII–HEX (space) format.

Note that each data byte is separated by a "space" character and that no addressing information is present.

ASCII Binary

ASCII Binary files, like ASCII–HEX, contain no addressing or checksum information. ASCII Binary allows for very fast file transfers to the programmer due to its simplicity. The data format begins with the ASCII STX character and is terminated by an ETX. Data is grouped into four-byte lines separated by a space. Each line of data begins with a "B" character and ends with an "F" character.

Figure 2 shows a 64-byte PROM file containing all zeros using ASCII Binary format. All data is loaded into the PROM sequentially starting at location 0.

Simple Binary

The simple Binary format consists of just binary data. There are no start or end characters. Although the binary file is simple to produce, it is not a recommended output format for the following examples because binary files cannot be easily read by text editors.

```
(STX)FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF(ETX)
```

Figure 1. ASCII–HEX Format

```
(STX)  B00000000F B00000000F B00000000F B00000000F
       B00000000F B00000000F B00000000F B00000000F
       B00000000F B00000000F B00000000F B00000000F
       B00000000F B00000000F B00000000F B00000000F(ETX)
```

Figure 2. ASCII Binary Format

DEC Binary

DEC Binary is a modification of the basic ASCII Binary file format. DEC Binary adds a starting address and a checksum for each line of data.

Motorola Exorcisor

Motorola Exorcisor is one of the most widely used formats. Motorola Exorcisor files are commonly referred to as "S" records because each line starts with an "S" followed by the record type. Each line also contains a byte count, starting address, and a checksum, which are delineated by carriage returns and line feeds.

Figure 3 shows an example of a 64-byte PROM file implementing "S" Records.

Calculating Record Checksum

The Checksum is calculated by first stripping off the start code ("S"), the record type, and the checksum. The remaining bytes are added together, converted to binary, and complimented (one's compliment). For example, the optional sign on "S0" line reads:

S0 06 00 01 00 01 F7

Stripping the appropriate characters leaves:

06 00 01 00 01

Adding the bytes yields

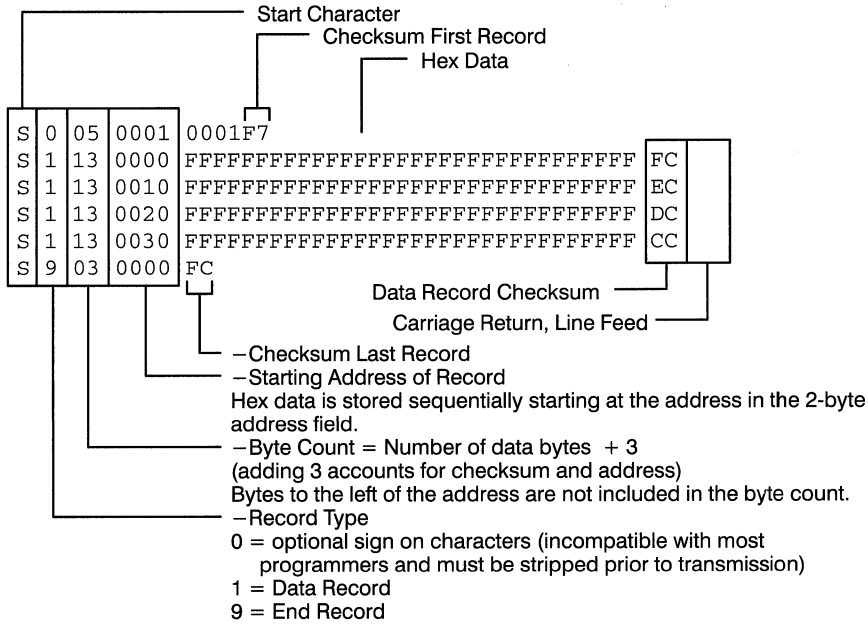


Figure 3. S Record Format

08 hex

The compliment of the value

F7..... Record checksum

End of Each Record

It is important to end each record with a carriage return and a line feed, which is used as a delineator.

"S" records are useful because they are so universal. However, this format can only be used for PROMs smaller than 64 Kbytes because the address field is limited to 4 bytes.

Motorola Exormax

Exormax is another "S" record file and is identical to Exorcisor with only one exception. Exormax allows for a 6-digit address field, which makes it useful for PROMs that are much larger than 64 Kbytes.

Exormax Record Number:

S0– Optional sign on record
S1– Data record (2 Byte Address field)
S2– Data Record (3 Byte Address Field)

Figure 4 shows an example of a 64-byte PROM file implementing "Exormax S" records.

Intel "Intellec" 8/MDS

Intellec is similar to S records in that each line contains a starting address, byte count, and checksum. However, each line begins with a colon.

Intellec Record Example:
":", Byte Count, Address, Record Type, Data, Checksum

Byte Count: Total number of data bytes ONLY.

Starting Address: 2-byte field where record will be placed in memory.

Record Type:
00 — Data Record
01 — End Record

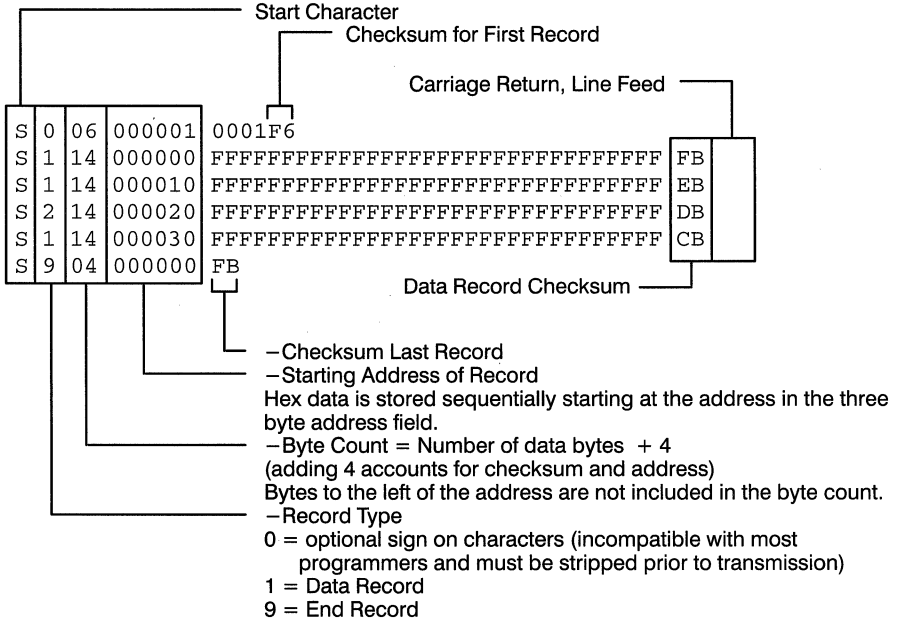


Figure 4. Exormax S Format

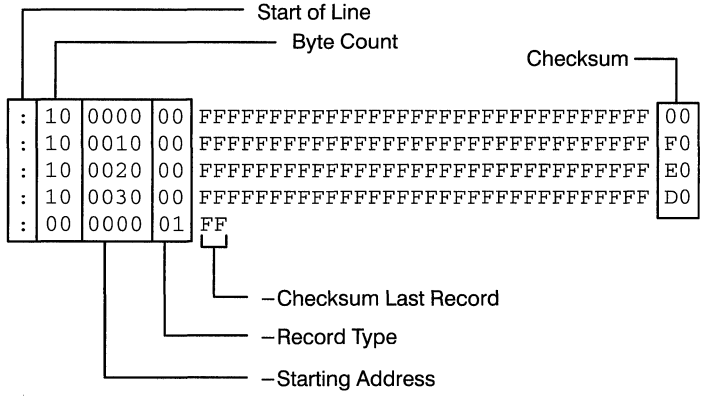


Figure 5. Intellec Format

Checksum: The sum of all preceding bytes including byte count, Address, and all data bytes. This number is expressed in two's compliment notation.

The end of each record is marked by a carriage return.

Figure 5 shows a 64-byte PROM file using Intellec format.

Since there is only a 2-byte address field, Intellec is generally used for PROMs smaller than 64 Kbytes.

Intel MCS86 (Intellec 86)

Intellec 86 is an extension of the standard Intellec format. It adds the feature of a Segment Base Address record (SBA). Adding the SBA to the 2-byte address field increases the total addressing capability to 1M locations. The file must begin with an SBA record because physical addresses are calculated using the Starting Address field and the most recent SBA.

Intellec 86 Data or End Record Example:
":", Byte Count, Address, Record Type, Data, Checksum

Intellec 86 SBA Record Example:
":", Byte Count, Address, Record Type "02", SBA, Checksum

Byte Count: Total number of data bytes ONLY.

Segment Base Address (SBA): A 2-byte field that extends the starting address fields of the following records by 4 bits. A new SBA can be inserted as many times as needed. Records sent after a new SBA will use the new SBA to calculate the address.

Starting Address: A 2-byte field where record will be placed in memory. The actual physical address for data placement must be calculated by using the SBA and Starting Address.

Record Type:
00 — Data Record
01 — End Record
(02 — SBA Record)

Checksum: The sum of all preceding bytes including byte count, address, and all data bytes. This number is expressed in two's compliment notation.

The end of each record is marked by a carriage return and line feed.

Figure 6 shows a 64-byte PROM file using Intellec 86 format. This example has an SBA Value of 8000h, which offsets the starting addresses as shown.

To calculate the starting address: (third data record)

Take the value of the most recent SBA	(8000h)
Shift the SBA left	8000
Add value of start address field	+ 0030
Result: Physical Start Address	80030

```
:02 0000 02 8000 7C
:10 0000 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF 00
:10 0010 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF F0
:10 0020 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
:10 0030 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF D0
:00 0000 01 FF
```

Figure 6. Intellec 86 Format

Again, the segment base address can be updated at any time and will affect the records that follow the change.

Tektronix Hex (TEK HEX)

TEK HEX is another simple file format that is accepted by most programming systems. It uses the "/" character as a start-of-record marker and includes a starting address for each record, byte count, data and two checksums. The first checksum is the summation of the bytes for the address and byte count fields. The second checksum is simply the summation of all of the data bytes. *Figure 7* shows an example of a file stored in TEK HEX format.

Start Character: "/" is used to mark the beginning of each line. Most programmers ignore any characters sent before the "/".

Start Address: This value is a 2-byte absolute address. It represents the starting address for the first data byte in the record. All following bytes in the record are stored sequentially.

Byte Count: The number of data bytes in the record are represented by the byte-count field. The end of record is marked by setting the byte count equal to "00".

First Checksum: The simple summation of the nibbles in the address and byte-count fields are represented by the first checksum in each record.

Second Checksum: Calculated by summing all of the nibbles of the data bytes in the record, the second checksum is placed at the end of the record.

Each record is terminated by a carriage return/line feed.

Extended Tektronix Hex (XTEK)

XTEK is a variation of the standard TEK HEX format. It uses the "/" character as a start of record marker and includes a starting address for each record, byte count, data, and two checksums. The first checksum is the summation of the nibbles for the address and byte-count fields. The second checksum is simply the summation of all of the data nibbles. *Figure 8* shows an example of a file stored in XTEK format.

Start Character: "/" is used to mark the beginning of each line. Most programmers ignore any characters sent before the "/".

Start Address: This value is a 2-byte absolute address. It represents the starting address for the first data byte in the record. All following bytes in the record are stored sequentially.

```
/0000 10 01 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0010 10 02 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0020 10 03 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0030 10 04 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0000 00 00
```

Figure 7. TEK HEX Format

Byte Count: The number of data bytes in the record are represented by the byte-count field. The end of record is marked by setting the byte count equal to "00".

First Checksum: The simple summation of the nibbles in the address and byte count fields are represented by first checksum in each record.

Second Checksum: The summation of the data nibbles is represented by the second checksum, which is placed at the end of the record.

Each record is terminated by a carriage return/line feed.

Using High-Level Languages to Create Files

Depending on the application, there are many different ways to create the actual file. PROMs that contain data derived from mathematical formulas such as look-up tables are easily implemented using a high-level language such as C or BASIC. These languages can easily deal with the complicated data types and mathematical data manipulation that is required for many applications. The data created by the program must be stored in a file so that it can be transferred to a programmer at some later time. The following examples show that opening a new file and writing the data is simple when using high-level languages.

As shown in *Figure 9*, this method written in C follows the simple form:

1. Header documentation – The Header documentation is usually written as comments to help the user understand the purpose and flow of the program. Documentation is not essential, but it is good practice.
2. Constant declaration – Following the Header, the constants can be declared as symbols to help the user update the program to accommodate changes in the design.
3. Variable definition – The variables should be defined to agree with the type of data being used.
4. Body – The body of the program will contain the commands necessary to create the PROM data. This usually takes the form of an outer "For"-type loop to iteratively step through all the possible combinations of address inputs, followed by nested commands that create a data instance to correspond to that combination of address lines.

The C program in *Figure 9* generates ASCII-Space or HEX-Space format output files for downloading to a PROM programmer.

Figure 10 is an example of using BASIC to produce a PROM programming file in the HEX space format.

PLD Development Packages

In general, most of the standard PLD development packages support PROMs. ABEL, CUPL, and LOG/iC are three of the most popular third-party packages. They support most of the industry standard PAL, PLA and PROM devices. These PLD development tools are well suited to creating PROM files that can be described by Boolean equations, truth tables, or state-machine syntax.

ABEL

ABEL, produced by Data I/O Corporation, is one of the most popular PLD development software packages on the market. The fact that ABEL supports PROMs is one of the industry's best-kept secrets. Since a PROM can be thought of as a PLD with a large number of product terms per output, it is relatively easy for a PLD compiler to generate code for a PROM. In fact, the source file (filename.abl) for

```
% 1A 6 06 4 1000 FFFFFFFFFFFFFFFF
% 1A 6 0E 4 1008 FFFFFFFFFFFFFFFF
% 1A 6 07 4 1010 FFFFFFFFFFFFFFFF
 % 0A 8 16 4 0000
```

Figure 8. XTEK Format

```
/* Example Program 1 */

/* The purpose of this program is to create a data file that could be
used as a COSINE look-up table.  The table has an angular resolution of
256 points per period and an amplitude resolution 256 steps or 8 bits.
*/


#include <stdio.h>        /* defines the input-output of PC */
#include <math.h>         /* defines the math package of PC */

int i,j;                  /* integers for loop variables */
float y,x,z;              /* floating pt variables for COSINE */
int data;                 /* data  variables for result */
int outfile;

main()
/* main denotes the start of the active part of the program */
{
  FILE *outfile;
/* makes outfile a pointer to the output file */

  outfile=fopen("promfile","w");
/* opens the output file for writing */

  fprintf(outfile,"%c",2);

/* prints control data to output file for download STX */

/* This section consists of 2 nested loops to generate every possible
combination of address inputs.  An incrementing variable z is used to
generate the angle y in radians.  x = the cosine of y.  Then x is justi-
fied to use the dynamic range of 256 states.  The result is stored as an
integer in data.  The data is written directly to the output file.  The
data is broken into blocks for easier reading.  */
  z=0;
  for(i=0;i<=15;i++)  {
    for(j=0;j<=15;j++)  {
      y=M_PI*((z)/128.0);
      x=(cos(y));
      x=x*127.99999;
      data = x+128;
      fprintf(outfile,"%02X ",data);
      z=z+1.0;
    }
    fprintf(outfile,"\n");
  }
  fprintf(outfile,"%c",3);
/* prints control char ETX to output file */

  fclose(outfile);
/* closes output file */

}
```

Figure 9. C Program to Generate ASCII-Space or HEX-Space Format Files

```
10 'Example program 2
20 '
30 'The purpose of this program is to create a data file
40 'that could be used as a COSINE look-up table.  The table
50 'has an angular resolution of 256 points per period and
60 'an amplitude resolution of 256 steps or 8 bits.
70 '
80 PI = 3.14159
90 OPEN "O",#1,"PROMFILE.HEX"    'open the file for output
100 '
110 'This section consists of 2 nested loops to generate every
120 'possible combination of address inputs.  An incrementing
130 'variable z is used to generate the angle y in radians.
140 'X = cosine of y.  Then X is justified to use the dynamic
150 'range of 256 states.  The result is stored as an integer
160 'in RANGE.  The data is written directly to the output file
170 'in the HEX SPACE format.
180 '
190 PRINT#1,CHR$(2)      'start the file with the STX char
200 Z = 0                'initialize the loop
210 FOR I = 0 TO 15
220     FOR J = 0 TO 15
230         Y = PI*((Z)/128)
240         X = COS(Y)
250         RANGE = INT(X*127.99999# + 128)
260         IF RANGE > 15 THEN 290
270         PRINT#1,"0";HEX$(RANGE);" ";
280         GOTO 300
290         PRINT#1,HEX$(RANGE);" ";
300         Z = Z + 1
310     NEXT J
320     PRINT#1,""
330 NEXT I
340 PRINT#1,CHR$(3);     'end the file with the ETX char
350 CLOSE
360 END
```

Figure 10. BASIC Program to Generate HEX-Space Format

a PROM and a PLD are almost identical. The only difference is in the device declaration. In the logic diagram package for ABEL, there are pin descriptions for 4-, 8-, and 16-bit PROMS.

Figure 11 shows how to use truth tables and equations to generate a PROM file that is a comparator with some additional built-in logic.

All methods of generating PLD files in ABEL are also available for generating PROM files.

```
module COMP_OR
title '4 bit comparator '
"

     PROMB device     'RA8P8';
"                        8 address lines and 8 data lines

"INPUTS           PIN #    PROM ADDRESS/DATA BIT

     A0          PIN  1;   " A0
     A1          PIN  2;   " A1
     A2          PIN  3;   " A2
     A3          PIN  4;   " A3
     B0          PIN  5;   " A4
     B1          PIN 17;   " A5
     B2          PIN 18;   " A6
     B3          PIN 19;   " A7

"OUTPUTS          PIN #

     AGB         PIN 14;   " D8    A IS GREATER THAN B
     ALB         PIN 13;   " D7    A IS LESS THAN B
     EQUAL       PIN 12;   " D6    A IS EQUAL TO B

     ALL_HIGH    PIN 11;   " D5    ALL BITS ARE HIGH
     OR_BITS_3   PIN  9;   " D4    Misc.  logical functions
     OR_BITS_2   PIN  8;   " D3
     OR_BITS_1   PIN  7;   " D2
     OR_BITS_0   PIN  6;   " D1

     X = .X.;

Declarations
     A_NIB  = [A3,A2,A1,A0];
     B_NIB  = [B3,B2,B1,B0];

Equations
  ALL_HIGH  = (A_NIB==15) & (B_NIB==15);
  OR_BITS_3 = A3 # B3;
  OR_BITS_2 = A2 # B2;
  OR_BITS_1 = A1 # B1;
  OR_BITS_0 = A0 # B0;
```

Figure 11. Using Truth Tables and Equations in ABEL to Generate a Comparator PROM File

```
truth_table
     ([A3,A2,A1,A0,  B3,B2,B1,B0]->[AGB,ALB,EQUAL])
      [ 0,  0,  0,  0,   0,  0,  0,  0]->[ 0  ,  0  ,  1];"A = B CONDITIONS
      [ 0,  0,  0,  1,   0,  0,  0,  1]->[ 0  ,  0  ,  1];
      [ 0,  0,  1,  0,   0,  0,  1,  0]->[ 0  ,  0  ,  1];
      [ 0,  0,  1,  1,   0,  0,  1,  1]->[ 0  ,  0  ,  1];
      [ 0,  1,  0,  0,   0,  1,  0,  0]->[ 0  ,  0  ,  1];
      [ 0,  1,  0,  1,   0,  1,  0,  1]->[ 0  ,  0  ,  1];
      [ 0,  1,  1,  0,   0,  1,  1,  0]->[ 0  ,  0  ,  1];
      [ 0,  1,  1,  1,   0,  1,  1,  1]->[ 0  ,  0  ,  1];
      [ 1,  0,  0,  0,   1,  0,  0,  0]->[ 0  ,  0  ,  1];
      [ 1,  0,  0,  1,   1,  0,  0,  1]->[ 0  ,  0  ,  1];
      [ 1,  0,  1,  0,   1,  0,  1,  0]->[ 0  ,  0  ,  1];
      [ 1,  0,  1,  1,   1,  0,  1,  1]->[ 0  ,  0  ,  1];
      [ 1,  1,  0,  0,   1,  1,  0,  0]->[ 0  ,  0  ,  1];
      [ 1,  1,  0,  1,   1,  1,  0,  1]->[ 0  ,  0  ,  1];
      [ 1,  1,  1,  0,   1,  1,  1,  0]->[ 0  ,  0  ,  1];
      [ 1,  1,  1,  1,   1,  1,  1,  1]->[ 0  ,  0  ,  1];

      [ 0,  0,  0,  0,   X,  X,  X,  1]->[ 0  ,  1  ,  0];"A < B CONDS.
      [ 0,  0,  0,  X,   X,  X,  1,  X]->[ 0  ,  1  ,  0];
      [ 0,  0,  X,  X,   X,  1,  X,  X]->[ 0  ,  1  ,  0];
      [ 0,  X,  X,  X,   1,  X,  X,  X]->[ 0  ,  1  ,  0];
      [ 1,  0,  X,  X,   1,  1,  X,  X]->[ 0  ,  1  ,  0];
      [ 1,  0,  0,  X,   1,  0,  1,  X]->[ 0  ,  1  ,  0];
      [ 1,  0,  0,  0,   1,  0,  0,  1]->[ 0  ,  1  ,  0];
      [ 0,  1,  0,  X,   0,  1,  1,  X]->[ 0  ,  1  ,  0];
      [ 0,  1,  0,  0,   0,  1,  X,  1]->[ 0  ,  1  ,  0];
      [ 0,  0,  1,  0,   0,  0,  1,  1]->[ 0  ,  1  ,  0];

      [ 1,  X,  X,  X,   0,  X,  X,  X]->[ 1  ,  0  ,  0];"A > B CONDS.
      [ 0,  1,  X,  X,   0,  0,  X,  X]->[ 1  ,  0  ,  0];
      [ 0,  0,  1,  X,   0,  0,  0,  X]->[ 1  ,  0  ,  0];
      [ 0,  0,  0,  1,   0,  0,  0,  0]->[ 1  ,  0  ,  0];
      [ 1,  1,  X,  X,   1,  0,  X,  X]->[ 1  ,  0  ,  0];
      [ 1,  0,  1,  X,   1,  0,  0,  X]->[ 1  ,  0  ,  0];
      [ 1,  0,  0,  1,   1,  0,  0,  0]->[ 1  ,  0  ,  0];
      [ 0,  1,  1,  X,   0,  1,  0,  X]->[ 1  ,  0  ,  0];
      [ 0,  1,  0,  1,   0,  1,  0,  0]->[ 1  ,  0  ,  0];
      [ 0,  0,  1,  1,   0,  0,  1,  0]->[ 1  ,  0  ,  0];

 end  COMP_OR
```

Figure 11. Using Truth Tables and Equations in ABEL to Generate a Comparator PROM File (continued)

LOG/iC

LOG/iC by ISDATA probably has the best support of PROM devices due to its ability to create a PROM file of any size. All the programmer has to do is to tell the compiler how many inputs and how many outputs the PROM should have. The above ABEL file is reproduced in *Figure 12* using LOG/iC.

Although not illustrated in the last two examples, both ABEL and LOG/IC are capable of using state machine input formats.

```
*IDENTIFICATION
 This example uses an 8 bit prom as a 4 bit comparator and does some
 additional misc.  logic

*X-NAMES                    !Define the input pins.
 B[3..0], A[3..0];          !Pins are defined MSB first,
                            !Therefor, B3 will be connected
                            !to address bit 7 and A0 will
                            !be connected to address bit 0.
*Y-NAMES                    !Define the output pins
 AGB,ALB,EQUAL,ALL_HIGH,
 OR_BITS[3..0];

*BOOLEAN EQUATIONS
 ALL_HIGH = B3&B2&B1&B0&A3&A2&A1&A0;
 OR_BITS3 = A3 # B3;
 OR_BITS2 = A2 # B2;
 OR_BITS1 = A1 # B1;
 OR_BITS0 = A0 # B0;

*FUNCTION-TABLE
$ ((A3,A2,A1,A0, B3,B2,B1,B0)):((AGB,ALB,EQUAL));
     0  0  0  0   0  0  0  0  :   0   0    1; A=B CONDITIONS
     0  0  0  1   0  0  0  1  :   0   0    1;
     0  0  1  0   0  0  1  0  :   0   0    1;
     0  0  1  1   0  0  1  1  :   0   0    1;
     0  1  0  0   0  1  0  0  :   0   0    1;
     0  1  0  1   0  1  0  1  :   0   0    1;
     0  1  1  0   0  1  1  0  :   0   0    1;
     0  1  1  1   0  1  1  1  :   0   0    1;
     1  0  0  0   1  0  0  0  :   0   0    1;
     1  0  0  1   1  0  0  1  :   0   0    1;
     1  0  1  0   1  0  1  0  :   0   0    1;
     1  0  1  1   1  0  1  1  :   0   0    1;
     1  1  0  0   1  1  0  0  :   0   0    1;
     1  1  0  1   1  1  0  1  :   0   0    1;
     1  1  1  0   1  1  1  0  :   0   0    1;
     1  1  1  1   1  1  1  1  :   0   0    1;
```

Figure 12. Using LOG/iC to Generate a Comparator PROM File

```
      0  0  0  0   -  -  -  1  :   0   1    0; A < B CONDS.
      0  0  0  -   -  -  1  -  :   0   1    0;
      0  0  -  -   -  1  -  -  :   0   1    0;
      0  -  -  -   1  -  -  -  :   0   1    0;
      1  0  -  -   1  1  -  -  :   0   1    0;
      1  0  0  -   1  0  1  -  :   0   1    0;
      1  0  0  0   1  0  0  1  :   0   1    0;
      0  1  0  -   0  1  1  -  :   0   1    0;
      0  1  0  0   0  1  0  1  :   0   1    0;
      0  0  1  0   0  0  1  1  :   0   1    0;

      1  -  -  -   0  -  -  -  :   1   0    0; A > B CONDS.
      0  1  -  -   0  0  -  -  :   1   0    0;
      0  0  1  -   0  0  0  -  :   1   0    0;
      0  0  0  1   0  0  0  0  :   1   0    0;
      1  1  -  -   1  0  -  -  :   1   0    0;
      1  0  1  -   1  0  0  -  :   1   0    0;
      1  0  0  1   1  0  0  0  :   1   0    0;
      0  1  1  -   0  1  0  -  :   1   0    0;
      0  1  0  1   0  1  0  0  :   1   0    0;
      0  0  1  1   0  0  1  0  :   1   0    0;

*ROM
 TYPE = 8_IN_8_OUT;
 INPUTS = 8;
 OUTPUTS = 8;

*RUN
 PROG = INTEL;           Produce an INTEL-HEX output format

*END
```

Figure 12. Using LOG/iC to Generate a Comparator PROM File (continued)

Conclusion

PROM files can be easily generated in a variety of ways. If a complex function is desired, a high-level language approach is probably the best method. However, if a logical function is the desired result, PLD development tools will more than suffice.

Interfacing the CY7C276 High-Speed PROM to the AT&T, AD, Motorola, and TI DSPs

Introduction

Digital signal processors (DSPs) have typically required two external storage devices—a relatively slow PROM (Programmable Read Only Memory) for non-volatile code storage, and an SRAM (Static Random Access Memory), faster than the PROM, from which to run the code. The reason for this is that PROM access times are typically too slow to meet the requirements of the DSP cycle times.

The Cypress CY7C276 is a 16K x 16 UV-erasable PROM that can meet the fast cycle time requirements of a DSP design. It can help reduce component count and cost by eliminating the need for SRAM. If the goal is to eventually place the code in the internal Mask ROM (MROM) of the DSP, (which exists on the AT&T, Motorola and TI devices) then the CY7C276 can be utilized for prototyping until the code is completely debugged. This can save the expense of going to MROM prematurely.

This application note discusses how to use the CY7C276 PROM as program memory for various DSPs. It will cover the topic of interfacing the CY7C276 high-speed PROM to some of today's most popular DSPs for program memory only. Data memory storage is typically done with SRAM and its interface is not included in this application note. The AT&T DSP1616, Analog Devices ADSP 2100A, Motorola DSP56000 and TI TMS320C5x family of devices are discussed. Also included is a detailed description of the CY7C276 (including architecture, programming options, and signal descriptions) and brief descriptions of the DSPs (architectures, signals and timing requirements). For ease of explanation, only one example from each product family is included. The other devices in each product family are similar and are left as an exercise for the reader. Detailed timing calculations that show code sizes up to 16K words in depth are included in the examples. Finally, a table is provided to help summarize the analysis.

An Introduction to the CY7C276

The CY7C276 is a 16K x 16 asynchronous UV-erasable PROM with an access time of 25 ns. There are three polarity-programmable chip selects (CS[2:0]), which provide on-chip decoding of up to eight banks of PROM for a total of 256 Kbytes of PROM. The polarity of the asynchronous output enable (OE) pin is also user programmable. The CY7C276 provides a 16-bit-wide output, thus halving the number of PROMs required when interfacing to 16-bit or wider DSPs. With an access time of 25 ns, the CY7C276 can be used in 40-MHz DSP systems with zero-wait-states.

In all following examples, the CY7C276 is programmed to have all three chip enables (CS[2:0]) and the output enable (OE) active LOW. This is achieved by programming a Hex 0008 in location H4000.

AT&T – DSP1616

The DSP1616 is a 16-bit fixed-point DSP based on the popular DSP1600 core. It is object code upward-compatible with the DSP16, 16A, 16C, and 1610 devices from AT&T.

The DSP1616 can run out of either the 12K 16-bit words of on-chip MROM or external memory of up to 56K 16-bit words. Data memory space can also be accessed externally, although this application note only describes the external program memory interface. *Table 1* shows memory maps for the various configurations of DSP1616. IROM is Internal ROM, EROM is External ROM and RAM1/RAM2 are internal memory locations that are not utilized in this design and are therefore left as "don't cares." Two parameters, LOWPR and EXM, determine which memory map is used. LOWPR controls the address in memory assigned to the RAM1 and RAM2 areas. EXM (EXternal Memory) is an input signal that determines whether the internal ROM (IROM) or the external ROM (EROM) will be addressed in the memory map at location 0.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by asserting the RSTB (Reset Bar) pin LOW. When the RSTB pin is driven HIGH, the DSP1616 comes out of reset and fetches an instruction from location zero of the program space. The physical location of address zero is determined by which memory map is selected. The DSP1616 is configured for external ROM by holding the EXM (External ROM enable) signal HIGH at the rising edge of RSTB. With the

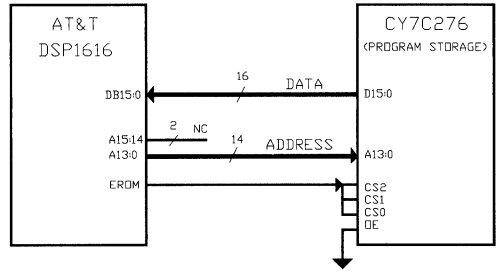


Figure 1. AT&T DSP1616 to PROM Interface

LOWPR bit set to 0 at reset, the corresponding memory map selected is Map2 (*Table 1*). This locates EROM at address 0.

DSP to Memory Interface

The DSP1616 uses the following signals to interface with external memory.

AB[15:0] – Address Bus. Outputs memory and I/O addresses.

DB[15:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{\text{EROM}}$ – (Program address External ROM enable) Access enable for external memory. Active LOW.

The implementation of the DSP to PROM interface is shown in *Figure 1*.

Table 1. DSP1616 Memory Maps

Decimal Address	MAP1 EXM = 0 LOWPR = 0	MAP2 EXM = 1 LOWPR = 0	MAP3 EXM = 0 LOWPR = 1	MAP4 EXM = 1 LOWPR = 1
0	IROM	EROM	RAM1	RAM1
1K			RAM2	RAM2
2K			Reserved	Reserved
8K			IROM	EROM
12K	RAM1	RAM1		
13K	RAM2	RAM2		
14K	Reserved	Reserved		
20K 64K−1	EROM	EROM	EROM	

Timing Notes

The DSP1616 has numerous mask-programmable options for the clock source. It can use a crystal oscillator, or a small signal (TTL, or CMOS level) oscillator. The external source can be supplied by either a crystal or an oscillator. These options are discussed in detail in the AT&T DSP1616 datasheet. The DSP1616 can run at either the same or half the input frequency. The datasheet calls this a 1x or 2x clock, referring to the ratio of the input clock to the processor (internal) clock. *Table 4* notes this as the 1x and 2x clock options respectively.

The calculations for the required access time of the CY7C276 are illustrated below. The timing diagram for this example is shown in *Figure 2*.

$t_{AA}(PROM) = (t_{c(CKO)} - 2) - t_{ASKW(DSP)} - t_{SU(D)R}$

where:

$t_{AA}(PROM)$ = PROM address access time required,

$t_{c(CKO)}$ = DSP CKO cycle time [$t_{c(CKO)} - 2$ is to compensate for the worst-case EROM cycle time, which would be 2 ns shorter than CKO, as specified in the DSP1616 datasheet]. CKO is the clock out signal from the DSP,

$t_{ASKW}(DSP)$ = Worst-case address valid time from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time required by DSP before EROM goes HIGH.

Substituting values from a DSP1616 datasheet gives us:

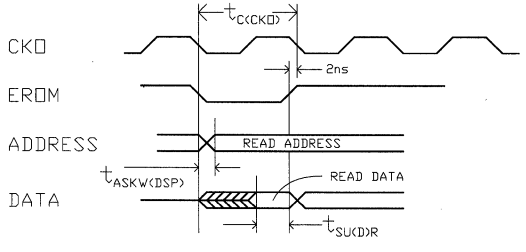


Figure 2. DSP1616 External Program Memory Timing

$t_{AA}(PROM) = (50 \text{ ns} - 2) - 2 \text{ ns} - 17 \text{ ns} = 29 \text{ ns}.$

for a 40-MHz DSP with the 2x input clock option.

Summary

From the above analysis, it can be seen that AT&T's DSP1616 can run code directly out of a CY7C276 at 40 MHz with the 2x input clock option with zero wait states. The CY7C276–25 (25-ns access time) can be used to satisfy this requirement.

Analog Devices – ADSP2100A

The ADSP2100A is a 24-bit fixed-point DSP that utilizes 24-bit instructions. It has a 14-bit address bus that directly addresses 16K 24-bit words externally and is expandable to 32K 24-bit words by using the program memory data access (PDMA) signal as a chip select. This example discusses the memory interface for 16K 24-bit words of program size. There is no on-chip memory for code/data storage. The ADSP2100A is the fastest available 24-bit DSP from Analog Devices that can access external ROM. (As of writing this application note.) It has a maximum clock frequency of 12.5 MHz.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by driving the $\overline{\text{RESET}}$ pin LOW. When the $\overline{\text{RESET}}$ pin is driven HIGH, the ADSP2100A comes out of reset and fetches an instruction from location H0004 of the program space (The first three locations of memory contain the interrupt vector addresses). In the case of the ADSP2100A, where there is no internal memory, the address (H0004) appears on the program memory address (PMA) bus followed by assertion of the program memory select ($\overline{\text{PMS}}$) and program memory read ($\overline{\text{PMRD}}$) signals. The DSP has completed the reset sequence and is now ready to execute code.

DSP to Memory Interface

The ADSP2100A uses the following signals to interface with external memory.

PMA[13:0] – Address Bus. Outputs memory and I/O addresses.

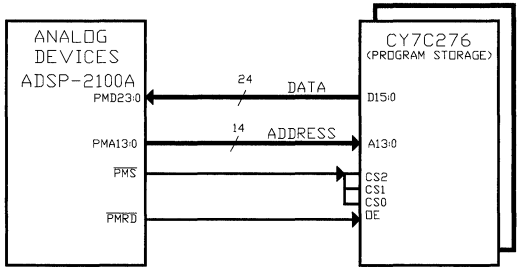


Figure 3. ADSP2100A to PROM Interface

PMD[23:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{\text{PMS}}$ – Program Memory Select. Used to access external memory.

$\overline{\text{PMRD}}$ – Program Memory Read. Used for external memory output enable.

The implementation of the DSP to PROM interface is shown in *Figure 3*.

Timing Notes

The ADSP2100A datasheet directly specifies the maximum allowable access times to run code directly out of PROM. The timing diagram for this example is shown in *Figure 4*.

The result with the DSP running at a frequency of 12.5 MHz is:

t_{AA}(PROM)max = 32 ns.

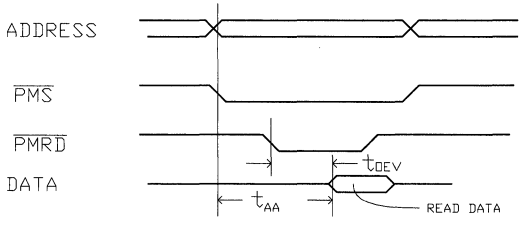


Figure 4. ADSP2100A External Program Memory Timing

where:

t_{AA}(PROM) = PROM address access time required. This is specified in the datasheet as PMA valid to PMD input valid.

Because this device uses the $\overline{\text{PMRD}}$ signal to control the OE of the CY7C276, the OE to data valid time must also be calculated. The following was also directly specified in the datasheet as $\overline{\text{PMRD}}$ LOW to PMD input valid.

t_{OEV}(PROM)max = 18 ns.

The CY7C276–30 satisfies both of these timing requirements.

Summary

From the above analysis, it can be seen that Analog Devices' ADSP2100A can run code directly out of two CY7C276s with zero wait states at its maximum frequency of 12.5 MHz. The CY7C276–25 or CY7C276–30 (25-ns and 30-ns access times, respectively) can be used to satisfy this requirement.

Motorola – DSP56000

The DSP56000 is a 24-bit general purpose DSP. It has 3.75K x 24-bits of on-chip ROM and can also run from external memory of 64K 24-bit words of program space. *Table 2* shows the memory maps for the various configurations of DSP56000. Mode 0 is the single-chip mode for use with internal ROM only. Mode 1 on the DSP56000 is for test purposes only and should not be invoked. Mode 2 is the normal expanded mode and is identical to Mode 0 except that the reset vector is in a different location. Mode 3 is Development Mode, which disables the internal ROM. All references to program memory space in this mode are directed to external program memory. There are two pins (MODA and MODB) that are sampled at the end of reset to determine which memory map is used.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by driving the $\overline{\text{RESET}}$ pin LOW. When the $\overline{\text{RESET}}$ pin is driven HIGH, the DSP56000 comes out of reset and

fetches an instruction from the reset vector location of the program space. The physical location of the reset vector is determined by which memory map is selected. If MODE 0 or 3 is selected, the reset vector is at location H0000 (location 0). If MODE 2 is selected, the reset vector is at location HE000. The DSP56000 is configured for external ROM by setting MODA and MODB HIGH at the rising edge of $\overline{\text{RESET}}$. The state of MODA and MODB will determine which memory map is selected for use (see *Table 2*). This example will use MODE 3 (Development Mode).

So, for MODE3, DSP56000 will start executing code from location zero of the external memory after reset is complete. All 64K words of external memory, except the first 64 locations (used for interrupts), are available for program storage. Location H0000 contains the reset vector which holds the programs starting address. This example will use two CY7C276 16K x 16 PROMs to achieve the 24-bit-wide program memory bus that is required. The upper eight bits of the PROM will not be used.

DSP to Memory Interface

The DSP56000 uses the following signals to interface with external memory.

A[13:0] – Address Bus. Outputs memory and I/O addresses.

D[23:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{\text{PS}}$ – Program Memory Select. Used to access external memory.

$\overline{\text{RD}}$ – Read Select. Used for external memory output enable.

The implementation of the DSP to PROM interface is shown in *Figure 5*.

Timing Notes

The DSP56000 takes two external clock cycles for a read operation. The address becomes available at about the midpoint of the first cycle referenced from the falling edge of clock. The data is read into the DSP at the middle of the second cycle or the second rising edge of clock (see *Figure 6*). This actually works to the PROM's advantage by lengthening the required access time.

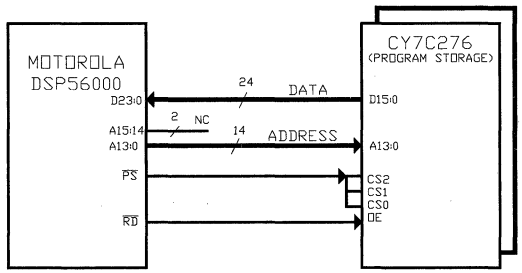


Figure 5. Motorola DSP56000 to PROM Interface

Table 2. DSP56000 Memory Maps

Decimal Address	MODE 0 MB = 0 MA = 0 Internal ROM Internal Reset	MODE 1* MB = 0 MA = 1 TEST MODE DO NOT USE	MODE 2 MB = 1 MA = 0 Internal ROM External Reset	MODE 3 MB = 1 MA = 1 No Int. ROM External Reset
0	Reset			Reset
3839 (H0EFF)	INTERNAL ROM		INTERNAL ROM	
3840	EXTERNAL		EXTERNAL	EXTERNAL
60K			Reset	
64K–1				

* Mode 1 is for test purposes only on the DSP56000 and should not be invoked by the user.

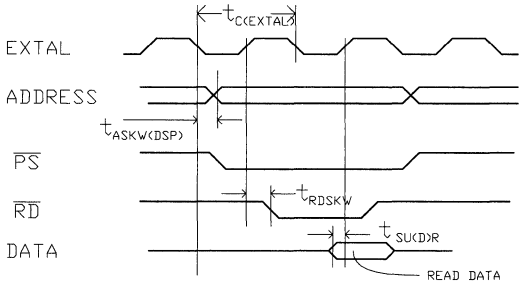


Figure 6. Motorola DSP56000 External Program Memory Timing

The calculations for the required access time of the CY7C276 are shown below. The timing diagram for this example is shown in *Figure 6*.

$t_{AA}(PROM)max = t_{c+} - t_{ASKW}(DSP) - t_{SU(D)R}$

where:

$t_{AA}(PROM)$ = PROM address access time required,

t_{c+} = DSP CLOCK IN cycle time x 1.5 (due to data read into DSP occurring at midpoint of second external clock cycle),

$t_{ASKW}(DSP)$ = Worst case address valid from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time before end of cycle.

This device uses the $\overline{RD}$ signal to control the OE of the CY7C276. Therefore, the OE access time must also be calculated. The following equation is for calculating the maximum allowable OE time of the CY7C276.

$t_{OEV}(PROM)max = t_{c(EXTAL)} - t_{RDSKW} - t_{SU(D)R}$

where:

$t_{c(EXTAL)}$ = clock cycle,

$t_{RDSKW}(DSP)$ = Worst case $\overline{RD}$ valid from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time before end of cycle.

The result with the DSP running at a frequency of 33 MHz is:

$t_{AA}(PROM)max = (30 \times 1.5)\ ns - 19\ ns - 0\ ns$
$= 45\ ns - 19\ ns$
$= 26\ ns.$

$t_{OEV}(PROM)max = 30ns - 16ns - 0ns = 14ns.$

Summary

Based on the above analysis, it can be seen that Motorola's DSP56000 can run code directly out of a CY7C276 with zero-wait-states at its maximum frequency of 33 MHz. The CY7C276–25 (25-ns access time) can be used to satisfy this requirement.

Texas Instruments – TMS320C5X

The TMS320C5X devices are a family of 16-bit fixed-point DSPs based on the older TMS320C25 CPU core. Significant modifications were added to improve performance. These devices are capable of running at twice the speed of the 'C2x family and are source-code upward compatible with all previous fixed-point DSPs from TI.

There are three devices in the 'C5x family, the 'C50, 'C51 and 'C53. They have 2K, 8K and 16K words of on-chip ROM, respectively. All three devices can run out of external ROM of up to 64K 16-bit words. All 64K words of external memory are available for program storage. *Table 3* shows the memory map for the TMS320C50 as an example. The SARAM is 9K words of program/data Single-Access RAM. This is memory that can only be read or written in a single machine cycle. It can reside on-chip or externally depending on the setting of the RAM bit in the PMST register. The DARAM is 1056 words of Dual-Access data RAM. It can be read from or written to in the same cycle. The DARAM can reside on-chip or externally depending on the setting of the CNF bit on the PMST register. The map in *Table 3* gives an example of the use of these two sections of memory.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by asserting the $\overline{RS}$ pin LOW. When the $\overline{RS}$ pin is driven HIGH,

the TMS320C5x comes out of reset and fetches an instruction from location 0. Location 0 (either on-chip or externally) contains the reset vector. The TMS320C5x is configured for external memory by holding the MP/$\overline{MC}$ pin HIGH at the rising edge of $\overline{RS}$.

Table 3. TMS320C50 Memory Maps

Decimal Address	'C50 MAP 1 MP/$\overline{MC}$ = 1 Internal	'C50 MAP 2 MP/$\overline{MC}$ = 0 External
0	Interrupts & Reserved (On-Chip)	Interrupts & Reserved (External)
48	On-Chip ROM	External
2K	On-Chip SARAM (RAM=1)	SARAM (RAM=0)
11K	External	External
63.5K 64K−1	On-Chip DARAM B0 (CNF=1)	DARAM External (CNF=0)

DSP to Memory Interface

The TMS320C5x uses the following signals to interface with external memory.

A[15:0] – Address Bus. Outputs memory and I/O addresses.

D[15:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{PS}$ – Program Memory Select. Used to access external memory.

$\overline{RD}$ – Read Select. Used for external memory output enable.

The implementation of the DSP to PROM interface is shown in *Figure 7.*

Timing Notes

The TMS320C5x can use either its internal oscillator or an external frequency source for a clock. The external source can either be a crystal or an oscillator. These options are discussed in detail in the TI User's Guide for the TMS320C5x. The TMS320C5x can run at either the same or half the input frequency. This means that the internal machine cycle and, subsequently, external accesses, can cycle at the same or one-half times the external frequency. The table at the end of this document notes this as the ÷1 and ÷2 clock options respectively.

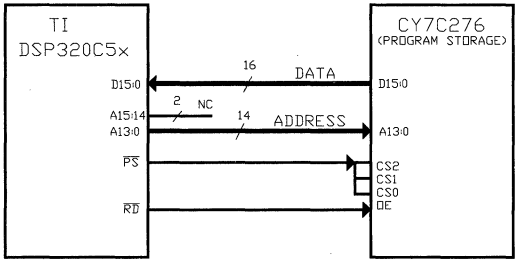


Figure 7. TMS320C5x to PROM Interface

The calculations for the required access time of the CY7C276 are illustrated below. The timing diagram for this example is shown in *Figure 8.*

$t_{AA}(PROM) = t_{c(CO)} - t_{ASKW}(DSP) - t_{SU(D)R}$

where:

$t_{AA}(PROM)$ = PROM address access time,

$t_{c(CO)}$ = DSP CLKOUT1 cycle time,

$t_{ASKW}(DSP)$ = Worst case address valid from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time before $\overline{RD}$ goes HIGH.

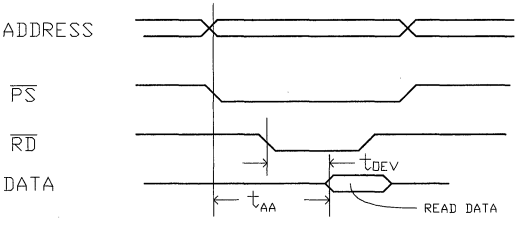


Figure 8. TMS320C5x External Program Memory Timing

The result with a frequency of 40 MHz and using the divide-by-two clock option the result is:

t_{AA}('276)max = 48.8 ns − 2 ns − 10 ns = 36.8 ns.

Summary

Based on the above analysis, it can be seen that TI's TMS320C5x series of DSPs can run code directly out of a CY7C276 at 40 MHz with zero-wait-states. The CY7C276−25 or CY7C276−30 (25-ns and 30-ns access times, respectively) can be used to satisfy this requirement.

Table 4 below has been provided to give a quick synopsis of the processors covered in this application note. It provides a quick cross reference of the CY7C276 PROM access times to DSP clock speeds and the number of wait states required.

Summary

These examples show how effectively the CY7C276 PROM can be used for executing code in DSP applications. The need for more costly SRAM is eliminated. There is no additional logic required to interface the PROM to the DSP thereby reducing pin count and cost in the design. As *Table 4* illustrates, most of the DSP speed grades can run code directly out of the CY7C276 with zero-wait-states.

Table 4. Wait State Requirements

DSP PART NUMBER	DSP Frequency and Clock Option If Applicable	# of Wait States Required for each PROM		
		CY7C276−25	CY7C276−30	CY7C276−35
AT&T DSP1616	20 MHz w/1x clock	0	1	1
	40 MHz w/2x clock	0	1	1
	20 MHz w/2x clock	0	0	0
ADSP2100A	12.5 MHz	0	0	1
	10.24 MHz	0	0	0
DSP56000	33 MHz	0	1	1
	27 MHz	0	0	0
	20.5 MHz	0	0	0
TMS320C5X	57 MHz with ÷1 clock option	2	3	3
	40 MHz with ÷1 clock option	1	2	2
	57 MHz with ÷2 clock option	1	1	1
	40 MHz with ÷2 clock option	0	0	1

Using the CY27H010 with the Rockwell V.FAST Chipset

The purpose of this application note is to describe how to use the Cypress CY27H010 1-megabit PROM with the Rockwell V.FAST chipset to create a high-speed fax/modem. A system block diagram and timing analysis are included.

Traditionally, PROMs have been ideal for non-volatile code storage in embedded systems such as modems, yet have been unable to provide the speed necessary to meet system requirements. In order to solve this performance bottleneck, designers typically download code from the slow PROM into a fast, yet expensive SRAM. Usually, this transfer takes place during system boot-up and is transparent to the user. Once in SRAM, code can be run much faster, usually with 0 wait states. The obvious trade-offs for this added performance are the cost of the SRAM, board area, and design complexity.

The introduction of the fast Cypress CY27H010 1-megabit (128Kx8) PROM has eliminated the need for this compromise in many systems. The CY27H010 delivers the performance required to run code at full speed directly out of the PROM. Not only will this simplify the design, it will also lower the system cost and board area. In addition to being fast enough for most high-speed applications, the CY27H010 is also large enough to satisfy most code storage requirements. These two factors are demonstrated below as the CY27H010 is used with the Rockwell V.FAST chipset at full speed, with 0 wait states.

The Rockwell V.FAST modem device set consists of three separate devices: (1) the L39 Micro Controller unit (MCU), which performs all of the command processing and host interface operations, (2) the Modem Data Pump (MDP), which can operate as either a data modem at up to 28.8 Kbps, or a fax modem up to 14.4 Kbps, and (3) the optional Compression Expansion Processor (CEP), which can increase system performance by performing dedicated compression and expansion functions in V.42 bis or MNP 5 modes. The CY27H010 provides the code storage for the MCU and is independent of the configuration (serial or parallel) or whether the CEP is being used. An additional SRAM is required to provide scratch pad memory for the MCU, but that topic is beyond the scope of this application note. If used, the CEP requires an additional SRAM and PROM for code storage and scratch pad memory. These additional devices are not involved in this discussion.

Typically, when designing with a Micro Controller like the L39, the engineer must become familiar with all of the various modes, functions, and registers of the device. This is essential in order to set the numerous registers that establish the appropriate functionality. An example of the variables that need to be set are: number of wait states when accessing external PROM, polarity of certain signals, ...etc. This tedious process has been simplified when using the Rockwell V.FAST chipset. Rockwell provides the firmware necessary to properly configure the MCU. A PC-based utility program is available that allows designers to modify the base configuration in order to suit their particular requirements. When the default settings are used, all of the required parameters are established that affect the MCU-PROM interface on the expansion bus. These parameters are: (1) 1X internal clock frequency (20.5-MHz external and internal, this provides a

48.1-ns cycle time), (2) establishing the functionality of the $\overline{\text{ROMSEL}}$ output on Port B, bit 2, and (3) 0 wait state operation when accessing the expansion bus.

One requirement placed on the hardware design is to enable or disable the 8 KB of on-chip ROM. The on-chip ROM is mask programmable and therefore is of little use during system development or when a large program is required. Once the code has solidified, this on-chip ROM can be used for code storage, provided the size of the code is less than 8 KB. This on-chip ROM can be disabled by grounding the TST pin on the MCU. By doing so, the device will automatically look to the expansion bus for ROM accesses and during the boot sequence.

Once configured, the MCU uses Port B, bit 2 as the $\overline{\text{ROMSEL}}$ output. This signal is used to select the appropriate external device, which in this case is the PROM. This signal should be tied directly to the $\overline{\text{CS}}$ input of the PROM. If multiple PROMs were being used, additional $\overline{\text{ROMSEL}}$ lines would be generated in order to select the correct device. The MCU also generates a $\overline{\text{READ}}$ signal that is strobed LOW during external read cycles. This signal should be connected to the $\overline{\text{OE}}$ of the CY27H010. By using the $\overline{\text{OE}}$ pin we are able to take advantage of the fast t_{DOE} in order to satisfy system timing. The MCU-PROM interface is shown in *Figure 1*.

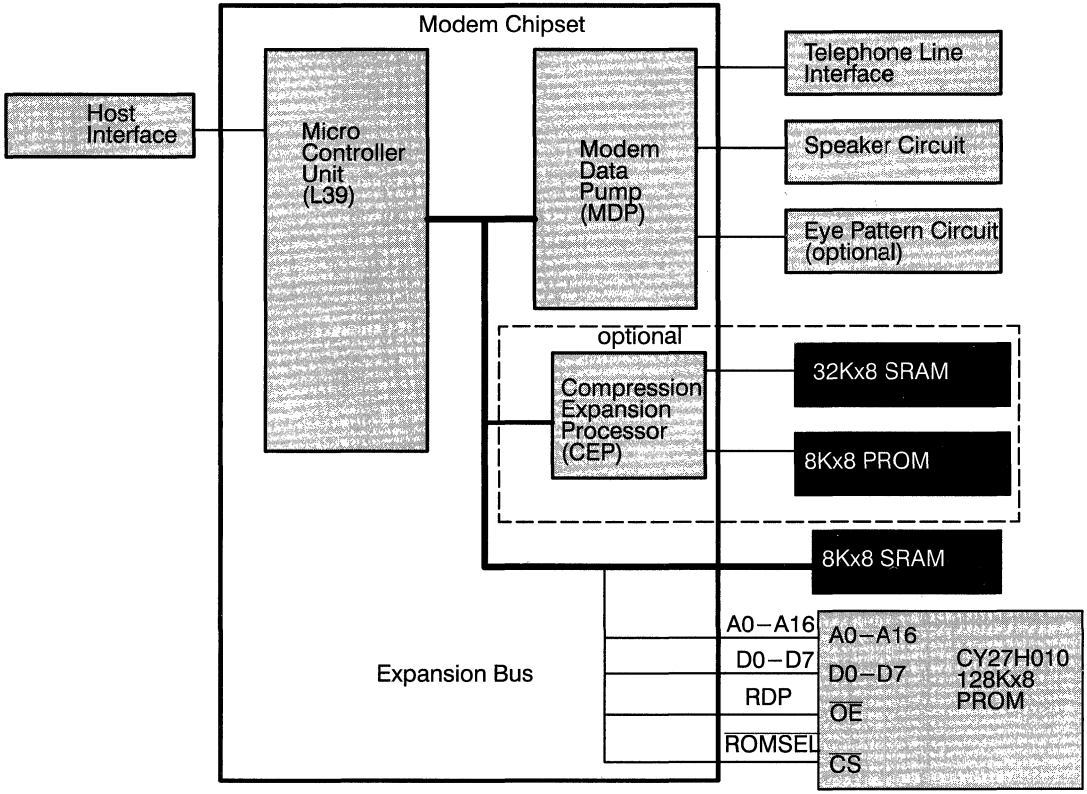


Figure 1. Rockwell V.FAST Modem Block Diagram

Timing Analysis

A basic read on the expansion bus is shown in *Figure 2*. As can be seen in the diagram, the address, $\overline{\text{ROMSEL}}$, and $\overline{\text{READ}}$ signals are generated from one falling edge of the clock, and the data is captured by the MCU on the next falling edge. A/C timing must now be verified. Although the critical path is through t_{DOE}, t_{AA} and t_{ACE} must be verified as well. All timing specifications were taken directly out of the L39 MCU technical manual.

t_{AA} (required) $= t_{CYC} - t_{AS} - t_{RDS}$
$= 48.1 - 12.0 - 4.5$
$= 31.6$ ns (t_{AA} max. for CY27H010–25 is 25 ns!)

t_{ACE} (required) $= t_{CYC} - t_{AS} - t_{RDS}$
$= 48.1 - 12.0 - 4.5$
$= 31.6$ ns (t_{ACE} max. for CY27H010–25 is 30 ns!)

t_{DOE} (required) $= t_{RW} - t_{RDS}$
$= 24.6 - 4.5$
$= 20.1$ ns (t_{DOE} max. for CY27H010–25 is 15 ns!)

All of the A/C requirements shown above can be satisfied with a CY27H010–25 device.

Conclusion

With the firmware provided by Rockwell, the functional interface between the MCU and the PROM has been greatly simplified. In addition, the timing provided in the Rockwell has made the A/C analysis straight forward. After comparing the required A/C numbers to those published in the CY27H010 data sheet, it is apparent that a CY27H010–25 device is able to provide the speed required by the Rockwell V.FAST chipset to run code directly out of PROM with 0 wait states.

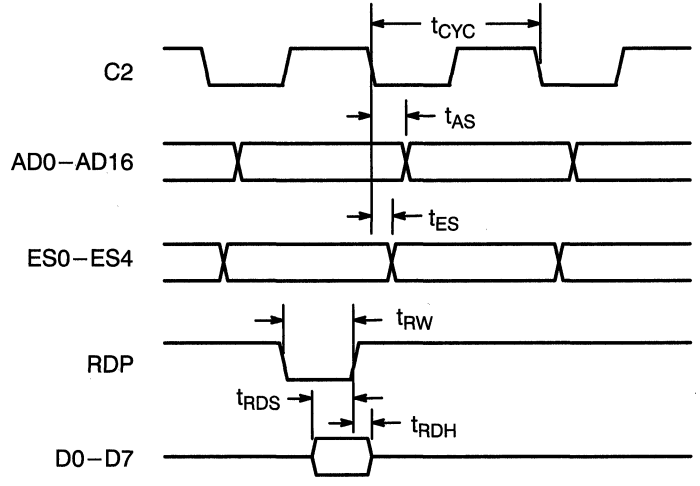


Figure 2. Expansion Bus Read Waveform

Interfacing a 5V Cypress PROM to a 3.3V System using a CYBUS3384 Bus Switch

This application note describes a method for interfacing a high-speed 5V Cypress PROM to a 3.3V system. The I/O level translation is achieved using a CYBUS3384 Bus Switch.

PROMs (Programmable Read Only Memories) are often used for code storage and can interface directly to the host processor bus. Many applications use fast Cypress PROMs to read code directly from the PROM (instead of downloading the code to a fast SRAM that administers the code to the processor). If a 3.3V host processor is being used that is not "5V safe," input levels may be exceeded and problems can arise. Additionally, high speed 3.3V PROMs may be difficult to locate. Using slower 3.3V PROMs can either decrease system performance or increase system cost, or both. Fortunately, this dilemma can be resolved by using a CYBUS3384 Bus switch to translate from 5V to 3.3V compatible levels with essentially no timing penalty. Since there is no speed penalty, the same high-speed 5V Cypress PROM can be used to achieve the same performance level. This immediate translation is essential to preserving system timing in high speed systems.

The CYBUS3384 was originally designed to three-state signals for busing applications. Due to the symetric nature of the MOS device being used, the CYBUS3384 can also be used in bidirectional applications (e.g., I/O pins commonly used on SRAMs). The "switch" consists of a simple NMOS pass gate controlled by a common (active LOW) enable signal as shown in *Figure 1*. When a LOW signal is applied to the control line, the signal applied to one side of the switch (side A) is allowed to propagate directly through to the output on the other side (side B). A HIGH signal applied to the control line would prevent the input from propagating to the output and would place the output in a high impedance, three-state condition. The output of the pass gate is a function of both the gate and drain voltages. The gate voltage is a function of V_{CC}. Therefore, by regulating V_{CC} of the Bus Switch we are able to control the voltage applied to the gate of the pass gate,

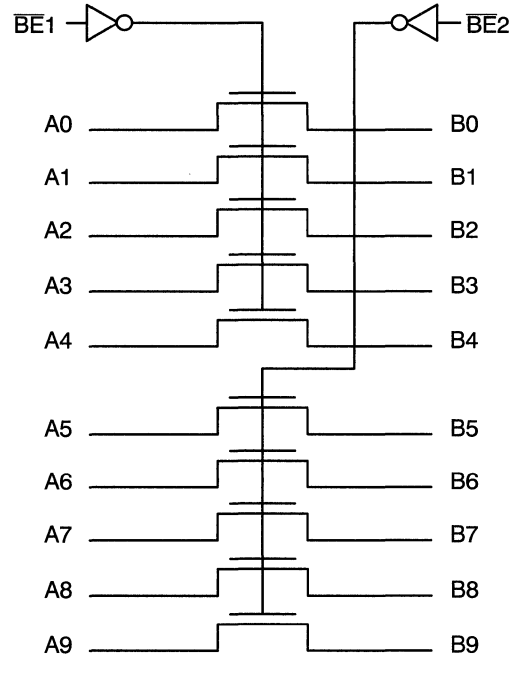


Figure 1. Configuration of the Bus Switch

which in turn limits the output swing of the device. If V_{CC} is properly regulated, the output levels can be 3.3V compatible.

The critical requirement for this circuit is to limit the V_{CC} applied to the Bus Switch. This can easily be accomplished with the existing 5V power supply and a simple zener diode-resistor network as shown in *Figure 2*. By adjusting either the resistor value or changing the zener diode, the V_{CC} applied to the Bus Switch and the output levels can be tuned to the desired values. For a 5V−>3.3V conversion, the resistor used should be between 40−100 ohms (1/4 watt) and the zener diode has a Vz=4.3V (I_{ZT}=10 mA). A 3.9V Zener can be used if a smaller I/O swing is desired. This type of configuration will only draw approximately 10 mA. It is important to select a low-current zener diode so the desired results can be achieved without burning excess power.

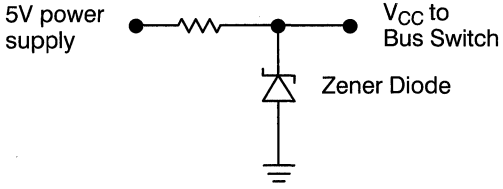


Figure 2. Regulator Circuit

The best feature of this 5V/3.3V translation is that no speed penalty is incurred. The maximum delay through the CYBUS3384 is 250 ps, well below the guardband of most high-speed designs. Therefore, Cypress's high-speed 5V PROMs can be used in 3.3V systems without any speed penalty by merely translating the I/O levels.

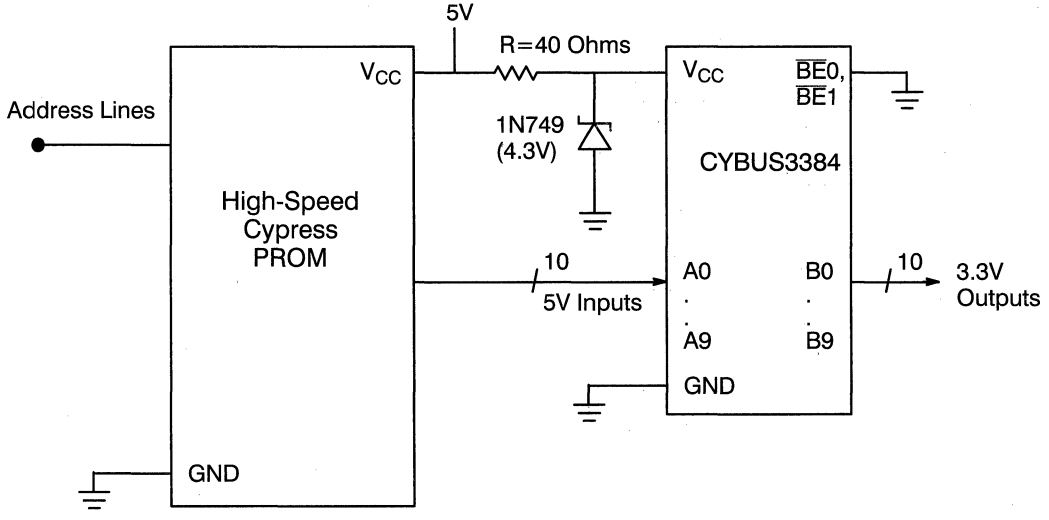


Figure 3. Final Implementation

UltraLogic/PLDs – 4

UltraLogic/PLDs Section Contents and Abstracts

This application note provides a detailed description of the metastable behavior in PLDs from both circuit and statistical viewpoints. Additionally, the information on the metastable characteristics of Cypress PLDs presented here can help achieve any desired degree of reliability.

This application note provides an overview of the CY7C335 Universal Synchronous EPLD architecture and *Warp2* VHDL Compiler for PLDs. Example designs demonstrate how the *Warp2* VHDL compiler takes advantage of the rich architectural features of the CY7C335.

This application note is intended to assist *Warp*™ users in converting designs written in DATA I/O's ABEL hardware description language to IEEE 1076 VHDL. It contains several language cross reference tables and many helpful hints. It also includes two real-world designs that have been converted from MACH™ 210-ABEL descriptions to FLASH371-VHDL descriptions.

The purpose of this application note is to compare and contrast the complexity and basic features of Abel-HDL with those of IEEE–1076 VHDL. Both of these languages are very robust in their support of different types of constructs that can be used to describe the same functionality at different levels of abstraction. It is beyond the scope of this document to exhaustively describe these possibilities or to present a complete tutorial for writing code in either language because of the great variety of constructs and syntax available with which to describe the functionality of a given circuit. Rather, a simple sample design that contains a mixture of synchronous and asynchronous logic circuits will be shown. Sample code is written in both Abel–HDL and VHDL that describes the example's functionality and synthesizes to create functionally identical hardware. The code written here represents a typical level of abstraction that balances readability with compactness. With experience, designers can develop their own preferences for style. For instance, state machines can be described in a number of ways: state tables, IF-THEN-ELSE statements, CASE-WHEN statements, or explicitly using a combination of Register-Transfer-Level (RTL) code (individually describe each gate/register as a component with its inputs and outputs) and/or Boolean equations.

This application note introduces Cypress's high-density complex programmable logic device family of products. The innovative architectural features of this family are discussed relative to competitor implementations. Some simple VHDL examples are shown that demonstrate usage or the features of the architecture using VHDL hardware description language available from Cypress's design development tool called *Warp*.

This application note gives some criteria that can be used to determine when the CY7B933 HOTLink Receiver should be forced to reframe its data, and it describes in detail a specific design of a reframe controller that implements these criteria. The design is described in VHDL and is implemented in the 32-macrocell CY7C371 FLASH CPLD.

This application note describes how to implement a dual-port RAM using a standard SRAM and a Cypres FLASH370 CPLD. Commercially available dual-port devices are limited in both width and depth. By increasing the size of the SRAM array, this design can be modified to simulate a dual-port that is much larger than those offered as an individual part. VHDL is included to show how the arbitration and control functionality are coded into the CPLD.

This application note is intended to help designers create efficient arithmetic designs targeting a FLASH370 Complex programmable logic device (CPLD). The discussion in this application note addresses arithmetic algorithms and implementations tailored to the features and resources offered in the FLASH370 family of CPLDs. These specialized arithmetic designs achieve a balanced trade-off between speed/area requirements for a given application. The implementation details and design trade-offs in building adders, subtractors, equality and magnitude comparators is addressed in detail in this application note. This application note includes many VHDL examples to illustrate the working and implementation of the algorithms presented. This application note is also intended to create a solid foundation from which designers can pick up ideas and concepts and create their own algorithms/implementations, with a good understanding of the constraints to be dealt with.

If on-board programmability is a must for a design, the FLASH370 CPLD devices can be used to satisfy this need. The 128-macrocell CY7C374 and CY7C375 devices can be programmed in a normal fashion by placing them in a programming station. On-board programming is accomplished by providing a few simple additions when designing the board. The actual on-board programming of the device is then done by placing the board into a programming mode and connecting the programming station to the board. All of the steps to be followed to achieve on-board programming for the CY7C374 and CY7C375 are described.

This application note explains how to generate simulation models for the Mentor Quicksim II simulation tool using the Cypress *Warp* tools. These models are fully functional and include timing delays based on the Cypress datasheets. These models can be generated from any *Warp* tool including the *Warp2* software, which is available for $99.

Key issues in FPGA architectures are identified and are related to the interconnect technology (the technology used to connect two wires under user programmability). Logic cell architecture, number of interconnects available, routability, and performance are related to SRAM based, large anti-fuse based, and ViaLink™ fused based interconnect technologies. The relationships are used to explain characteristics of certain device families using the various fuse technologies. Characteristics include fitting capability, internal propagation delays, and other factors of interest to FPGA users.

This application note takes the reader through the design process to implement a DRAM controller in a pASIC380 FPGA. The purpose is to introduce the features of the pASIC380 family as well as how to take advantage of those features with the *Warp* design environment and VHDL. Using the static timing analyzer and dynamic timing simulator, path analysis and design verification are illustrated.

The Peripheral Component Interconnect (PCI) bus is a high-bandwidth, "plug and play" bus designed to meet the performance and bandwidth demands of today's applications. Interfacing to the PCI bus requires strict adherence to the PCI Local Bus Specifiation. Translating from PCI to the peripheral application demands a flexible, PCI-compliant solution. With the flexibility of FPGAs, the task of interfacing between PCI to the peripheral application can be accomplished. This application note provides an overview of the PCI bus and its associated transactions, and presents an example PCI Target interface design, as well as addressing some design challenges encountered when implementing a PCI interface.

Calculating power consumption for a pASIC device may be required prior the completion of the detailed design. This can be difficult without detailed knowledge of the number of logic cells used and the toggle rate for each of the cells. However, with a general knowledge of the percent of the device used and the average toggle rate for various sections of the design, the power can be easily estimated. This application note presents a quick power estimation procedure. A worksheet along with graphs for rapid estimation of worksheet power values is included. An example is also provided.

This application note explains the basic design process for an FPGA device using the *Warp3* software. The note also explains the Cypress pASIC380 FPGA architecture and fuse technology in detail. A DMA controller design is used as the example design. A portion of the design is done using VHDL entry and the rest is captured using schematic elements. Detailed state diagrams and example VHDL code along with the schematic printouts are included.

This application note describes many of the options encountered during a state machine design cycle. The different methods of describing a state machine design are covered briefly. The different types of state machines are described. Most of the application note is a design example of a clock generator for a bit-slice processor. The last section shows the necessary steps to implement the clock generator in a CY7C361 device. The appendices include source code, reduced equations, pinouts, and simulation results.

This application note describes how to construct a hierarchical design using *Warp* VHDL. It first discusses the features of VHDL that are designed to simplify hierarchical design. The reader is then walked through a design sample that is modified to illustrate increasingly advanced topics.

Galileo from Exemplar Logic and the Design Compiler from Synopsys provide two pathways for programmable logic users to target Cypress's UltraLogic devices. Both of these tools integrate tightly with Cypress's *Warp* design tool to complete the UltraLogic design flow. This application note intends to familiarize the reader with these third-party design tools and their integration with the Cypress UltraLogic design pathway.

Are Your PLDs Metastable?

This application note provides a detailed description of the metastable behavior in PLDs from both circuit and statistical viewpoints. Additionally, the information on the metastable characteristics of Cypress PLDs presented here can help you achieve any desired degree of reliability.

Metastable is a Greek word meaning "in between." Metastability is an undesirable output condition of digital logic storage elements caused by marginal triggering. This marginal triggering is usually caused by violating the storage elements' minimum set-up and hold times.

In most logic families, metastability is seen as a voltage level in the area between a logic HIGH and a logic LOW. Although systems have been designed that did not account for metastability, its effects have taken their toll on many of those systems.

In most digital systems, marginal triggering of storage elements does not occur. These systems are designed as synchronous systems that meet or exceed their components' worst-case specifications. Totally synchronous design is not possible for systems that impose no fixed relationship between input signals and the local system clock. This includes systems with asynchronous bus arbitration, telecommunications equipment, and most I/O interfaces. For these systems to function properly, it is necessary to synchronize the incoming asynchronous signals with the local system clock before using them.

Figure 1 shows a simple synchronizer, whose asynchronous input comes from outside the local system. The synchronizer operates with a system clock that is synchronous to the local system's operation. On each rising edge of this system clock, the synchronizer attempts to capture the state of the asynchronous input. *Figure 2* shows the expected result. Most of the time, this synchronizer performs as desired.

Digital systems are supposed to function properly all the time, however. But because there is no direct relationship between the asynchronous input and the system clock, at some point the two signals will both be in transition at very nearly the same instant. *Figure 3* shows some of the synchronizer's possible

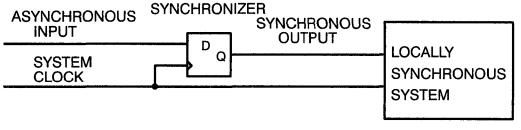


Figure 1. Simple Synchronizer

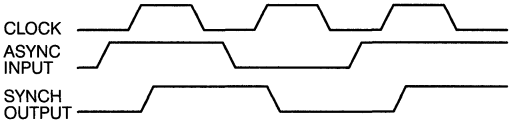


Figure 2. Expected Synchronizer Output

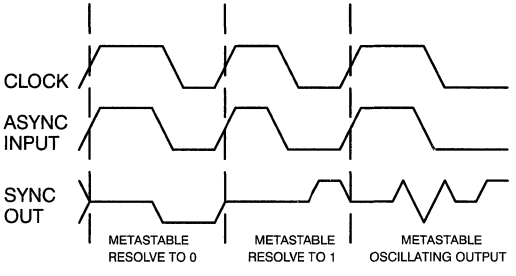


Figure 3. Possible Metastable States of Synchronizer

metastable outputs when this input condition occurs. These types of outputs would not occur if the synchronizer made a decision one way or the other in its specified clock-to-output time. A flip-flop, when not properly triggered, might not make a decision in this time. When improperly triggered into a metastable state, the output might later transition to a HIGH or a LOW or might oscillate.

When other components in the local system sample the synchronizer's metastable output, they might also become metastable. A potentially worse problem can occur if two or more components sample the metastable signal and yield different results. This situation can easily corrupt data or cause a system failure.

Such system failures are not a new problem. In 1952, Lubkin (Reference 1) stated that system designers, including the designers of the ENIAC, knew about metastability. The accepted solution at that time was to concatenate an additional flip-flop after the original synchronizer stage (*Figure 4*). This added flip-flop does not totally remove the problem but does improve reliability. This same solution is still in wide use today.

Recovery from metastability is probabilistic. In the improved synchronizer, the first flip-flop's output might still be in a metastable state at the end of the sample clock period. Because the flip-flops are sequential, the probability of propagating a metastable condition from the second flip-flop stage is the square of the probability of the first flip-flop remaining metastable for its sample clock period. This type of synchronizer does have the drawback of adding one clock cycle of latency, which might be unacceptable in some systems.

As system speeds increase and as more systems utilize inputs from asynchronous external sources, metastability-induced failures become an increasingly significant portion of the total possible system failures. So far, no known method totally eliminates the possibility of metastability. However, while you cannot eliminate metastability, you can employ design techniques that make its probability relatively small compared with other failure modes.

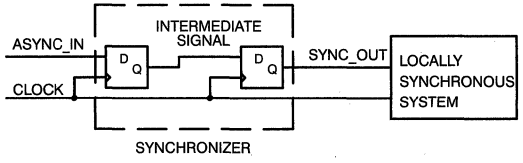


Figure 4. Two-Stage Synchronizer

Explanation of Metastability

In a flip-flop, a metastable output is undefined or oscillates between HIGH and LOW for an indefinite time due to marginal triggering of the circuit. This anomalous flip-flop behavior results when data inputs violate the specified set-up and hold times with respect to the clock.

In the case of a D-type flip-flop, the data must be stable at the device's D input before the clock edge by a time known as the set-up time, t_s. This data must remain stable after the clock edge by a time known as the hold time, t_h (*Figure 5*). The data signal must satisfy both the set-up and hold times to ensure that the storage device (register, flip-flop, latch) stores valid data and to ensure that the outputs present valid data after a maximum specified clock-to-output delay t_{co_max}. As used in this application note, t_{co_max} refers to the interval from the clock's rising edge to the time the data is valid on the outputs. In most cases, t_{co_max} refers to the maximum t_{co} specified by a datasheet, as opposed to the average or typical t_{co} value.

If the data violates either the set-up or hold specifications, the flip-flop output might go to an anomalous state for a time greater than t_{co_max} (*Figure 5*).

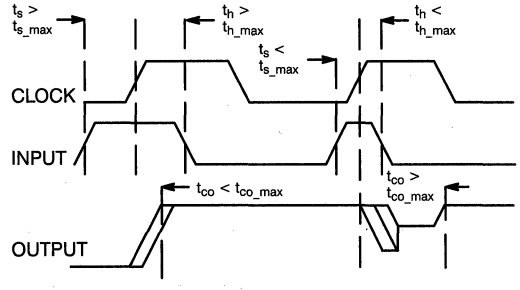


Figure 5. Triggering Modes of a Simple Flip-Flop

The additional time it takes the outputs to reach a valid level can range from a few hundred picoseconds to tens of microseconds. The amount of additional time beyond t_{co_max} required for the outputs to reach a valid logic level is known as the metastable walk-out time. This walk-out time, while statistically predictable, is not deterministic.

Figure 6, from Reference 2, shows the variation in output delay with data input time. The left portion of the graph shows that when the data meets the required set-up time, the device has valid output after a predictable delay, which equals t_{co}. The middle portion of the graph indicates the metastable region. If the data transitions in this region, valid output is delayed beyond t_{co_max}. The closer the input transitions to the center of the metastable region, violating the device's triggering requirements, the longer the propagation delay. If the data transitions after the metastable region, the device does not recognize the input at that clock edge, and no transition occurs at the output. As given in Reference 3, you can predict the region t_w, where data transitions cause a propagation delay longer than t, from the formula:

$$t_w = t_{co}\, e^{\frac{-(t - t_{co})}{\tau}} \qquad \text{Eq. 1}$$

where τ depends on device-specific characteristics such as transistor dimensions and the flip-flop's gain-bandwidth product.

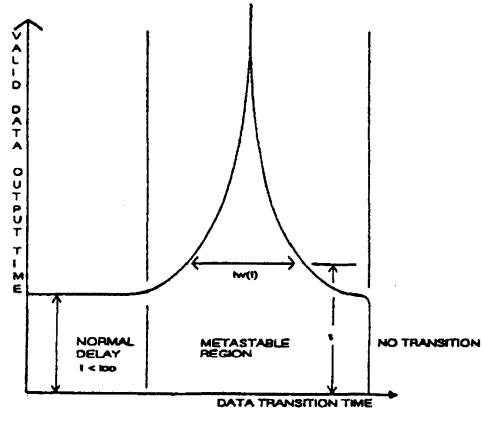


Figure 6. Output Propagation vs. Data Transition

Figure 7. Triggering Modes of a Simple Flip-Flop

Figure 7 shows another way of looking at metastability. A flip-flop, like any other bistable device, has two minimum-potential energy levels, separated by a maximum-energy potential. A bistable system has stability at either of the two minimum-energy points. The system can also have temporary stability—metastability—at the energy maximum. If nothing pushes the system from the maximum-energy point, the system remains at this point indefinitely.

A hill with valleys on either side is another bistable system. A ball placed on top of the hill tends to roll toward one of the minimum-energy levels. If left undisturbed at the top, the ball can remain there for an indeterminate amount of time. As this figure indicates, the characteristics of the top of the hill as well as natural factors affect how long the ball stays there. The steepness of the hill is analogous to the gain-bandwidth product of the flip-flop's input stage.

Causes of Metastability

Systems with separate entities, each running at different clock rates, are called globally asynchronous systems (Reference 4). The entities might include keyboards, communication devices, disk drives, and processors. A system containing such entities is asynchronous because signals between two or more entities do not share a fixed relationship.

Metastability can occur between two concurrently operating digital systems that lack a common time reference. For example, in a multiprocessing system, it is possible that a request for data from one system can occur at nearly the exact moment that this signal is sampled by another part of the system. In this case, the request might be undefined if it does not obey the set-up and hold time of the requested system.

When globally asynchronous systems communicate with each other, their signals must be synchronized. Arbitration must occur when two or more requests

for a shared resource are received from asynchronous systems. An arbiter decides which of two events should be serviced first. A synchronizer, which is a type of arbiter with a clock as one of the arbited signals, must make its decision within a fixed amount of time. A device can synchronize an input signal from an external, asynchronous device in cases such as a keyboard input, an external interrupt, or a communication request.

Care must be taken when two locally synchronous systems communicate in a globally asynchronous environment. A synchronization failure occurs when one system samples a flip-flop in the other system that has an undefined or oscillating output. This event can distribute non-binary signals through a binary system (Reference 5).

In synchronizers, the circuit must decide the state of the data input at the clock input's rising edge. If these two signals arrive at the same time, the circuit can produce an output based on either decision, but must decide one way or the other within a fixed amount of time.

Attacking Metastability

The design of synchronous systems is much different than the design of globally asynchronous systems. The design of a synchronous digital system is based on known maximum propagation delays of flip-flops and logical gates. Asynchronous systems by definition have no fixed relationship with each other, and therefore, any propagation delay from one locally synchronous system to the next has no physical meaning.

Two different methods are available to produce locally synchronous systems from globally asynchronous systems. The first method involves creating self-timed systems. In a self-timed system, the entity that performs a task also emits a signal that indicates the task's completion. This handshaking signal allows the use of the results when they are ready instead of waiting for the worst-case delay. Such handshaking signals allow communications between locally synchronous systems.

The advantage of the self-timed method is that it permits machines to run at the average speed instead of the worst-case speed. The disadvantages are that a self-timed system must have extra circuitry to compute its own completion signals and extra circuitry to check for the completion of any tasks assigned to external entities.

Petri Nets, data flow machines, and self-timed modules all use the self-timed method of communication among locally synchronous systems. Self-timed structures do not completely eliminate metastability, however, because they can include arbiters that can be metastable. Most systems do not include self-timed interfaces due to the additional circuitry and complexity.

The second method of producing locally synchronous systems from globally asynchronous systems is the simple synchronizer. This is the most common way of communicating between asynchronous objects. The metastability errors that might arise from these systems must be made to play an insignificant role when compared with other causes of system failure.

Many metastability solutions involve special circuits (References 6 and 7). Some of these solutions do not reduce metastability at all (References 13 and 8). Others, however, do reduce metastability errors by pushing the occurrence of metastability to a place where sufficient time is available for resolving the error. Most of these circuits are system dependent and do not offer a universal solution to metastability errors.

The easiest and the most widely used solution is to give the synchronizing circuit enough time to both synchronize the signal and resolve any possible metastable event before other parts of the system sample the synchronized output. This solution requires knowledge of the metastable characteristics of the device performing the synchronization.

Many semiconductor companies have developed circuits such as arbiters, flip-flops, and latches that are specifically designed to reduce the occurrence of metastability. Although these parts might have good metastability characteristics, they have very limited application. The circuits can only function as flip-flops or arbiters and do not have the flexibility of PLDs. Cypress Semiconductor has designed the flip-flops in the company's PLDs to be metast-

ability hardened. This allows you to use Cypress PLDs in a wide range of systems requiring synchronization.

Circuit Analysis of Metastability

Many authors have written papers detailing the analysis of metastability from a circuit standpoint (References 5, 7, 8, 9, 10, 11, and 12). In Reference 11, for example, Kacprzak presents a detailed analysis of an RS flip-flop's metastable operation. He states that a flip-flop has two stages of metastable operation (*Figure 8*).

During the initialization phase, the Q and $\overline{Q}$ outputs move simultaneously from their existing levels to the metastable voltage V_m, which is the voltage at which $V_q = \overline{V}_q$.

The second or resolving phase occurs when the outputs once again drift toward stable voltages. Once a flip-flop has entered a metastable state, the device can stay there for an indeterminate length of time. The probability that the flip-flop will stay metastable for an unusually long period of time is zero, however, due to factors such as noise, temperature imbalance within the chip, transistor differences, and variance in input timing. During the second phase of metastability, for very small deviations around the metastable voltage, V_m, the flip-flop behaves like two cross-coupled linear amplifier stages that gain $V_d = V_q - \overline{V}_q$. When the gain of the cross-coupled loop exceeds unity, the differential voltage increases exponentially with time.

The length of time the flip-flop takes to resolve cannot be exactly determined. The probability that the flip-flop will resolve within a specific length of time, however, can be predicted. This probability depends on the electrical parameters of the flip-flop acting as a linear amplifier around the metastability voltage. The solution (Reference 11) to the differential voltage $V_d(t)$ driving the resolving phase is given by

$$V_d(t) = V_d(t_0)\, e^{\frac{(t - t_0)}{\tau}} \qquad \text{Eq. 2}$$

where τ depends directly on the amplifier gain and capacitance, and where $V_d(t_0)$ represents the differential voltage at some time t_0. You can use this equation to determine the length of time that the output voltage will take to drift from the metastable voltage V_m to a specified voltage difference V_d.

Horstmann (Reference 5) states that a flip-flop, like any other system with two stable states, can be described by an energy function with two local energy minima where P(x) = 0 (*Figure 9*). Any bistable system has at least one metastable state, which is an unstable energy level within the system and represents the local maximum of the energy function. The system's gradient can be represented by a force, F(x), that is zero at stable and metastable states (inflection points of the energy function).

Figure 10 shows a simplified first-order model of an RS flip-flop used to predict and visualize metast-

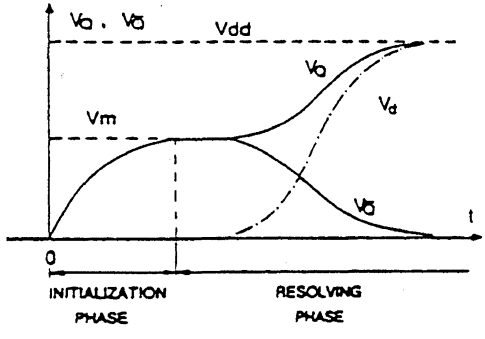


Figure 8. Two Phases of Metastability

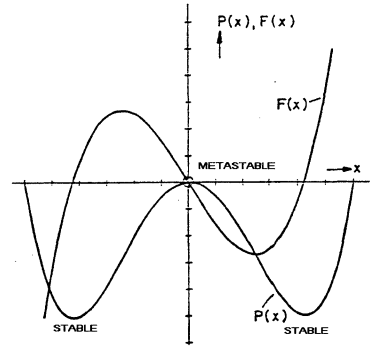


Figure 9. Energy/Force Function of a Bistable System

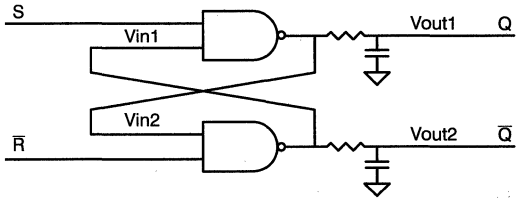


Figure 10. First-Order Flip-Flop Approximation

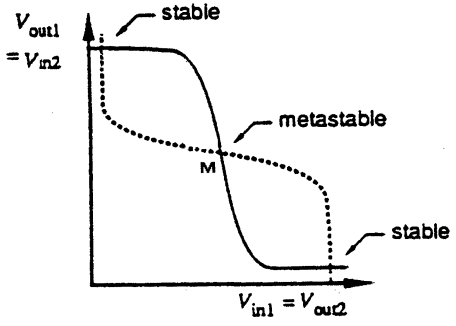


Figure 11. Energy Transfer Diagram of Simple RS Flip-Flop

ability. A flip-flop energy transfer curve (*Figure 11*) shows the relationship between the two outputs. The two stable states are local energy minima of the system. The metastable state, M, is a local energy maximum and represents an unstable state with loop gain near M that is greater than one.

Figure 12 shows the trigger line for the first-order approximation of the flip-flop. The dashed line RS represents the device's normal trigger line, which does not follow the transfer curve because, during triggering, the feedback loop has not been established. If at varying points along the trigger line the feedback loop is re-established, the nodes of the device follow the curves that lead to the line $S_0 - S_1$. Once on this line, the circuit exponentially drifts toward stability at either S_0 or S_1, depending on which side of the line $Q = \overline{Q}$ the feedback loop was re-established. The curves are solutions to the first-order model circuit equations for the device shown in *Figure 10*.

When the feedback loop is restored near the line $Q = \overline{Q}$, the system moves toward the unstable state M

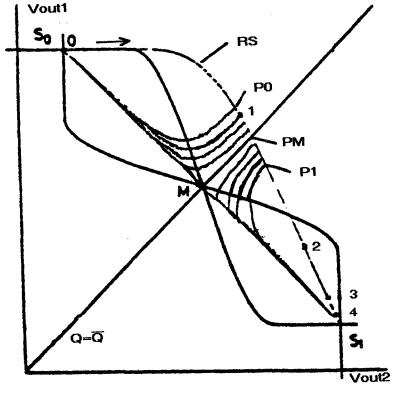


Figure 12. Energy Transfer Curves showing Trigger Paths

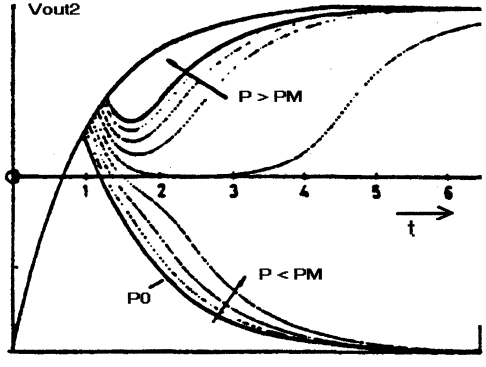


Figure 13. Time Scale Showing Trigger Paths

and can take an indefinite amount of time to exit from this metastable state. You can see this from the graph by noticing that S_0 and S_1 are equally likely solutions for system stability from M. Once the feedback loop is re-established, the system exponentially decays toward M and then exponentially grows toward S_0 or S_1.

Figure 13 shows the system's possible trigger events using the implied time scale of the state-space curves. The solution of these simplified first-order equations indicates that the fastest metastable resolution time occurs when the circuit's gain-bandwidth product is maximized.

Flannagan (Reference 12), in an attempt to maximize the gain-bandwidth product, solves simplified flip-flop equations to determine the phase trajectory near the metastable point. His results, which are supported by other authors, indicate that p and n devices with equal geometries produce the optimal gain-bandwidth product for metastable event resolution.

Statistical Analysis of Metastability

To begin the analysis of metastability, assume that the flip-flop's probability of resolving its metastable state does not depend on its previous metastable state. In other words, the metastable device has no memory of how long it has been in a metastable region. The analysis of metastability also assumes that the flip-flop's probability of resolving its metastable state in a given time interval does not depend on the metastable resolution in another disjoint time interval. The probability that a metastable event will resolve in a given interval (0,t) is only proportional to the length of the interval.

These assumptions yield an exponential distribution that describes the probability that the flip-flop resolves its metastability at a time t. The exponential distribution has the form

$$f(x) = \mu e^{-\mu t} \qquad \text{Eq. 3}$$

where μ is the expected value of metastability resolution per unit time (settling rate).

Using this equation and given that the flip-flop was metastable at time t = 0, the probability of a metastable event lasting a time t or longer is

$$P(met_t \mid met_{t=0}) = \int_t^{\infty} \mu e^{-\mu t} dt = e^{-\mu t} \qquad \text{Eq. 4}$$

The next part of the analysis involves the probability that the flip-flop is metastable at time t = 0. This part of the analysis assumes that the probability that the data transitions in a given time interval depends only on the length of the interval. A Poisson process with rate f_d describes the probability of the data transitioning at a time t:

$$p(x) = \frac{e^{-f_d t} (f_d t)^x}{x!} \qquad \text{Eq. 5}$$

where *x* is the number of transitions.

If a data transition within a bounded time interval, W, of the clock edge causes a metastable condition, the expected number of transitions of this Poisson process with rate f_d in time interval W is

$$E(X) = \sum_{x=0}^{\infty} \frac{x e^{-f_d W} (f_d W)^x}{x!} = f_d W \qquad \text{Eq. 6}$$

Because this expected number of transitions is the same as the probability that the flip-flop is metastable at t = 0, the equation for the probability at t = 0 is

$$P(met_{t=0}) = f_d W \qquad \text{Eq. 7}$$

Using *Equations 5* and *7*, the probability that a given clock cycle results in metastability that lasts at most a time t is

$$P(met_t) = P(met_t \mid met_{t=0}) P(met_{t=0}) = f_d W e^{-\mu t} \qquad \text{Eq. 8}$$

Substituting $\frac{1}{t_{sw}}$ for μ allows this variable to be expressed as a settling time constant of the flip-flop. Further, a synchronization failure for a given clock cycle exists whenever a metastable event lasts a specified time (t_r) or longer. Using these two substitutions, the probability that the flip-flop is metastable in a given clock cycle is

$$P(fail_{1\ clock}) = f_d W e^{\frac{-t_r}{t_{sw}}} \qquad \text{Eq. 9}$$

Because the data transitions are independent, the number of failures in n clock cycles has a binomial distribution with an expected number of failures:

$$E(fail_{n\ cycles}) = n P(fail_{1\ cycle}) \qquad \text{Eq. 10}$$

Assuming a sample clock frequency, f_c, that represents the number of clock cycles, n, per unit time, the expected number of failures per unit time is

$$E(fail_{unit\ time}) = f_c f_d W e^{\frac{-t_r}{t_{sw}}} \qquad \text{Eq. 11}$$

Assuming that all data transitions are independent and that the clock has a fixed period, the mean time between failures (MTBF) is

$$MTBF = \frac{1}{E(fail_{unit\ time})} = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W} \qquad \text{Eq. 12}$$

where MTBF is a measure of how often, on the average, a metastable event lasts a time t_r or longer.

Metastability Data

The strong resemblance between *Equation 12* and *Equation 2* is based on the predictions of the first-order circuit analysis of an RS flip-flop. In fact, the metastability resolving time constant, t_{sw}, is directly related to the variable τ, which is based on the flip-flop's gain-bandwidth product.

The device-dependent variable W depends mostly on the window of time within which the combination of the input and clock generate a metastable condition. This parameter also depends on process, temperature, and voltage levels. The MTBF equation is usually plotted with t_r (the resolving time allowed for metastable events) on the X axis and the natural log of the MTBF plotted on the Y axis (see the appendix in this note). Because the metastability equation is plotted on a semi-log scale, the graph of t_r vs ln(MTBF) is a line described by the equation

$$\ln(MTBF) = \frac{t_r}{t_{sw}} - \ln(f_c f_d W) \qquad \text{Eq. 13}$$

Graphically, the parameter t_{sw} is 1/slope of the line on this graph. The equation for t_{sw} from the graph is

$$t_{sw} = \frac{t_{r1} - t_{r2}}{\ln(MTBF_1) - \ln(MTBF_2)} \qquad \text{Eq. 14}$$

To determine how often, on the average, a given synchronizer in a system will go metastable (MTBF), you must know the two device-specific parameters W and t_{sw}, which should be available from the manufacturer. *Table 5*, discussed later in this note, lists these values for Cypress PLDs. Additional values you need are the average frequency of both the system data and the synchronizer clock and the amount of time after the synchronizer's maximum clock-to-Q time that is allowed to resolve metastable events.

For example, consider the method for determining the MTBF for a Cypress PALC22V10 registered PLD used as a synchronizer in a system with the following characteristics:

W = 0.125 ps

t_{sw} = 190 ps

f_c = system clock frequency = 25 MHz

f_d = average asynchronous data frequency = 10 MHz

In addition to these values, the PLD's maximum operating frequency, f_{max}, is taken directly from the datasheet. The frequency is specified as the internal feedback maximum operating frequency. It is calculated as

$$f_{max} = \frac{1}{t_{cf} + t_s} = 41.6\ MHz$$

where t_{cf} is the clock-to-feedback time. If the data sheet does not specify t_{cf}, you can use t_{co} as t_{cf}'s upper bound.

Using f_{max}, you calculate the amount of time that a metastable event is allowed to resolve, t_r, with

$$t_r = \frac{1}{f_c} - \frac{1}{f_{max}} = \frac{1}{25\ MHz} - \frac{1}{41.6\ MHz} = 16\ ns$$

Now you enter these values into the MTBF equation, making sure to keep all units in seconds:

$$MTBF = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W}$$

$$= \frac{e^{\frac{16 \times 10^{-9}\ s}{190 \times 10^{-12}\ s}}}{25 \times 10^6 s^{-1} \times 20 \times 10^6 s^{-1} \times 0.125 \times 10^{-12} s}$$

$$= 59.7 \times 10^{33} s$$

$$= 1.89 \times 10^{27}\ years = Almost\ forever$$

If the operating frequency of the system, f_c, is simply changed to 33.3 MHz,

$$MTBF = \frac{e^{\frac{6 \times 10^{-9} s}{190 \times 10^{-12} s}}}{33.3 \times 10^6 s^{-1} \times 20 \times 10^6 s^{-1} \times 0.125 \times 10^{-12} s}$$

$$= 623 \times 10^9\ s$$

the system fails, on the average, about every 19,700 years—still beyond the system's normal lifetime.

And if f_c is changed to f_{max} (41.6 MHz),

$$MTBF = \frac{e^{\frac{0 \times 10^{-9} s}{190 \times 10^{-12} s}}}{41.6 \times 10^6 s^{-1} \times 20 \times 10^6 s^{-1} \times 0.125 \times 10^{-12} s}$$

the system fails, on the average, every 9.62 ms.

A 16-ns difference in resolve time, t_r, results in almost 36 orders of magnitude difference in MTBF. Obviously, accurate data is needed to design a system with a high degree of reliability without being overly cautious.

Characterization of Metastability

Many authors (References 6, 8, 9, 10, 11, and 12) have performed numerous experiments on circuits to predict the likelihood of device metastability. These researchers have used several testing theories and apparatus that can be classified into three basic types (Reference 14).

Intermediate voltage sensors constitute the first type. Two voltage comparators determine whether the output voltage, Q, lies between two given voltages. The fixture produces an error output if Q has a level that is neither HIGH nor LOW, hence metastable. *Figure 14* shows an intermediate voltage sensor.

The second type of apparatus uses an output proximity sensor to determine if the Q and $\overline{Q}$ outputs have approximately the same voltages, which would indicate that the device is metastable. *Figure 15* shows an output proximity sensor.

The last type of apparatus uses a late-transition sensor to test for metastability. Note that if one or more gates separate the sensor from the metastable signal, the metastability might not be detected. The test circuitry must infer the occurrence of metastability by some other means. *Figure 16* shows an example of a late-transition sensor. The sample input is detected at time t_1, then at a later time t_2. If these two signals disagree, the device under test was metastable at t_1.

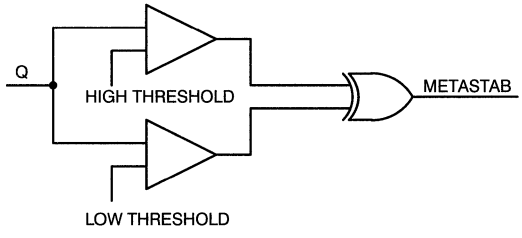


Figure 14. Intermediate Voltage Sensor

Information from Manufacturers

Many semiconductor companies provide metastability data on their parts. However, most companies do not present the data in a format the engineer can use. They either present inconclusive and incomplete data or they assume the engineer can use the data without further explanation. Few companies compare their devices with similar devices.

PLD manufacturers provide little data largely because of a fear that telling the design community that devices can fail in synchronizing applications will cause designers to use a competitor's parts. The truth is that no company can provide a device that is guaranteed never to become metastable when used as a synchronizer. At a given operating frequency, with a given asynchronous input, and given enough time, the device becomes metastable.

Cypress provides you with data you can use to build a system to any given level of reliability when using Cypress PLDs. Cypress has performed numerous tests and collected extensive data on Cypress PLDs, as well as PLDs from other companies. This data

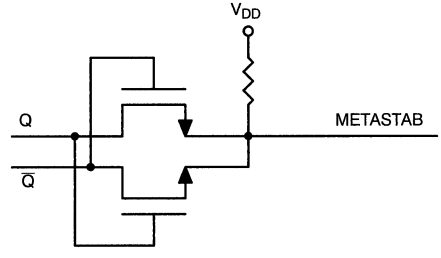


Figure 15. Output Proximity Sensor

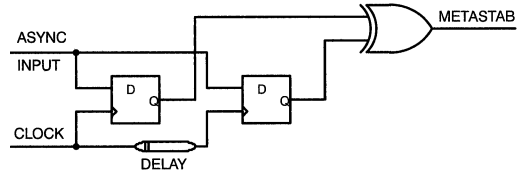


Figure 16. Late Transition Sensor

gives you a perspective of the parts that are best suited for a specific application. Specific data on the metastability characteristics of Cypress PLDs is found in this application note in the Test Results section. Metastability data collected by Cypress for other companies' PLDs is available upon request.

The Test Circuit

Cypress uses a test that falls into the category of the late-transition detection. Directly measuring the outputs of the flip-flop in a PLD are impossible due to the additional circuitry that lies between the flip-flop and the outside world. The metastability detection circuitry must, instead, infer the flip-flop's state.

Figure 17 shows the metastability test circuit implemented in each test PLD. This circuit allows the PLD under test to effectively test itself. The device under test will both produce and record metastable conditions.

Figure 18 is a state diagram showing the operation of the device. During normal operation, the two flip-flops' outputs (F_1, F_2) transition between states S_1 and S_2, depending on the synchronizer's state. During normal operation, the Exclusive-OR on these

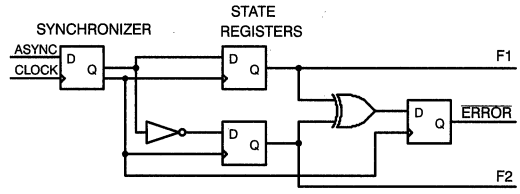


Figure 17. Metastability Test Circuit

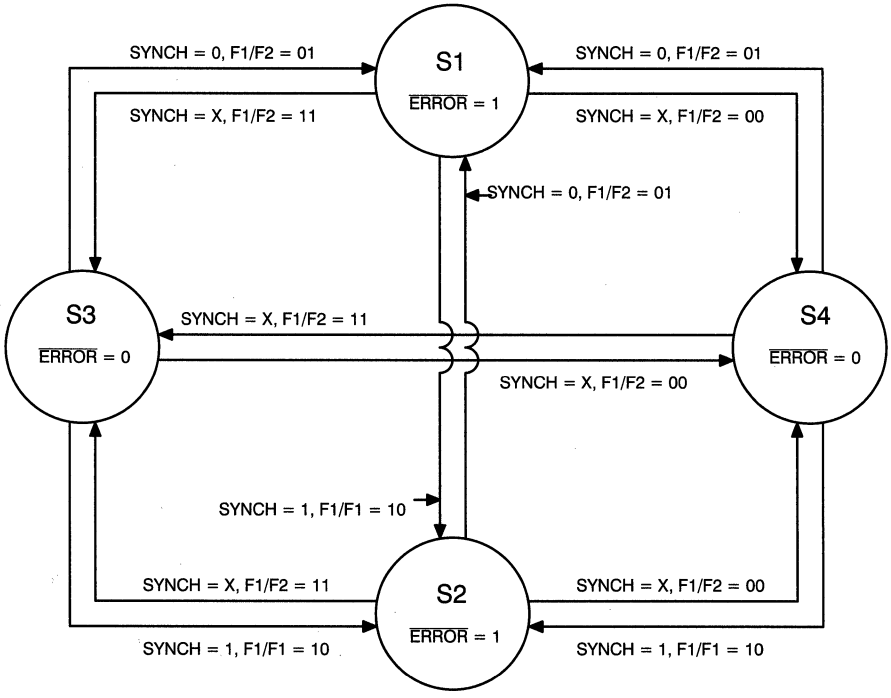


Figure 18. Metastability Testing State Diagram

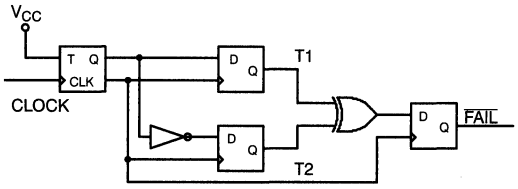


Figure 19. Maximum Operating Frequency Test

outputs produces a HIGH. This indicates either that metastability has not occurred within the device or that metastability that has occurred has resolved before the next clock cycle.

If a metastable event cannot resolve before the next clock cycle, the state machine move to states S_3 or S_4. In this case, the state flip-flops have interpreted the signal from the synchronization register differently; exclusive-ORing this signal produces a LOW at the device's output, indicating that unresolved metastability has occurred.

This test circuit does not catch all metastable events. Specifically, it does not record metastable events that resolve before the next clock cycle. But metastability causes an error only when it has not resolved by the time the signal is needed. The Cypress tests thus reveal the information designers need to know: how often metastability creates an error in the system.

The test circuit also includes the ability to check the maximum operating frequency of the device under test (*Figure 19*). At each clock edge, the first register's output toggles. When the device reaches its maximum operating frequency, the PLD array cannot resolve the changing signal fast enough to produce a valid output. At this speed, one register might resolve the signal correctly and one might not, or both might produce invalid signal resolutions. In any case, when Exclusive-ORing the state T_1/T_2 of the two maximum-frequency testing registers results in anything other than a HIGH, the part's maximum operating frequency is exceeded.

The Test Board

A four-layer printed circuit board with two signal planes, a ground plane, and a power plane is used to

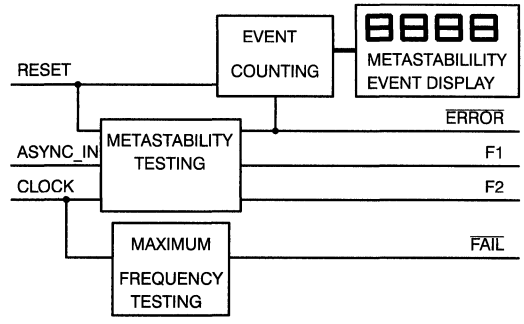


Figure 20. Metastability Test Board Block Diagram

perform the metastability measurements. Using this four-layer board gives a quiet testing environment with reliable, repeatable results. *Figure 20* shows a block diagram of the test board, with the complete schematic shown in *Figure 21*. The device under test (DUT) is decoupled with 0.01-µF and 100-pF capacitors. The test circuit is designed to fit all industry-standard and Cypress-proprietary PLDs. The socket allows DUT pins 1, 2, and 4 to serve as clock pins. Pin 3 is the device's asynchronous input. The $\overline{\text{ERROR}}$ condition is located on pin 27 of a 28-pin device, and the $\overline{\text{FAIL}}$ condition is on pin 20. Two additional outputs, F_1 and F_2, monitor the state of the metastability test circuit flip-flops.

All inputs and outputs connect with BNC connectors located around the board. The clock line, which is terminated with a 50Ω resistor to match the coax input impedance, is buffered with a 74AS04 and isolated from other signals by a ground trace. The input line is also terminated with a 50Ω resistor and buffered with a 74AS04. Four PLDs drive a four-digit LED display that counts metastability occurrences.

After going LOW in response to a metastable event, the $\overline{\text{ERROR}}$ signal automatically transitions HIGH again at the next system clock. This LOW-to-HIGH pulse produces a clock to the input of the first PLD, which in turn increments the display of metastable events. When a digit reaches 9, the next occurrence of metastability generates a cascade signal to the next higher digit.

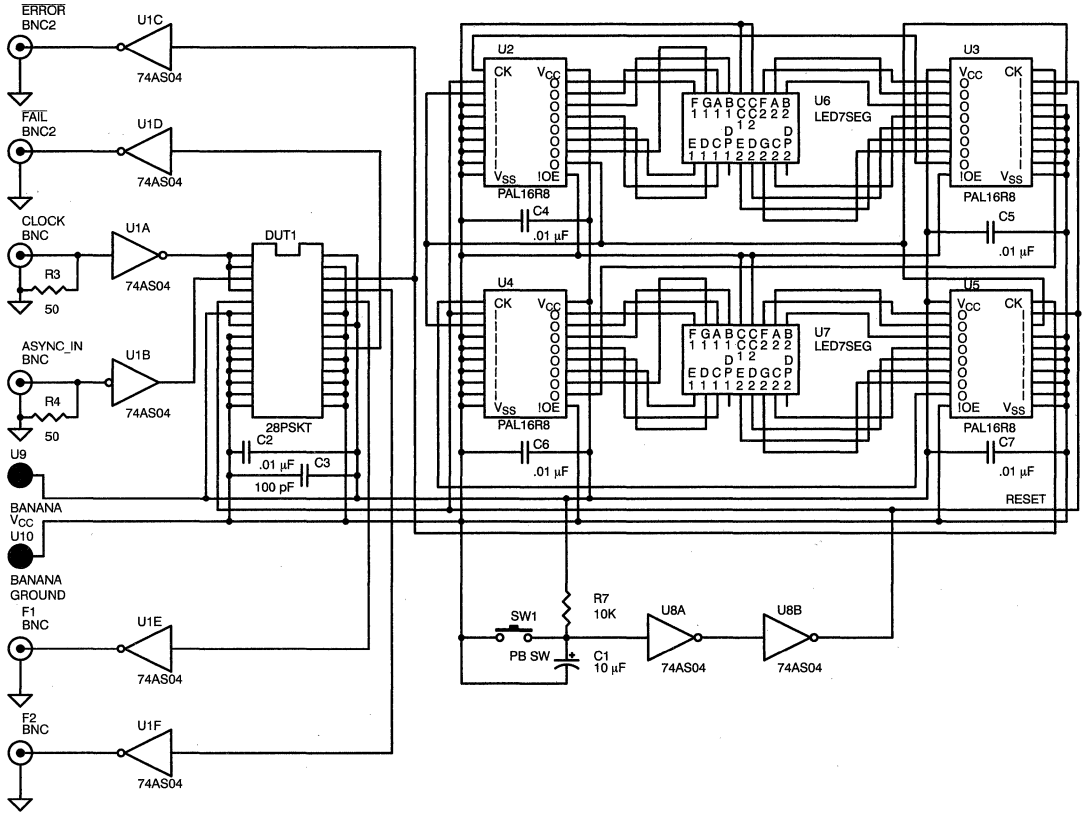


Figure 21. Metastability Test Board Schematic

In this way, the test board can record a maximum of 9,999 metastable events. If a metastable event is received at 9,999, all LEDs switch to E, indicating that an overflow condition occurred. A reset button resets all counters and initializes the DUT.

Test Set-Up

Figure 22 shows a block diagram of the test set-up used for metastability testing. Two independent pulse generators (Hewlett-Packard 8082As) produce the CLOCK and the ASYNC_IN signal to the test board. A Tektronix DAS9200 logic analyzer records metastable events. A 2465 CTS digital oscilloscope with frequency counter accurately determines the DUT's maximum operating frequency and the ASYNC_IN and CLOCK frequencies.

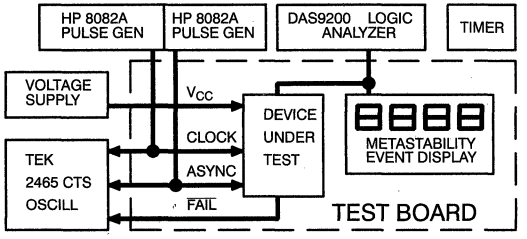


Figure 22. Metastability Test Set-Up

Test Procedure

Cypress has tested all its 20-, 24-, and 28-pin PLDs. The fastest speed grades of each device type were tested because these devices have the best metastable resolution time and thus make the best synchronizers. Several parts from each device type were tested to ensure an average metastability characteristic for that product. Where possible, parts from different date codes were selected to eliminate variations among different wafer lots.

Testing for a specific device starts by creating the high-level description written in VHDL to be used with the *Warp2* VHDL Compiler. *Figures 23 and 24* list the behavioral description used for generating a JEDEC file. All devices were programmed using JEDEC files generated by *Warp2*, except for the CY7C344. The MAX+PLUS development environment was used to produce a design file for this device.

Each part is programmed, then tested for its maximum operating frequency, f_{max}. By attaching the $\overline{FAIL}$ output to the oscilloscope and observing the clock frequency at which the device started to malfunction ($\overline{FAIL}$ going LOW periodically), the maximum operating frequency for that part is determined. f_{max} indicates the maximum rate at which metastability measurements can be taken with accurate results. Above this frequency, metastable events are indistinguishable from errors caused by exceeding f_{max}.

To determine each device's metastability characteristics, measurements are taken of the number of metastable events that occurred in a given time interval for several different clock and data frequencies.

Equation 13 can be used to describe the graph of the metastability characteristics of the device:

$$\ln(MTBF) = \frac{t_r}{t_{sw}} - \ln(f_c f_d W)$$

The slope of the line, t_{sw}, can be determined only by forcing the Y intercept of the graph ($\ln(f_c f_d W)$) to a constant value when using *Equation 14*:

$$t_{sw} = \frac{t_{r1} - t_{r2}}{\ln(MTBF_1) - \ln(MTBF_2)}$$

Note that t_{sw} is a constant, device-specific parameter.

Because W is also a constant, device-specific parameter, it is only necessary to hold the product $f_c f_d$ constant to make $\ln(f_c f_d W)$ constant. The independent variable t_r is varied by changing f_c to produce changes in the dependent variable ln(MTBF). Decreasing the frequency f_c from its f_{max} value increases the metastable resolution time, t_r, and decreases the probability that a metastable event will last longer than t_r.

As f_c is decreased below a certain limit, the MTBF becomes too large to measure accurately. A metastable event occurring every minute is chosen as the upper limit for MTBF measurements. The range of clock rates for metastability testing is then between f_{max} and the metastable-event-per-minute clock rate. Between these two rates, a selected frequency constant ($f_c f_d$) ensures that no point in this range has a clock frequency less than twice the data frequency. This is because a data signal that transitions more than once per clock period cannot be effectively sampled.

After determining this constant, data is taken from several test points within the test range by varying f_c and f_d. The data at each test point is averaged among all test devices, and the equation for the line through these points is determined using a linear regression analysis. The correlation between the line and the data points verifies that the metastability equation accurately describes the test data. From the calculated results, the constants W and t_{sw} are extracted.

Test Results

Table 5 and the Appendix list the results of the metastability analysis of Cypress PLDs. *Table 5* also lists the maximum data book operating frequency, f_{max}; the metastability equation constants, W and t_{sw}; the metastability resolve time, t_r, required for a 10-year MTBF; and the process for that part.

You can use this data to determine the maximum metastability resolve time (t_r) that you must use in a system to yield a given degree of reliability. The graphs and constants (W and t_{sw}) can be used with any speed grade of the device, but it is suggested that the fastest speed grade of the specific PLD be used

```
package test is
  component metastability port (
    clock, async_in, reset : in bit;
      fail, perror, f1, f2 : out bit);
  end component;
end test;

entity metastability is port (
  clock, async_in, reset : in bit;
    fail, perror, f1, f2 : out bit);
end metastability;

use work.bv_math.all

architecture fsm of metastability is

signal  sync  : bit;
signal  tsync : bit;
signal  t1,t2 : bit;
signal  f1_tmp, f2_tmp : bit;
signal  error_tmp : bit;
signal  fail_tmp : bit;

begin
  proc1: process begin

    wait until clock = '1';
    sync <= async_in;

  end process;

  proc2: process begin

    wait until clock = '1';
    f1_tmp <= sync;
    f2_tmp <= inv(sync);

  end process;

  proc3: process begin

    wait until clock = '1';
    error_tmp <= ((((inv(reset) and f1_tmp) and inv(f2_tmp))
      or ((inv(reset) and inv(f1_tmp)) and f2_tmp))
      or (reset and inv(error_tmp)));

  end process;
```

Figure 23. *Warp2* VHDL Behavioral Description for Metastability Testing

```
  proc4: process begin

    wait until clock = '1';
    if (tsync = '1') then
      tsync <= '0';
    else
      tsync <= '1';
    end if;

  end process;

  proc5: process begin

    wait until clock = '1';
    t1 <= tsync;
    t2 <= inv(tsync);

  end process;

  proc6: process begin

    wait until clock = '1' ;
    fail_tmp <= (t1 xor t2);

  end process;

fail <= inv(fail_tmp);
perror <= inv(error_tmp);
f1 <= inv(f1_tmp);
f2 <= inv(f2_tmp);

end fsm;
```

Figure 23. *Warp2* VHDL Behavioral Description for Metastability Testing (continued)

for optimum synchronizer performance. These graphs indicate the time (t_r) and the device's minimum clock period that must be used to produce a desired degree of reliability.

For example, to determine the operating parameters of the Cypress PALC22V10−20 from *Table 5* when using the device as a synchronizer, determine the desired MTBF. With a 10-yr (315×10^6s) MTBF, for instance, a synchronization failure will occur once every 10 years on the average. The maximum operating frequency (f_{max}) from the PALC22V10's data sheet is 41.6 MHz. From this information, you can determine the minimum time (t_r) beyond the device's minimum operating period that must be added for metastability resolution:

$$MTBF = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W}$$

$$t_r = t_{sw}(\ln(MTBF) + \ln(f_c f_d W))$$

$$t_r = (0.190 \times 10^{-9}s)[\ln(315 \times 10^6 s) + \ln(41.6 \times 10^6 \times 41.6 \times 10^6 \times 0.125 \times 10^{-12})]$$

$$= 4.73\ ns$$

This analysis assumes that the clock, f_c, operates at f_{max} (41.6 MHz) and that the average asynchronous data frequency is no more than half the clock fre-

quency. The latter condition ensures effective data sampling by the synchronizer. f_d, as explained in the Statistical Analysis of Metastability section represents the rate at which the data changes state. f_d is twice the average frequency of the asynchronous data input because, during any given asynchronous data period, the asynchronous data changes state twice: once from LOW to HIGH and again from HIGH to LOW. Because either of these state changes can cause a metastable event, f_d must be set to twice the average asynchronous data frequency when determining the worst-case MTBF.

Due to the real-world uncertainty in factors such as trace delays and the skew in clock generators, 5 ns is used instead of 4.73 ns for t_r. The synchronizer's maximum operating frequency, f_c, in this system is then

$$f_c = \frac{1}{t_s + t_{cf} + t_r} = \frac{1}{10ns + 12ns + 5ns} = 37.0\ MHz$$

The effective MTBF using these new values for t_r and f_c is

$$MTBF = \frac{\frac{5\times10^{-9}s}{e^{0.190\times10^{-9}s}}}{37.0 \times 10^{6}s^{-1} \times 37.0 \times 10^{6}s^{-1} \times 0.125 \times 10^{-12}s}$$

$$= 1.57 \times 10^{9} = 49.7\ yrs$$

Another example focuses on the CY7C330−50 used as a synchronizer in a system whose output registers are clocked at an f_c of 35.7 MHz, and the data has an average frequency of 10 MHz. The MTBF for this device used as a synchronizer is calculated by first determining the metastable resolution time, t_r, allowed for synchronization. The maximum operating frequency of the part is specified in Cypress's *Data Book* as

$$f_{max} = \frac{1}{t_{co} + t_{is}}$$

where t_{CO} in this case specifies the clock-to-feedback delay, and t_s specifies the set-up time of the output registers. t_r is calculated with the equation:

$$t_r = \frac{1}{f_c} - \frac{1}{f_{max}} = \frac{1}{35.7\ MHz} - \frac{1}{50.0\ MHz} = 8\ ns$$

With this result, the MTBF is

$$MTBF = \frac{\frac{8\times10^{-9}s}{e^{0.290\times10^{-9}s}}}{35.7 \times 10^{6}s^{-1} \times 20.0 \times 10^{6}s^{-1} \times 1.02 \times 10^{-12}s}$$

$$= 1.31 \times 10^{9}\ s = 41.6\ yrs$$

This equation uses the same values for W and t_{sw} with this 50-MHz device as with the 66-MHz device listed in *Table 5*. As stated previously, the constants listed in *Table 5* are valid for all speed grades of a specific device. Also note that the 10-MHz average data frequency is doubled to produce the frequency of data transitions, f_d.

The last example illustrates how to use a Cypress PALC22V10C−10 as a synchronizer. For a 10-year MTBF, assuming the maximum f_c from Cypress's *Data Book* and f_d, the required t_r is

$$t_r = (0.547 \times 10^{-9}s)\,[\ln(315 \times 10^{6}s) + \ln(90.9 \times 10^{6} \times 90.9 \times 10^{6} \times 8.08 \times 10^{-15})]$$

$$= 13.0\ ns$$

Using this result, the synchronizer's maximum operating frequency is reduced from 90.9 MHz to

$$f_c = \frac{1}{\frac{1}{f_{max}} + t_r} = \frac{1}{\frac{1}{90.9\ MHz} + 13.0\ ns} = 41.6\ MHz$$

Two-Stage Synchronization

As explained earlier, you can use a second register in series to perform two-stage synchronization (*Figure 4*). This is accomplished by feeding the output of the first synchronization register to the input of the second synchronization register. In PLDs, this method is common because the first synchronization stage can synchronize the asynchronous input signal, and the second synchronization stage can perform a Boolean function on a combination of the input and output signals. Boolean functions can be performed at either stage; the metastability characteristics listed in *Table 5* apply to PLD registers' asynchronous inputs that are used directly as well as asynchronous inputs used as a Boolean combination of existing inputs and outputs.

Table 5. Metastability Characteristics of Cypress PLDs

Device	f_{max} (MHz)	W (s)	t_{sw} (s)	t_r for 10-yr MTBF (ns)
PALC16R8–25	28.5	9.503E–12	0.515E–9	14.68
PAL16R8–5	125	94.48E–12	0.299E–9	9.48
PALC20G10–20	41.6	3.730E–12	0.173E–9	4.91
PALC20RA10–15	33.3	2.860E–12	0.216E–9	5.87
PAL22V10C–7	111	0.389E–12	0.546E–9	15.50
PAL22V10CF–7	111	0.398E–12	0.570E–9	16.21
PALC22V10D–7	100	32.35E–12	0.347E–9	10.56
PALC22V10B–15	50.0	55.76E–12	0.261E–9	8.19
PALC22V10–20	41.6	0.125E–12	0.190E–9	4.73
CY7C330–66	66.6	1.020E–12	0.290E–9	8.12
CY7C331–20	31.2	0.298E–9	0.184E–9	5.91
CY7C335–100	58.8	0.288E–12	0.189E–9	4.95
CY7C344–20	41.6	0.966E–9	0.223E–9	7.55

When implementing a two-stage synchronizer in a PLD, the probability that a synchronizer is metastable after the second stage of synchronization is the square of the probability that a synchronizer is metastable after the first stage of synchronization. The MTBF equation is

$$MTBF = \left(\frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W} \right)^2$$

From this result, the equation for t_r becomes

$$t_r = \frac{t_{sw}(\ln(MTBF) + 2 \times \ln(f_c f_d W))}{2}$$

Using this result for a two-stage synchronizer in a Cypress PALC22V10C, the t_r for a 10-year MTBF is reduced from 13.0 ns to

$$t_r = (0.5)(0.547 \times 10^{-9} s)\,[\ln(315 \times 10^6 s) + \ln(90.9 \times 10^6 \times 90.9 \times 10^6 \times 8.08 \times 10^{-15})]$$
$$= 7.65\ ns$$

The maximum f_c increases from 41.6 MHz to

$$f_c = \frac{1}{\frac{1}{f_{max}} + t_r} = \frac{1}{\frac{1}{90.9\ MHz} + 7.65\ ns} = 53.6\ MHz$$

This example shows that if the cycle of latency caused by the additional synchronization stage is acceptable, you can dramatically increase the synchronizer's maximum operating frequency.

References

1. Lubkin, S., (Electronic Computer Corp.), "Asynchronous Signals in Digital Computers," *Mathematical Tables and Other Aids to Computation*, Vol. 6, No. 40, October 1952, pp. 238–241.
2. Nootbaar, Keith, (Applied Microcircuits Corp.), "Design, Testing, and Application of a Metastable-Hardened Flip-Flop," WESCON 87 (San Francisco, CA, Nov. 17–19, 1987), Electronic Conventions Management, Los Angeles, CA 90045.
3. Stoll, Peter A., "How to Avoid Synchronization Problems," *VLSI Design*, November/December 1982, pp. 56–59.
4. Chapiro, Daniel M., *Globally-Asynchronous Locally-Synchronous Systems*, Stanford University, Department of Computer Science Report No. STAN–CS–84–1026, October 1984.
5. Horstmann, Jens U., Eichel, Hans W., Coates, Robert L., "Metastability Behavior of CMOS

ASCI Flip-Flops in Theory and Test," *IEEE Journal of Solid-State Circuits*, Vol. 24, No. 1, February 1989, pp. 146–157.

6. Wormald, E.G., "A Note on Synchronizer or Interlock Maloperation," *Professional Program Session Record 16*, WESCON 87, November 17–19, 1987, Electronic Conventions Management, Los Angeles, CA 90045.

7. Pechouchek, Miroslav, "Anomalous Response Times of Input Synchronizers," *IEEE Trans. Computers*, Vol. C–25, No. 2, February 1976, pp. 133–139.

8. Chaney, T. J., "Comments on 'A Note on Synchronizer or Interlock Maloperation," *IEEE Trans. Computing*, Vol. C–28, No. 10, Oct. 1979, pp. 802–804.

9. Couranz, George R., Wann, Donald F., "Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region," *IEEE Trans. Computers*, Vol. C–24, No. 6, June 1975, pp. 604–616 .

10. Veendrick, Harry J.M., "The Behavior of Flip-Flop Used as Synchronizers and Prediction of Their Failure Rate," *IEEE Journal of Solid-State Circuits*, Vol. SC–15, No. 2., April 1980, pp. 169–176.

11. Kacprzak, Tomasz, Albicki, Alexander, "Analysis of Metastable Operation in RS CMOS Flip-Flops," *IEEE Journal of Solid-State Circuits*, Vol. SC–22, No. 1, February 1987, pp. 57–64.

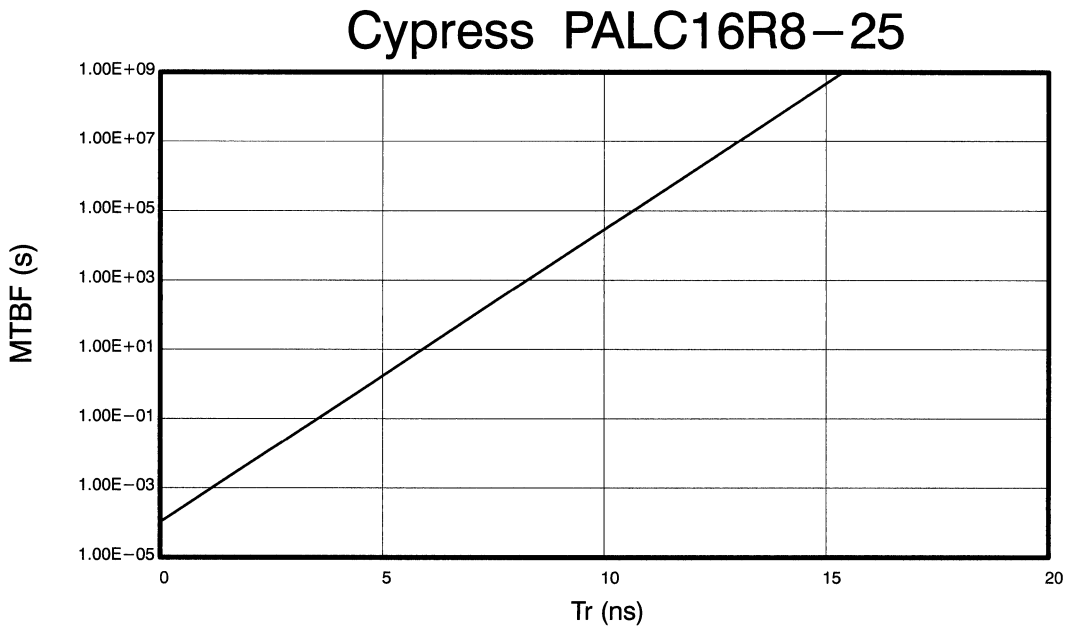
12. Flannagan, Stephen T., "Synchronization Reliability in CMOS Technology," *IEEE Journal of Solid-State Circuits*, Vol. SC–20, No. 4, Aug 1985, pp. 880–882.

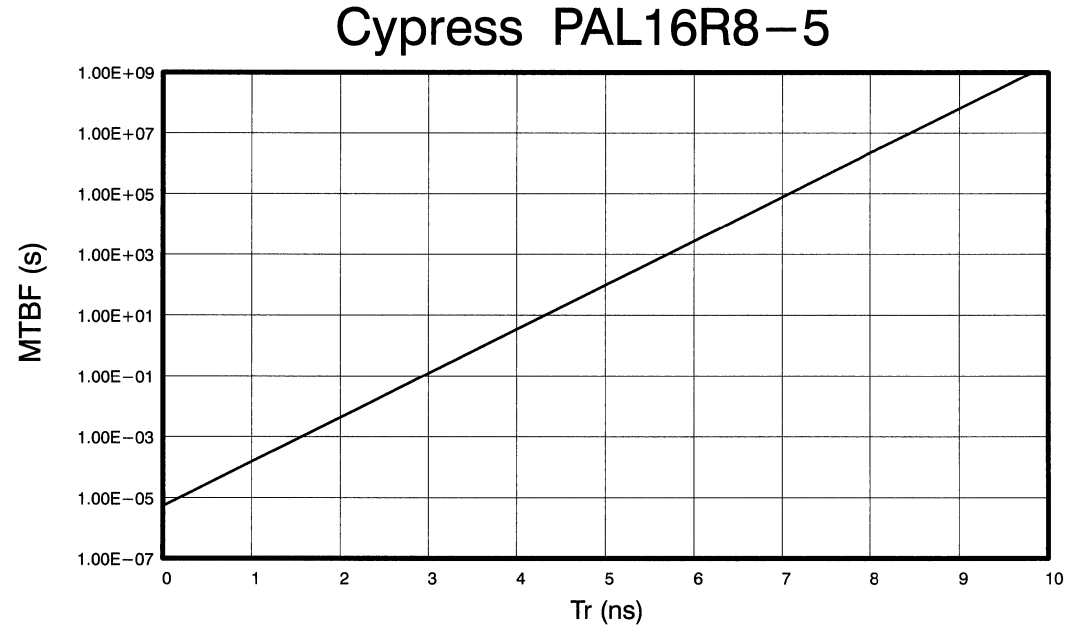
13. Wakerly, John F., *A Designers Guide to Synchronizers and Metastability*, Center for Reliable Computing Technical Report, CSL TN #88–341, February, 1988 Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA.

14. Freeman, Gregory G., Liu, Dick L., Wooley, Bruce, and McClusky, Edward J., *Two CMOS Metastability Sensors*, CSL TN# 86–293, June 1986, Computer Systems Laboratory, Electrical Engineering and Computer Science Departments, Stanford University, Stanford, CA.

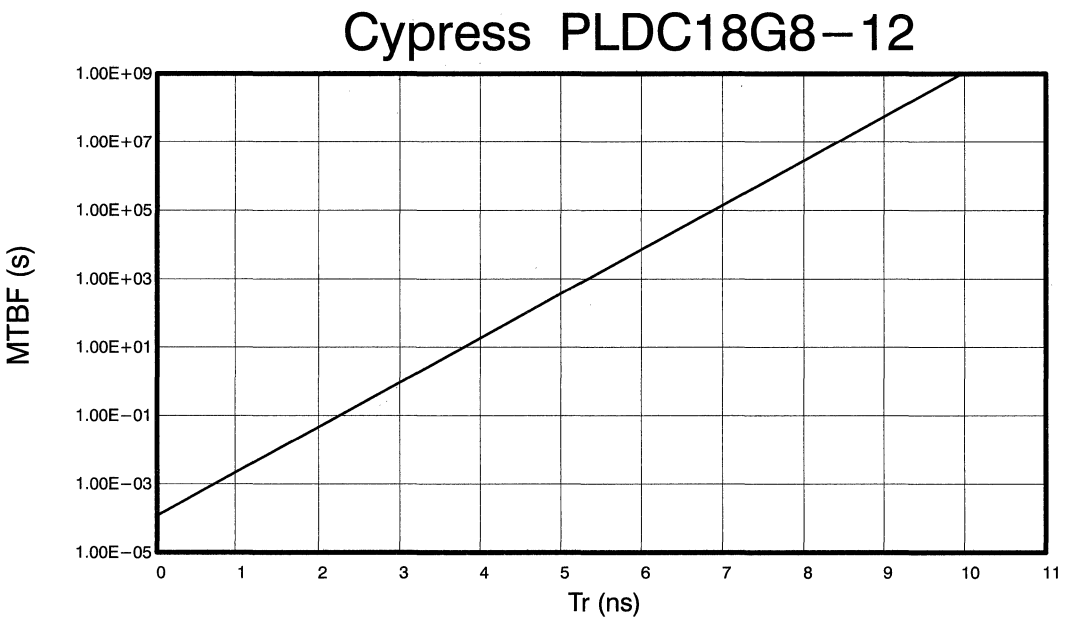
15. Rubin, Kim, "Metastability Testing in PALs," WESCON 87 (San Francisco, CA, Nov. 17–19, 1987), Electronic Conventions Management, Los Angeles, CA 90045.

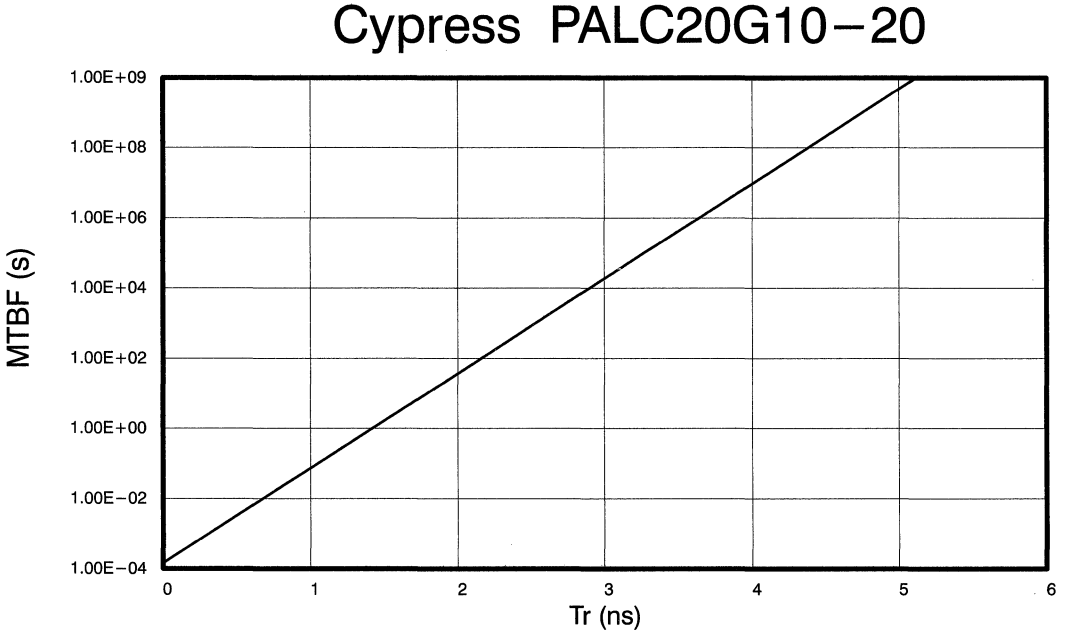
Appendix A. Metastability Graphs of Cypress Devices



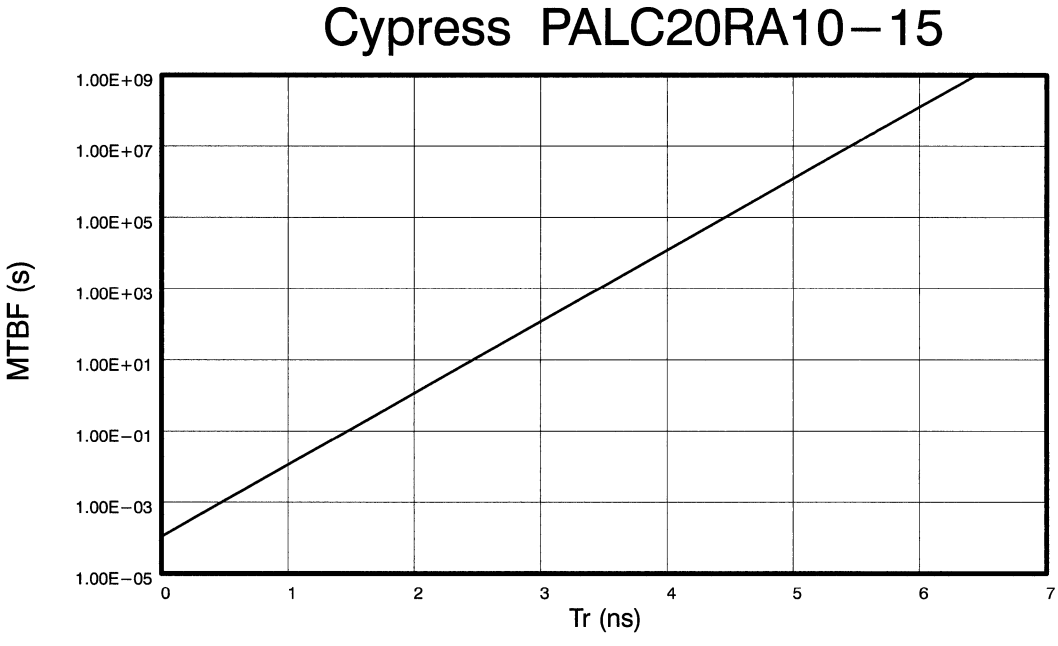


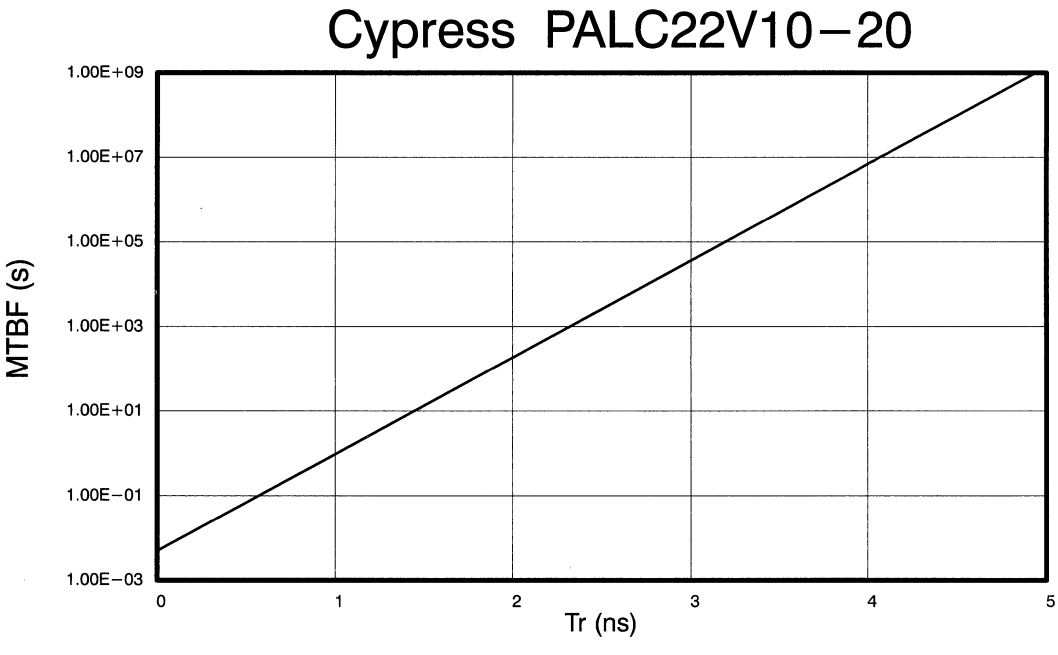
Appendix A. Metastability Graphs of Cypress Devices (continued)



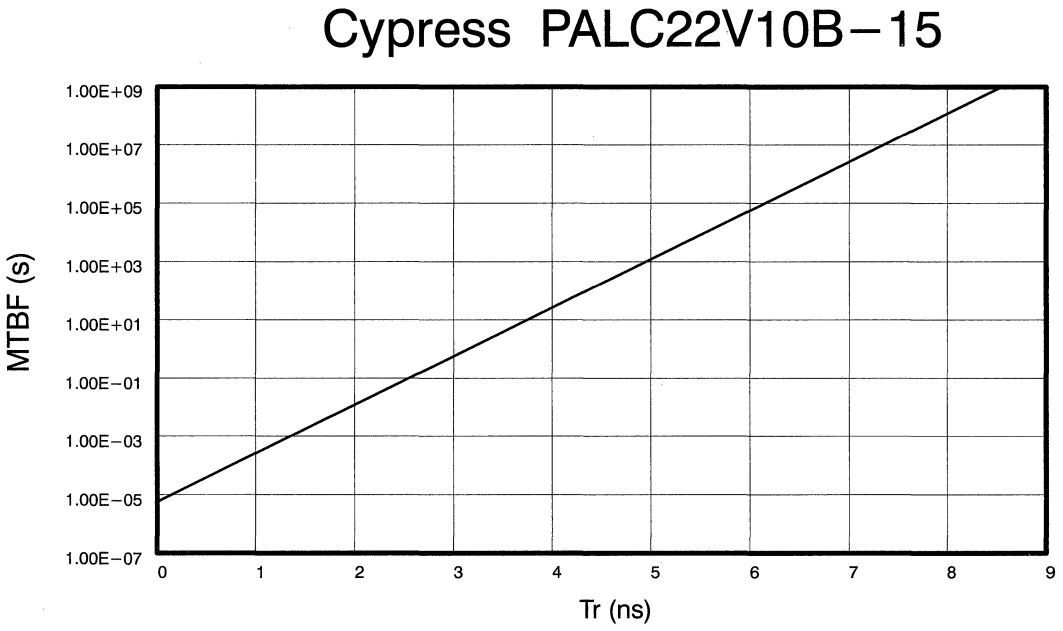


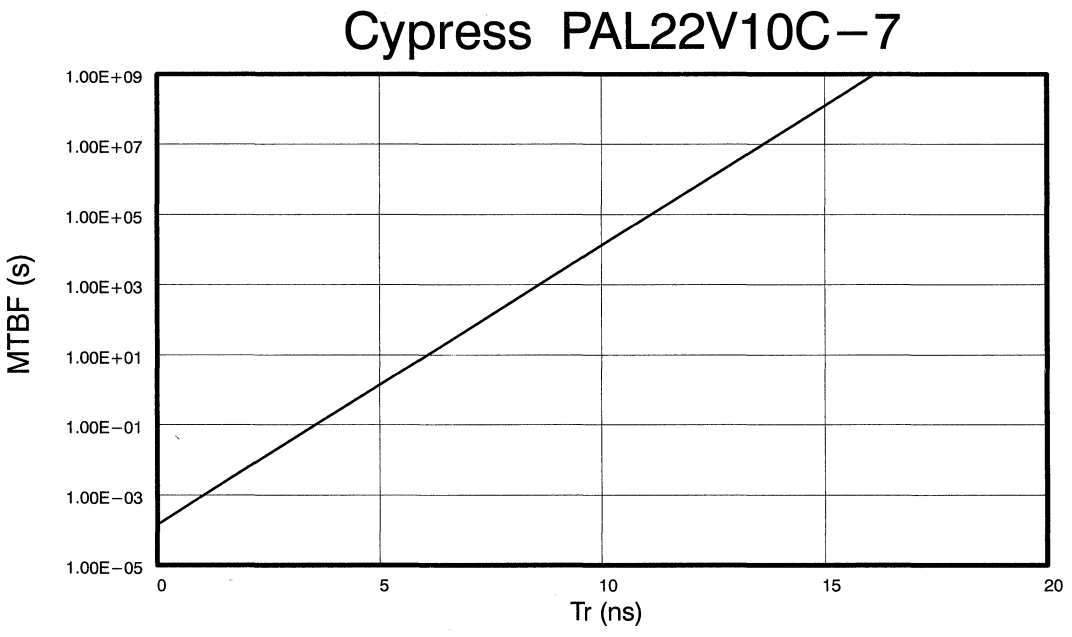
Appendix A. Metastability Graphs of Cypress Devices (continued)



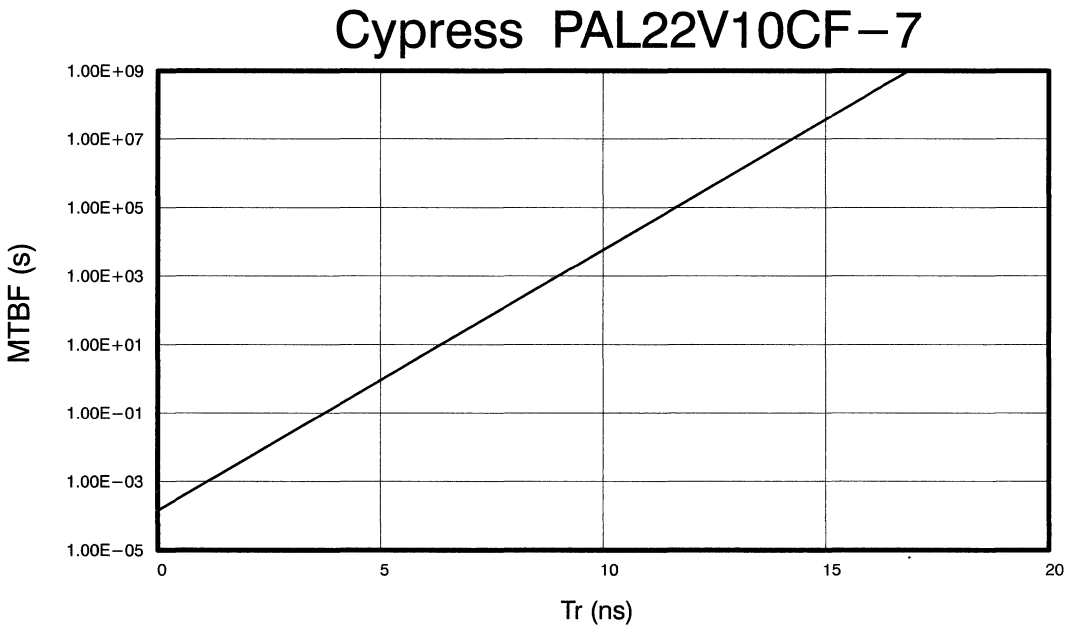


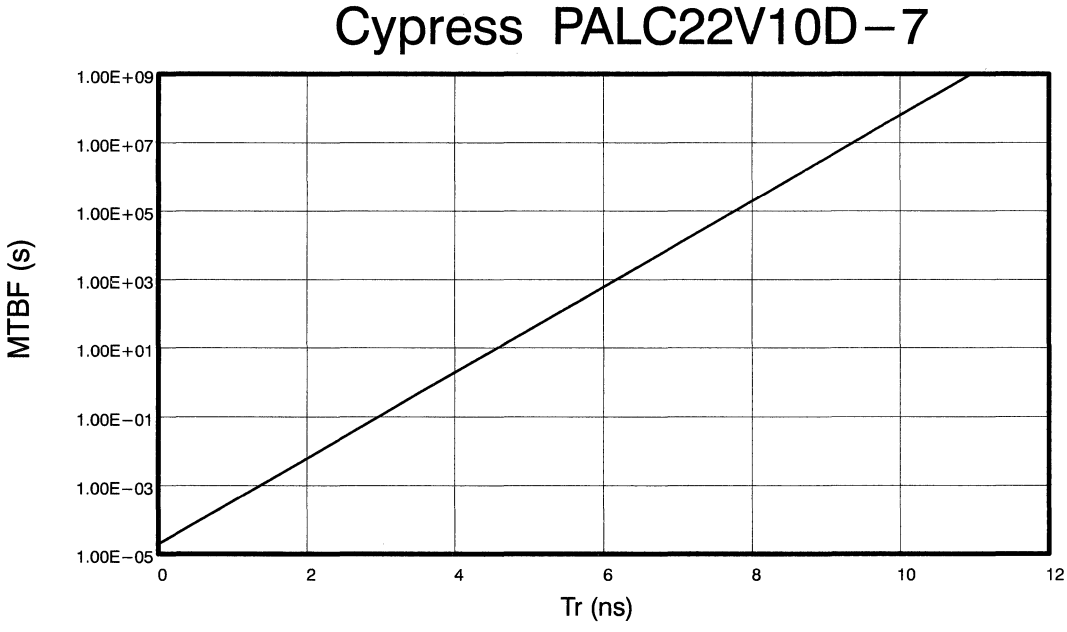
Appendix A. Metastability Graphs of Cypress Devices (continued)



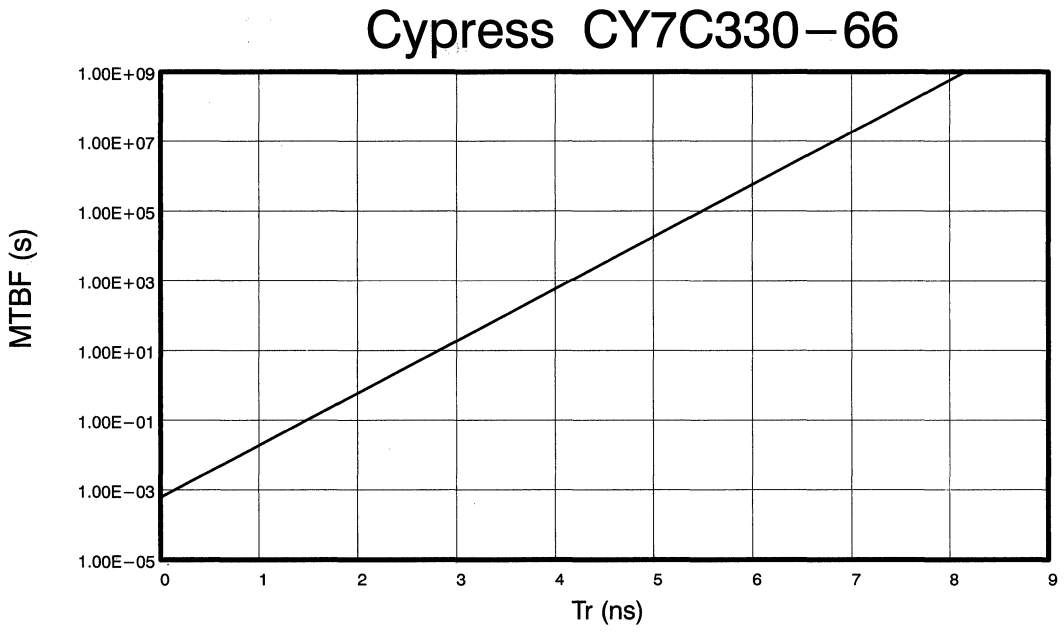


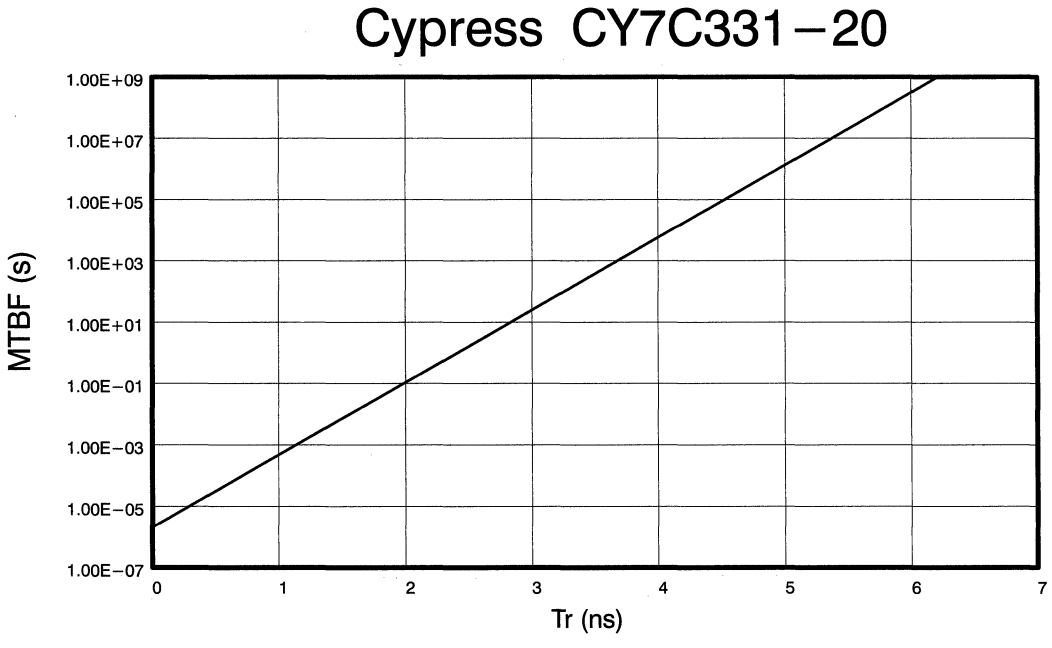
Appendix A. Metastability Graphs of Cypress Devices (continued)



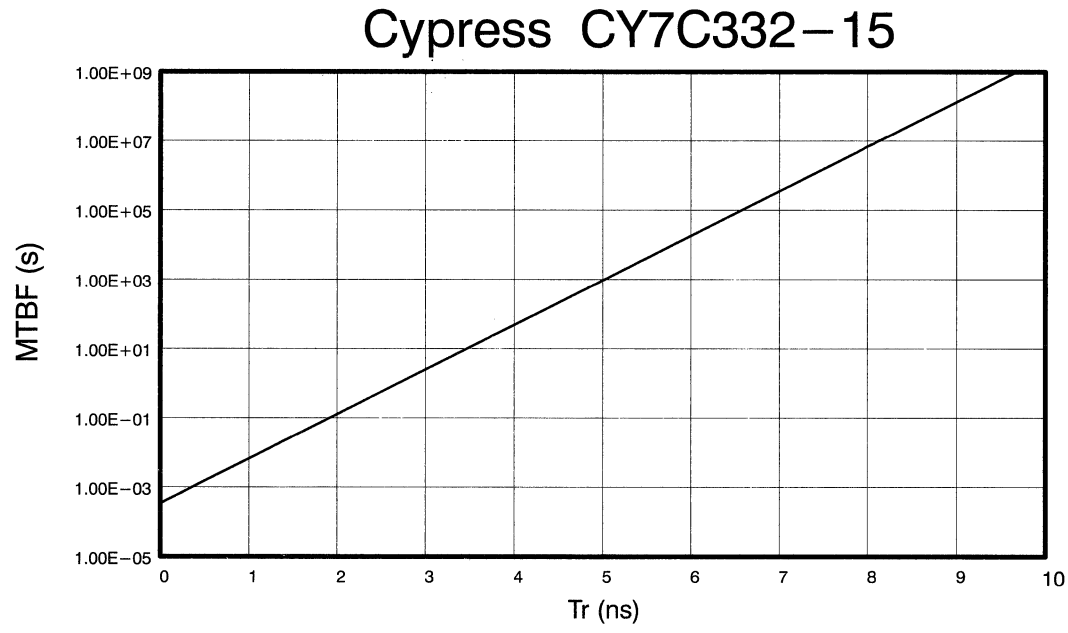


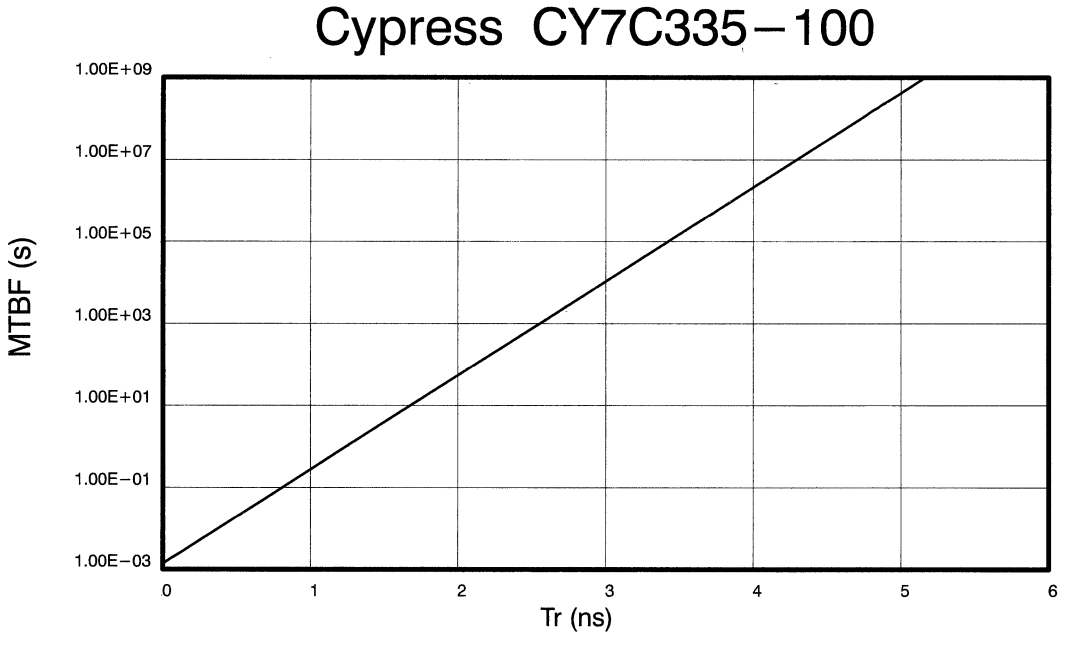
Appendix A. Metastability Graphs of Cypress Devices (continued)



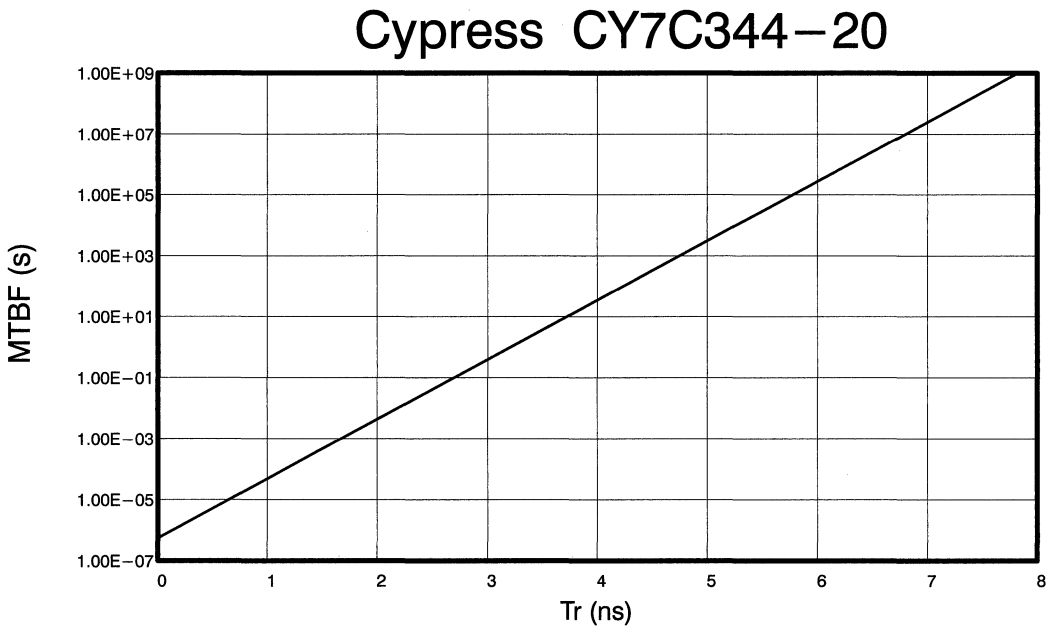


Appendix A. Metastability Graphs of Cypress Devices (continued)





Appendix A. Metastability Graphs of Cypress Devices (continued)



Designing with the CY7C335 and *Warp2*™ VHDL Compiler

This application note provides an overview of the CY7C335 Universal Synchronous EPLD architecture and *Warp2*™ VHDL Compiler for PLDs. Example designs demonstrate how the *Warp2* VHDL compiler takes advantage of the rich architectural features of the CY7C335.

The CY7C335 is a synchronous EPLD optimized for high-performance state machines and other clocked systems that operate at speeds of up to 100 MHz. The CY7C335 uses Cypress's low-power, 0.8-micron CMOS UV erasable technology and is packaged in 28-pin, 300-mil dual in-line and LCC/PLCC packages.

The CY7C335 builds on the popularity of the high-speed CMOS PALC22V10 and exceeds the capability of the 26V12 and 26CV12. The CY7C335 offers significantly higher density solutions and can replace as many as four 22V10s. It has 258 variable product terms for 16 state registers (ranging from 9 to 19 product terms per macrocell), macrocells that can be configured as JK-, RS-, T-, or D-type, bidirectional pins, bypassable input registers, three clocks, and a product term output enable for each macrocell.

In addition to supporting the features of the CY7C335, the *Warp2* VHDL compiler enables the designer to create designs, using any combination of high-level behavioral descriptions, Boolean equations, state tables, or RTL structures, that can easily be retargetted to any Cypress PLD.

Warp2 is a state-of-the-art VHDL compiler that facilitates device-independent designs by synthesizing for a powerful subset of IEEE1076. Optimization and reduction algorithms automatically select T- or D-type flip-flops and perform automatic state and pin assignment. *Warp2* includes a graphical user interface (which runs under Windows™ for the PC, and OpenLook™ or Motif™ for the Sun) and comes complete with a functional simulator for graphical waveform simulation.

Overview of the CY7C335

Figure 1 is the block diagram of the CY7C335. Three separate clock signals—two input and two output clocks (one shared)—can be used on pins 1, 2, and 3. Alternatively, pins 2 and 3 can be used as two of twelve inputs that may be registered or fed directly to the programmable AND array. Pin 14 can be used as an input or as a common output enable for each I/O pin. Outputs can also be enabled by product terms. The device features center ground and supply pins that reduce ground bounce due to parasitic effects, particularly lead inductance.

Figure 2 illustrates the input macrocell. Each D-type input register can use either ICLK1 or ICLK2. Alternatively, the input register bypass multiplexer can be programmed to allow the signal to feed directly to the array as combinatorial input.

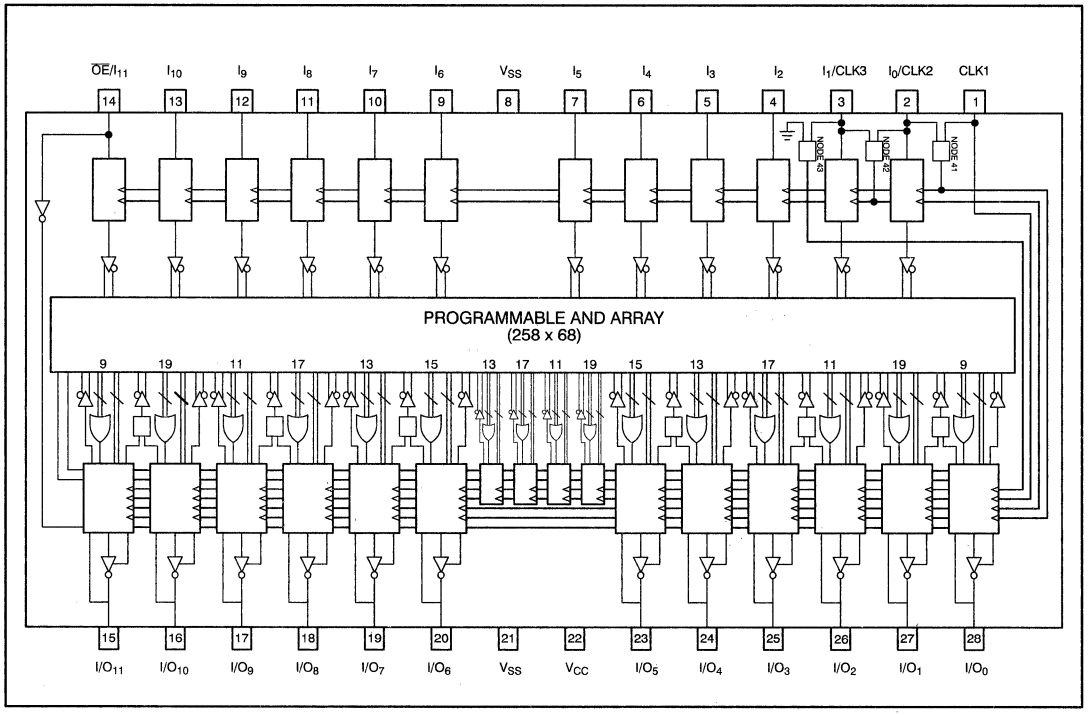


Figure 1. CY7C335 Block Diagram

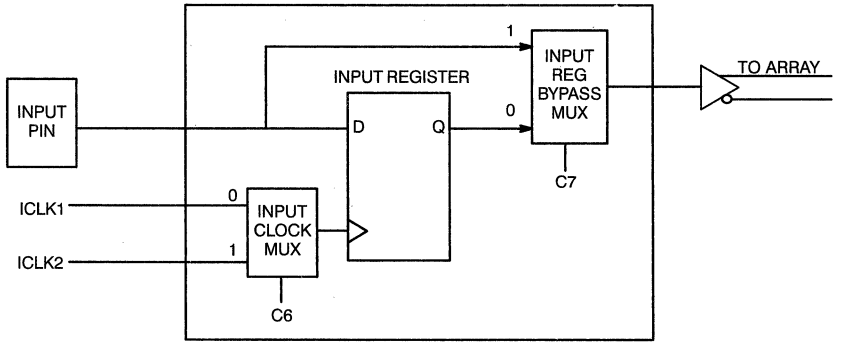


Figure 2. CY7C335 Input Macrocell

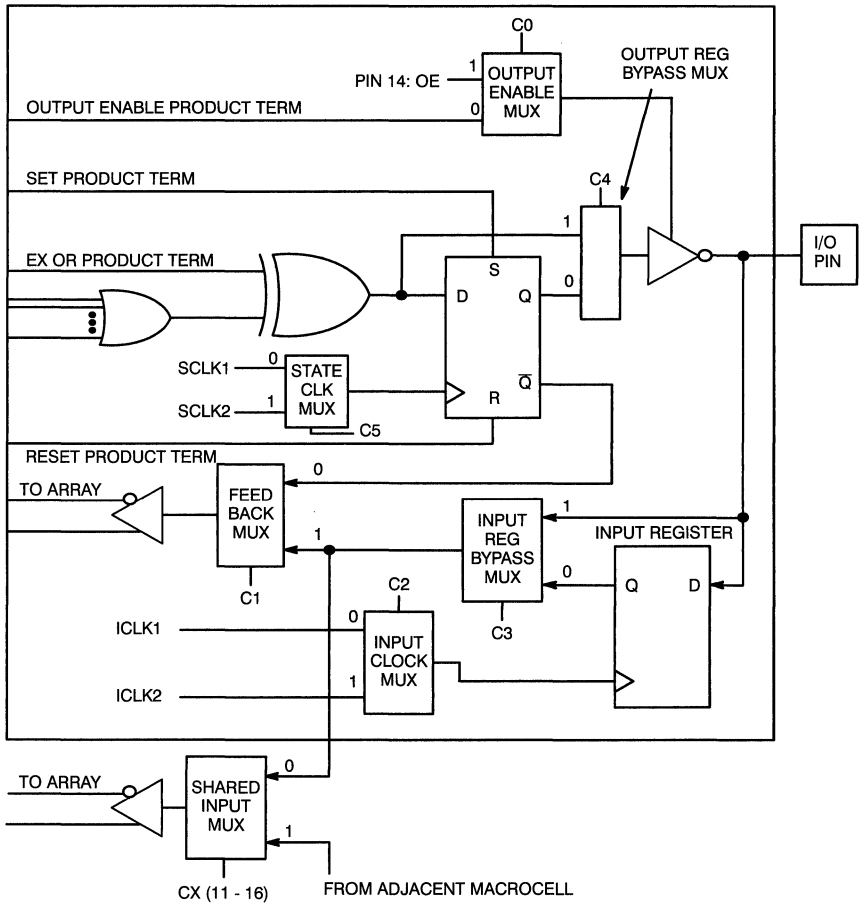


Figure 3. CY7C335 Input/Output Macrocell

Twelve configurable I/O macrocells enable JK-, RS-, T-, or D-type state registers to optimize for minimal product terms. *Figure 3* illustrates the I/O macrocell, which includes the following features: (1) registered or combinatorial output; (2) global (by pin 14) or product term output enable; (3) global, synchronous, product-term set and reset; (4) three clocks—two can be used as input clocks and two can be used as output clocks (with one shared); (5) input/output flexibility (the cell can be configured as input only, output only, or a dedicated input with a buried register by using the shared input multiplexer and thereby maximizing cell resource utilization).

In addition to the input and I/O macrocells, the CY7C335 features four hidden macrocells, one of which is shown in *Figure 4*. Buried registers are highly useful for state machines, internal counters, or other applications that need registers that are not also used as outputs.

The clocking scheme is shown in *Figure 5*. The CY7C335 can utilize three separate clocks. Two clocks are inputs to each of the input clock multiplexers and state clock multiplexers. If two clocks are used on both the input and the state registers, then one of the clocks is shared, because a total of

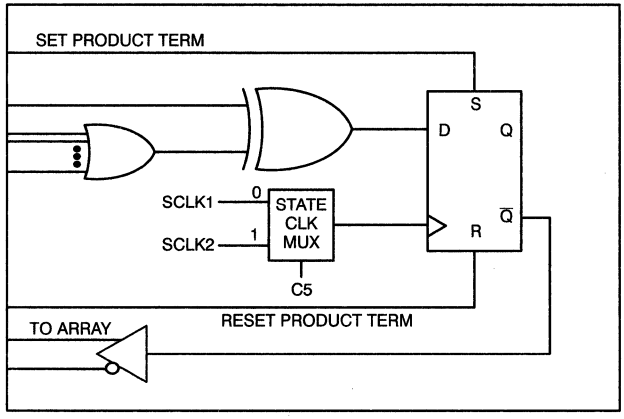


Figure 4. CY7C335 Hidden Macrocell

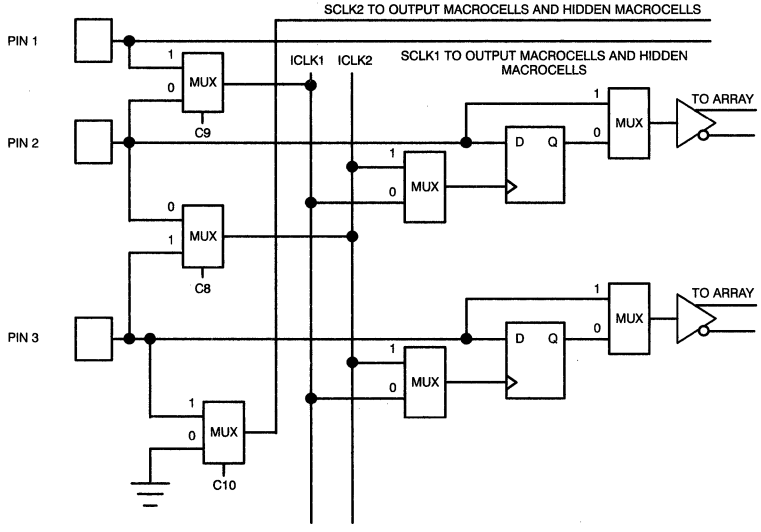


Figure 5. CY7C335 Input Clocking Scheme

three clocks are supported. Pin 1 is a dedicated state clock pin, designated SCLK1 (state clock). Pins 2 and 3 may be used as either inputs or clocks, as shown in *Figure 5*.

Overview of *Warp2*

Warp2 is a state-of-the-art VHDL compiler for designing with Cypress PLDs and PROMs. *Warp2* accepts VHDL designs, synthesizes and optimizes the entered design, and outputs a JEDEC file for the CY7C335. *Warp2* also provides a graphical wave-

form simulator for functional simulation. *Figure 6* illustrates the *Warp2* design flow.

VHDL Compiler

As an open, non-proprietary, IEEE1076 compliant language that is the standard for behavioral design entry and simulation, VHDL allows designers to easily describe complex hardware systems.

Warp2's VHDL enables designers to describe device-independent designs at different levels of abstraction, including behavioral descriptions, Boolean equations, state tables, and structural descriptions. In addition, VHDL and *Warp2* support hierarchical designs, allowing either a "top-down" or "bottom-up" approach to design.

Design Examples

The following design examples demonstrate how to use *Warp2* and VHDL to take advantage of the CY7C335 architectural features. The purpose is to show some VHDL constructs that are particularly useful for the CY7C335 architecture as well as point out designs that are well suited for the device. Further information on VHDL constructs may be found in the *Warp2 Reference Manual* or one of several texts available on the language. For each of the examples, the complete VHDL source code and an excerpt of the report file may be found in the appendices.

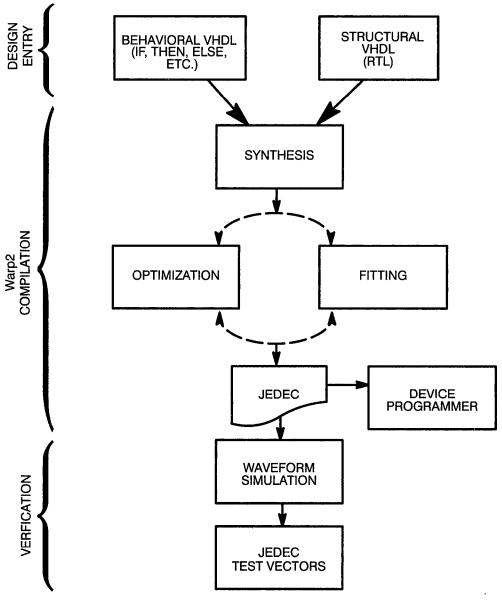


Figure 6. *Warp2* Design Flow

Pipelined Buffer

This example demonstrates how to use VHDL code to implement a pipelined buffer (see *Figure 7*) with multiple clocks and output enables. The CY7C335 is well suited for pipelined applications because it has input registers in both the input macrocells as well as the I/O macrocells.

The complete VHDL source code is printed in Appendix A of this note. The pipeline architecture is reprinted in *Figure 8*.

The pipeline is implemented in three processes. The first process registers (using CLK1 on the input registers) the upper four bits of the input signal I. The second process registers the lower four bits, using CLK2. The signal INTMP represents the q output of these registers. The third process registers the signal INTMP, with OUTTMP being the q output of these registers. This signal reaches the I/O pins if output is enabled, as explained below.

Below the three processes is a generation scheme which is used to instantiate eight *triout* components

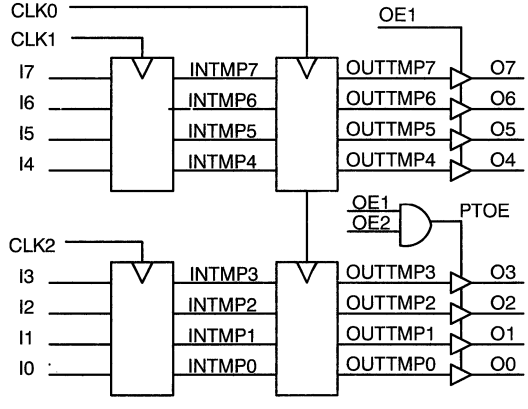


Figure 7. Pipelined Buffer Block Diagram

```
use work.rtlpkg.all;
architecture archpipe of pipe is
signal intmp, outtmp: bit_vector(7 downto 0);
signal ptoe: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      intmp(7 downto 4) <= i(7 downto 4);
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      intmp(3 downto 0) <= i(3 downto 0);
  end process;

proc3: process
  begin
    wait until clk0 = '1';
      outtmp <= intmp;
  end process;

  ptoe <= oe1 AND oe2;

g1:  for j in 7 downto 0 generate
  g2:  if j > 3 generate
       t1x: triout port map(outtmp(j), oe1, o(j));
    end generate;
  g3:   if j < 4 generate
       t2x: triout port map(outtmp(j), ptoe, o(j));
    end generate;
  end generate;

end archpipe;
```

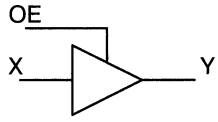
Figure 8. Pipeline Architecture

(see *Figure 9*). The *triout* components are used to implement an output enable. The upper four bits of the output are enabled by OE1 (which is assigned to pin 14 by *Warp*) and the lower four bits use a product term output enable, PTOE.

The complete VHDL source code for this example is in Appendix A. A report file excerpt, showing resource utilization, is shown in Appendix B. This excerpt shows that 8 of 12 I/O macrocells were utilized. However, not all resources (the input registers, for example) in those macrocells were utilized.

Comparator with Registered Inputs

In high-speed systems, such as microprocessor local buses that operate at 40, 50, or 66 MHz, data or addresses must be captured from the bus (when qualified with a strobe) with set-up times of 3 to 5 ns. Few logic functions can be implemented in this time, and




```
component triout
 port(
    x: in bit; --input to buffer
    oe: in bit; -- output enable
    y: out bit; -- output
 );
end component;
```

Figure 9. Triout Component

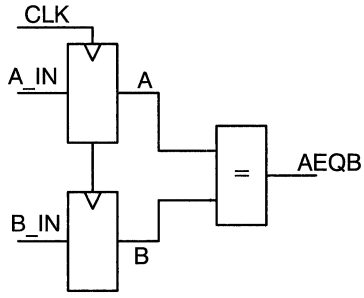


Figure 10. Comparator

for this reason data or addresses are captured and then processed in pipeline fashion. The CY7C335, with its input registers, is well suited for such high-speed systems.

In this simple, register-intensive example, all 18 inputs are registered and the output is combinatorial (*Figure 10*). As noted in Appendix D, this design leaves much of the CY7C335's resources free for additional logic. The 22V10, however, would be unable to fit a 5-bit comparator with registered inputs. Ten macrocells would be consumed when registering the inputs, leaving no macrocells for the AEQB combinatorial output. The 22V10 fares poorly in such pipelined systems because it does not have input registers and must therefore waste output macrocell resources.

The VHDL source code can be found in its entirety in Appendix C. The architecture is reprinted below.

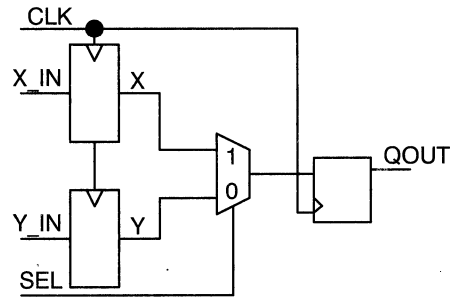


Figure 11. Multiplexer

```
architecture archcomp of comp is
signal a, b: bit_vector(0 to 8);
begin
proc1: process begin
 wait until clk = '1';
   a <= a_in;
   b <= b_in;
 end process;

 aeqb <= '1' when a=b else '0';
end archcomp;
```

The process is used to register the inputs on the rising edge of CLK1. The equation for AEQB is placed outside of the process because it is a combinatorial output.

Multiplexer with Registered Inputs and Outputs

Registered multiplexers and demultiplexers demand a large number of inputs and outputs. This example (see *Figure 11*) takes advantage of the CY7C335 input and output registers, two groups of six-bit-wide signals are captured via the input registers and signal SEL selects one of the groups, which is then registered on the output. The complete VHDL source code can be found in Appendix E. The architecture is reprinted below.

```
architecture archmux of mux is
signal x, y: bit_vector(5 downto
0);
begin
proc1: process begin
 wait until clk = '1';
   x <= xin;
   y <= yin;
```

```
    if sel = '1' then
      qout <= x;
    elsif sel = '0' then
      qout <= y;
    end if;
  end process;
end archmux;
```

On the rising edge of CLK1, the inputs are registered while the outputs are propagated. Thus, data on the inputs is not propagated to the outputs until the second rising edge.

Decoder

Faster microprocessors require decoders to operate at higher frequencies. Many high-density PLDs and FPGAs cannot meet speed requirements, leaving designers to opt for ASIC-based solutions which can be time consuming and expensive. The CY7C335 is another option.

Consider a 16-bit address that requires decoding to address system memory elements (SRAM, PROM, EEPROM and "shadow" RAM) and two peripheral ports. At times other than boot-up, the microprocessor fetches instructions from shadow RAM that is loaded from PROM during boot-up. *Figure 12* shows the VHDL architecture that decodes the memory map shown in *Figure 13*. Appendix H shows that the CY7C335's resources easily handle this application while operating at speeds to 100 MHz.

Up/Down Counter with Upper and Lower Limits

This example demonstrates how to use VHDL code to implement the up/down counter shown in *Figure 14*. The CY7C335 is particularly well suited for this design because it supports three clocks and has flexible I/O. This design requires the following resources: three clocks (two inputs and one state), eight input registers for the lower limit, eight input registers for the upper limit, one input each for the preset HIGH, preset LOW, reset, and output enable signals, eight state registers for the counter, one state register each for the comparators, and one state register for the counter direction signal.

A total of 20 inputs and 8 outputs are required; consequently, this design utilizes bidirectional signals.

The counter output is three-stated to load six bits of the upper limit into input registers of I/O macrocells. For example, the least-significant counter bit is stored in a state register and the least-significant upper-limit bit is stored in the input register of the same macrocell. The least-significant upper-limit bit feeds into the array via the shared input multiplexer. (The shared-input multiplexers are placed between adjacent I/O macrocells, and allow for input when the macrocell register is buried.) The CY7C335 provides six of these multiplexers. The two most significant bits of the upper limit are passed into the array through an I/O pin configured as a dedicated input. The two most significant bits of the upper limit and counter may be externally tied together so the design can be bidirectional.

The up/down counter counts between limits stored in the input registers. The lower-limit (LL) is loaded into the registers on the rising edge of CLK1 while the upper limit is loaded on the rising edge of CLK2. On CLK0, if preH is asserted, then the upper limit is loaded into the counter, and if preL is asserted, then the lower limit is loaded into the counter.

The 22V10 would not suffice for this design. Although the 22V10 has been an attractive choice of devices to implement counters and state machines, it suffers a limitation in addition to its poor handling of pipelined systems: it does not have any buried registers.

In counters and encoded state machines, registers often need not be apparent to the outside, meaning the registers can be buried within the device. In the 22V10, all macrocells are connected to I/O pins. Thus, even when a macrocell register is being used in a buried sense, the I/O pin is committed, thereby preventing the pin from being used as an additional input to the device.

In addition to overcoming the 22V10's shortcoming with pipelined systems by having input registers in both the input and I/O macrocells, the CY7C335 provides a solution to the 22V10's density problems with regards to counters and state machines by providing four buried registers. Additionally, pairs of macrocells have a shared input multiplexer that allows up to six additional inputs, even when all twelve

I/O macrocells have their registered outputs feeding back into the AND array.

The VHDL source code for this example is in Appendix I of this note, and the architecture is reprinted in *Figure 15*.

The up/down counter is implemented in three processes, a generation scheme, and two concurrent statements. In the first process, the lower limit is registered on the rising edge of CLK1. The signal LOWER registers the input signal LL. The second process registers the upper limit on the rising edge of CLK2. The third process implements (1) the up/down counter with reset, preset LOW, and preset HIGH, (2) two comparators, and (3) the direction signal (DIR) that indicates to count up (logic 1) or down (logic 0). The comparators and the direction signal are clocked by CLK0, forcing the counter to change direction from up to down or vice-versa two clock cycles after the count matches one of the limits. For this reason, the upper limit should be loaded with a value two less than the greatest desired count, and the lower limit should be loaded with a value two greater than the least desired count.

The generation scheme below the three processes is a means to instantiate 6 *bufoe* components (see *Figure 16*) and two *triout* components. The *bufoe* com-

```
use work.bv_math.all;
architecture behav of decode is
signal address: bit_vector(15 downto 0);
begin
  address <= a & "000";

proc1: process begin
  wait until clk = '1';
    promsel <= '0';
    shadowsel <= '0';
    periph1 <= '0';
    periph2 <= '0';
    sramsel <= '0';
    eesel <= '0';
    if valid = '1' then
      if address >= x"0000" and address < x"4000" then
        if bootover = '0' then
          promsel <= '1';
        else
          shadowsel <= '1';
        end if;
      elsif address >= x"4000" and address < x"4008" then
        periph1 <= '1';
      elsif address >= x"4008" and address < x"4010" then
        periph2 <= '1';
        elsif address >= x"8000" and address < x"C000" then
        sramsel <= '1';
      else address >= x"C000" then
        eesel <= '1';
      end if;
    end if;
  end process;
end behav;
```

Figure 12. VHDL Architecture

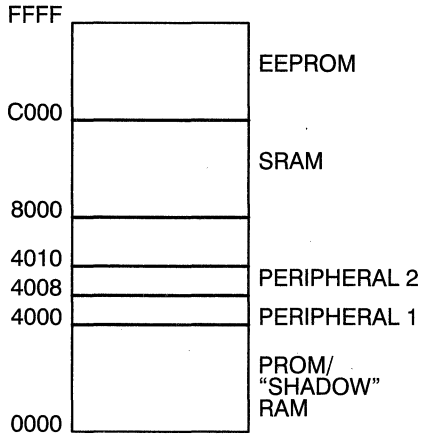


Figure 13. Decoder Memory Map

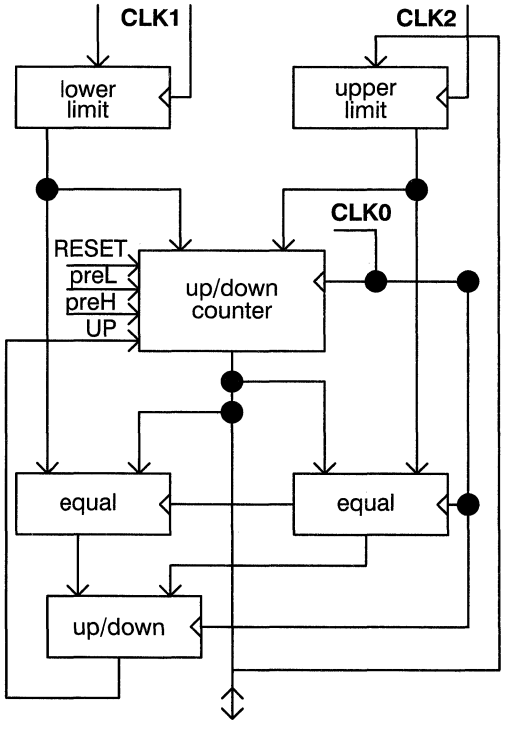


Figure 14. Up/Down Counter

ponents are used to implement the output enable and provide a feedback path for the upper limit. The CY7C335 has six shared input multiplexers that allow six bits of the signal count to utilize the state registers while enabling six bits of the upper limit to be loaded into the input register associated with the same macrocell. The remaining two bits of count will be placed in I/O macrocells in which the input registers are not used, and the two bits of the upper limit will be in two I/O macrocells configured as inputs. To enable bidirectional operation, the input and output pins for the associated upper limit and count bits can be connected externally. This is the reason for instantiating two triout components on the most significant two bits of the count.

Serial Decoder

The CY7C335's state registers and abundant product terms make it a good choice in which to implement state machines. The following VHDL code uses a state machine to implement a serial decoder that searches for a synchronization word within serially transmitted data. The sync word is the byte 11101000 and is expected to be repeated every 16 bytes. When the sync word is found, MATCH is asserted. When the sync word is found separated by 15 bytes three consecutive times, LOCK is asserted. The state diagram for this example is shown in *Figure 17.*

The architecture of this design is printed in *Figure 18* and the complete VHDL code is in Appendix K. The resources that this design uses (Appendix L) show that there is room for more logic within the device. For instance, the comparator with registered inputs described earlier could fit in the device along with this design.

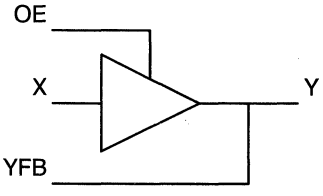
The first process within the architecture defines the state transitions. The second process is one that is synchronized by the clock. The output MATCH is determined by the present inputs and the currents state. This implements a Mealy machine. The counter process counts the number of bits after a match, and the synchronizer process checks to see if a match occurs 15 bytes after the previous one. If a match is separated by 15 bytes for three consecutive times, then on the fourth consecutive match separated by 15 bytes, LOCK is asserted.

```
use work.bv_math.all;
use work.rtlpkg.all;

architecture archupdown of updown is
signal lower, upper, ul, count: bit_vector(0 to 7);
signal cequ, ceql, dir: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      lower <= ll;
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      upper <= ul;
  end process;

proc3: process
  begin
    wait until clk0 = '1';
    -- implement counter
    if reset = '1' then
      count <= x"00";
    elsif preL = '1' then
      count <= lower;
    elsif preH = '1' then
      count <= upper;
    elsif (dir = '1') then
      count <= inc_bv(count);
    else
      count <= dec_bv(count);
    end if;     -- implement comparators & direction signal
    if count = upper then
      cequ <= '1';
    else
      cequ <= '0';
    end if;
    if count = lower then
      ceql <= '1';
    else
      ceql <= '0';
    end if;
```

Figure 15. Architecture

```
      if ceql = '1' then
        dir <= '1';
      elsif cequ = '1' then
        dir <= '0';
      else
        dir <= dir;
      end if;
    end process;

  g1:  for i in 0 to 7 generate
    bidir: if i < 6 generate
      bx:  bufoe port map(count(i), outen, countio(i), ul(i));
      end generate;
    trist: if i > 5 generate
      tx:  triout port map(count(i), outen, countio(i));
      end generate;
    end generate;

    ul(6) <= ul6;
    ul(7) <= ul7;

  end archupdown;
```

Figure 15. Architecture (continued)




```
component bufoe
 port(
    x: in bit; --input to buffer
    oe: in bit; --output enable
    y: inout x01z; --x01z output
    yfb: out bit; -- feedback
    );
end component;
```

Figure 16. bufoe Component

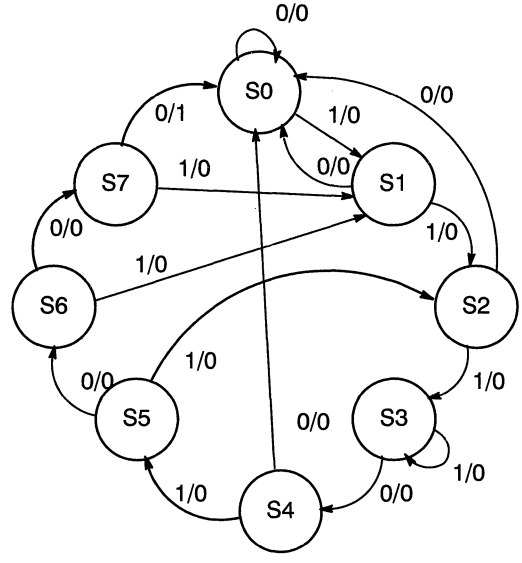


Figure 17. State Diagram

```
use work.int_math.all;
use work.bv_math.all;
architecture archserial of serial is
type states is (state0, state1, state2, state3, state4, state5, state6,
     state7);
signal state, nextstate: states;
signal match_cnt: bit_vector(1 downto 0);
signal bit_cnt: bit_vector(6 downto 0);

begin
fsm:
process begin
 match <= '0';
 case state is
   when state0 =>
     if data = '1' and (lock = '0' or bit_cnt = "1111000") then
       nextstate <= state1;
     else
       nextstate <= state0;
     end if;
   when state1 =>
     if data = '1' then
       nextstate <= state2;
     else
       nextstate <= state0;
     end if;
when state2 =>
     if data = '1' then
       nextstate <= state3;
     else
       nextstate <= state0;
     end if;
   when state3 =>
     if data = '0' then
       nextstate <= state4;
     else
       nextstate <= state3;
     end if;
   when state4 =>
     if data = '1' then
       nextstate <= state5;
     else
       nextstate <= state0;
     end if;
   when state5 =>
     if data = '0' then
       nextstate <= state6;
     else
       nextstate <= state2;
     end if;
```

Figure 18.

```
    when state6 =>
      if data = '0' then
        nextstate <= state7;
      else
        nextstate <= state1;
      end if;
    when state7 =>
      if data = '0' then
        nextstate <= state0;
        match <= '1';
      else
        nextstate <= state1;
      end if;
--No "when others" needed since CASE is completely defined.
 end case;
end process;

mealy:
process begin
 wait until clk = '1';
   state <= nextstate;
end process;

counter:
process begin
 wait until clk = '1';
 if match = '1' then
   bit_cnt <= "0000000";
 else
   bit_cnt <= inc_bv(bit_cnt);
 end if;
end process;

synchronizer:
process begin
 wait until clk = '1';
 if bit_cnt = "1111111" then
   if match = '1' then
     if match_cnt = "11" then
       lock <= '1';
     else
       match_cnt <= inc_bv(match_cnt);
     end if;
   else
     match_cnt <= "00";
     lock <= '0';
   end if;
 end if;
end process;

end archserial;
```

Figure 18. (continued)

Appendix A. *Warp2* VHDL Source Code for Pipelined Buffer

```
entity pipe is
  port( clk0, clk1, clk2: in bit;
    oe1, oe2: in bit;
    i: in bit_vector(7 downto 0);
    o: out x01z_vector(7 downto 0));
end pipe;

use work.rtlpkg.all;
architecture archpipe of pipe is
signal intmp, outtmp: bit_vector(7 downto 0);
signal ptoe: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      intmp(7 downto 4) <= i(7 downto 4);
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      intmp(3 downto 0) <= i(3 downto 0);
  end process;

proc3: process
  begin
    wait until clk0 = '1';
      outtmp <= intmp;
  end process;

  ptoe <= oe1 AND oe2;

g1: for j in 7 downto 0 generate
  g2: if j > 3 generate
      t1x: triout port map(outtmp(j), oe1, o(j));
    end generate;
  g3:  if j < 4 generate
      t2x: triout port map(outtmp(j), ptoe, o(j));
    end generate;
  end generate;

end archpipe;
```

Appendix B. *Warp2* Report File Excerpt for Pipelined Buffer

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	8	12
Buried Macrocells	0	4
	21 /	29 = 72 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	o_0_	1	9
16	Unused	0	19
17	o_2_	1	11
18	Unused	0	17
19	o_4_	1	13
20	o_7_	1	15
23	o_6_	1	15
24	o_5_	1	13
25	Unused	0	17
26	o_3_	1	11
27	Unused	0	19
28	o_1_	1	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		8 / 230	= 3 %

Appendix C. *Warp2* Source Code for Comparator

```
entity comp is port (
  clk: in bit;
  a_in, b_in: bit_vector(0 to 8);
  aeqb: out bit);
end comp;

architecture archcomp of comp is
signal a, b: bit_vector(0 to 8);
begin
proc1: process begin
  wait until clk = '1';
    a <= a_in;
    b <= b_in;
  end process;

  aeqb <= '1' when a=b else '0';

end archcomp;
```

Appendix D. *Warp2* Report File Excerpt for Comparator

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	7	12
Buried Macrocells	0	4
	20 /	29 = 68 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	Used As Input	0	9
16	Used As Input	0	19
17	Used As Input	0	11
18	Used As Input	0	17
19	Used As Input	0	13
20	Used As Input	0	15
23	Unused	0	15
24	Unused	0	13
25	Unused	0	17
26	Unused	0	11
27	aeqb	18	19
28	Unused	0	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		18 / 230	= 7 %

Appendix E. *Warp2* Source Code for Multiplexer

```
entity mux is port(
  clk, sel: in bit;
  xin, yin: in bit_vector(5 downto 0);
  qout: out bit_vector(5 downto 0));
end mux;

architecture archmux of mux is
signal x, y: bit_vector(5 downto 0);
begin
proc1: process begin
  wait until clk = '1';
    x <= xin;
    y <= yin;
    if sel = '1' then
      qout <= x;
    elsif sel = '0' then
      qout <= y;
    end if;
  end process;
end archmux;
```

Appendix F. *Warp2* Report File Excerpt for Multiplexer

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	7	12
Buried Macrocells	0	4
	20 /	29 = 68 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	qout_0_	2	9
16	Used As Input	0	19
17	qout_2_	2	11
18	Unused	0	17
19	qout_4_	2	13
20	Unused	0	15
23	Unused	0	15
24	qout_5_	2	13
25	Unused	0	17
26	qout_3_	2	11
27	Unused	0	19
28	qout_1_	2	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		12 / 230	= 5 %

Appendix G. *Warp2* VHDL Source Code for Decoder

```
entity decode is port(
  a: in bit_vector(15 downto 3);
  rdwritebar, valid, bootover, clk: in bit;
  sramsel, promsel, eesel, shadowsel, periph1, periph2: out bit);
end decode;

use work.bv_math.all;
architecture behav of decode is
signal address: bit_vector(15 downto 0);
begin
  address <= a & "000";

proc1: process begin
  wait until clk = '1';
    promsel <= '0';
    shadowsel <= '0';
    periph1 <= '0';
    periph2 <= '0';
    sramsel <= '0';
    eesel <= '0';
    if valid = '1' then
      if address >= x"0000" and address < x"4000" then
        if bootover = '0' then
          promsel <= '1';
        else
          shadowsel <= '1';
        end if;
      elsif address >= x"4000" and address < x"4008" then
        periph1 <= '1';
      elsif address >= x"4008" and address < x"4010" then
        periph2 <= '1';
        elsif address >= x"8000" and address < x"C000" then
        sramsel <= '1';
      else address >= x"C000" then
        eesel <= '1';
      end if;
    end if;
  end process;
end behav;
```

Appendix H. *Warp2* Report File Excerpt for Decoder

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	9	12
Buried Macrocells	0	4
	22 /	29 = 75 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	eesel	1	9
16	Used As Input	0	19
17	periph2	1	11
18	Used As Input	0	17
19	shadowsel	1	13
20	Used As Input	0	15
23	Unused	0	15
24	promsel	1	13
25	Unused	0	17
26	periph1	1	11
27	Unused	0	19
28	sramsel	1	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		6 / 230	= 2 %

Appendix I. *Warp2* Source Code for UpDown

```
entity updown is
  port( clk0, clk1, clk2: in bit;
    outen, preL, preH, reset: in bit;
    ll: in bit_vector(0 to 7);
    ul6, ul7: in bit;
    countio: inout x01z_vector(0 to 7));
end updown;

use work.bv_math.all;
use work.rtlpkg.all;

architecture archupdown of updown is
signal lower, upper, ul, count: bit_vector(0 to 7);
signal cequ, ceql, dir: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      lower <= ll;
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      upper <= ul;
  end process;

proc3: process
  begin
    wait until clk0 = '1';
    if reset = '1' then
      count <= x"00";
    elsif preL = '1' then
      count <= lower;
    elsif preH = '1' then
      count <= upper;
    elsif (dir = '1') then
      count <= inc_bv(count);
    else
      count <= dec_bv(count);
    end if;
  end process;

proc4: process
  begin
    wait until clk0 = '1';
    if count = upper then
      cequ <= '1';
```

Appendix I. *Warp2* Source Code for UpDown (continued)

```
    else
      cequ <= '0';
    end if;
    if count = lower then
      ceql <= '1';
    else
      ceql <= '0';
    end if;
    if ceql = '1' then
      dir <= '1';
    elsif cequ = '1' then
      dir <= '0';
    else
      dir <= dir;
    end if;
  end process;

g1: for i in 0 to 7 generate
  bidir: if i < 6 generate
    bx: bufoe port map(count(i), outen, countio(i), ul(i));
    end generate;
  trist: if i > 5 generate
    tx: triout port map(count(i), outen, countio(i));
    end generate;
  end generate;

  ul(6) <= ul6;
  ul(7) <= ul7;

end archupdown;
```

Appendix J. *Warp2* Report File Excerpt for UpDown

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	12	12
Buried Macrocells	3	4
	28 /	29 = 96 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	countio_2_	7	9
16	Used As Input	0	19
17	countio_0_	3	11
18	Used As Input	0	17
19	countio_6_	7	13
20	countio_4_	7	15
23	Used As Input	0	15
24	countio_5_	7	13
25	countio_3_	7	17
26	countio_7_	7	11
27	Used As Input	0	19
28	countio_1_	6	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	ceql_BEH_i27..	16	17
33	dir	2	11
34	cequ	16	19
		85 / 230	= 36 %

Appendix K. *Warp2* VHDL Source Code for Serial Decoder

```
entity serial is port(
  clk, reset, data: in bit;
  match: buffer bit;
  lock: buffer bit);
end serial;

use work.int_math.all;
use work.bv_math.all;
architecture archserial of serial is
type states is (state0, state1, state2, state3, state4, state5, state6,
    state7);
signal state, nextstate: states;
signal match_cnt: bit_vector(1 downto 0);
signal bit_cnt: bit_vector(6 downto 0);

begin
fsm:
process  begin
  match <= '0';
  case state is
    when state0 =>
      if data = '1' and (lock = '0' or bit_cnt = "1111000") then
        nextstate <= state1;
      else
        nextstate <= state0;
      end if;
    when state1 =>
      if data = '1' then
        nextstate <= state2;
      else
        nextstate <= state0;
      end if;
    when state2 =>
      if data = '1' then
        nextstate <= state3;
      else
        nextstate <= state0;
      end if;
    when state3 =>
      if data = '0' then
        nextstate <= state4;
      else
        nextstate <= state3;
      end if;
    when state4 =>
      if data = '1' then
        nextstate <= state5;
      else
```

Appendix K. *Warp2* VHDL Source Code for Serial Decoder (continued)

```
        nextstate <= state0;
      end if;
    when state5 =>
      if data = '0' then
        nextstate <= state6;
      else
        nextstate <= state2;
      end if;
    when state6 =>
      if data = '0' then
        nextstate <= state7;
      else
        nextstate <= state1;
      end if;
    when state7 =>
      if data = '0' then
        nextstate <= state0;
        match <= '1';
      else
        nextstate <= state1;
      end if;
  end case;
end process;

mealy:
process begin
  wait until clk = '1';
    state <= nextstate;
end process;

counter:
process begin
  wait until clk = '1';
  if match = '1' then
    bit_cnt <= "0000000";
  else
    bit_cnt <= inc_bv(bit_cnt);
  end if;
end process;

synchronizer:
process begin
  wait until clk = '1';
  if bit_cnt = "1111111" then
    if match = '1' then
      if match_cnt = "11" then
        lock <= '1';
```

Appendix K. *Warp2* VHDL Source Code for Serial Decoder (continued)

```
      else
        match_cnt <= inc_bv(match_cnt);
      end if;
    else
      match_cnt <= "00";
      lock <= '0';
    end if;
  end if;
end process;

end archserial;
```

Appendix L. *Warp2* Report File Excerpt for Serial Decoder

```
Information: Macrocell Utilization.
```

Description	Used	Max
Dedicated Inputs	0	9
Clock/Inputs	1	3
Enable/Inputs	0	1
I/O Macrocells	10	12
Buried Macrocells	4	4
	15 / 29	= 51 %

```
Information: Output Logic Product Term Utilization.
```

Node#	Output Signal Name	Used	Max
15	lock	9	9
16	Unused	0	19
17	data	5	11
18	bit_cnt_0_	1	17
19	serial_0_sta..	4	13
20	bit_cnt_2_	3	15
23	bit_cnt_1_	2	15
24	serial_0_sta..	4	13
25	match	1	17
26	bit_cnt_3_	4	11
27	Unused	0	19
28	bit_cnt_4_	5	9
29	Unused	0	1
30	Unused	0	1
31	match_cnt_0_	8	13
32	bit_cnt_6_	7	17
33	match_cnt_1_	9	11
34	bit_cnt_5_	6	19
		68 / 230	= 29 %

Getting Started Converting .ABL Files to VHDL

Introduction

This application note is intended to assist *Warp*™ users in converting designs written in DATA I/O's ABEL™ 7 hardware description language to IEEE 1076 VHDL. It contains several language cross reference tables and many helpful hints. It also includes two real-world designs that have been converted from MACH™ 210–ABEL descriptions to FLASH371–VHDL descriptions.

VHDL versus ABEL

VHDL is different from ABEL and virtually all other popular hardware description languages in one very significant way. It is an open language based on IEEE standard number 1076.

VHDL is different in other ways, too. VHDL is a high-level language. As such, it is much more powerful than ABEL. For instance, it supports hierarchical design entry, structural (low-level instantiation of components) and behavioral (IF-THEN-ELSE) design entry. VHDL supports process and concurrent process statements. It also supports various types of signals such as integer, real, character, bit, Boolean, physical unit, and any others that a user can define. It supports sequential and concurrent statements, variables, and signals. VHDL supports sub-programs, FOR loops, WHILE loops, arrays, concurrent procedure calls, and more.

Surprisingly, certain aspects of VHDL related to low-level behavioral logic description are very similar to ABEL. In fact, some key words and relational operators are identical or logically similar.

Conversion Preparation

Preparing to convert an ABEL (.ABL) file should include the following steps:

1. Locate and have at hand one good VHDL language reference book. See the *Warp* documentation for a bibliography.
2. Obtain copies of the *Warp* VHDL design examples titled Basic, Intermediate, and Advanced.
3. Locate two Cypress application notes, one titled "Designing State Machines with *Warp2*™ VHDL" and another titled "VHDL Techniques for Optimal Design Fitting." Both are contained in the *Cypress Applications Handbook* (1993).
4. Install the *Warp* VHDL compiler on your hardware platform.

Conversion Approach

There are many different ways to convert a given design. The same design may be expressed in a number of different ways, all yielding compiled designs with the same functionality. The general approach suggested for converting ABEL (.ABL) files to VHDL (.VHD) consists of five basic steps:

1. Analyze the design and determine:
   a. Which signals are registered and which are not. Group them into two categories.
   b. Which types of design entry the .ABL file includes: state machines, comparators, counters, decoders, multiplexers, adders, multipliers, shift registers, or state tables.
   c. Whether or not group (set) declarations are used.

d. Which signals are input, output, I/O, and/or active LOW.

2. Replace as many of the keywords and operators with the corresponding VHDL keywords and operators using your favorite SEARCH and REPLACE text editor and a backup copy of the .ABL or .DOC file.
3. Add the VHDL entity (black box inputs and outputs), architecture (description of the logic circuitry), and process (encapsulates a set of sequential-behavioral functions) statements.
4. Add the proper library USE statements to the file such as USE WORK.CYPRESS.ALL.
5. Iteratively compile the design revising incomplete or incorrectly converted syntax.

Some designs will be much easier to convert than others. The more regular the design the easier it will be to convert. The most efficient and highest level of conversion will be achieved by using the source (.ABL) file, the five steps above, and the cross reference information below.

The simpler approach is to use the .DOC file exclusively. Using the .DOC file works but does have one significant drawback. The .DOC file tends to be verbose. It is verbose because it describes the design at a low level. A converted ABEL (.DOC) design thus results in unnecessarily verbose VHDL. In other words, it results in inefficient code.

When converting using the .DOC file, place all of the registered signals into a process with a WAIT UNTIL CLOCK = '1' and place all of the combinatorial signals outside the process. This avoids the necessity of PROCESS sensitivity lists and IF-THEN-ELSE statements. The converted designs below and all of the *Warp* example designs attempt to describe functions behaviorally and at a higher level. For this reason no low-level design conversion examples are included.

Refer to the following sections and tables for helpful information when converting ABEL .ABL and .DOC descriptions.

Comments

Comments in ABEL are denoted by the quote symbol ("). Comments in VHDL are denoted by two consecutive dashes −−. For example:

ABEL	VHDL
"Inputs	−−Inputs
"Outputs	−−Outputs

VHDL-ABEL Special Constant Cross Reference

ABEL	VHDL	Description
.C.	requires user definition	Clocked input (0 −>1−> 0)
.D.	requires user definition	Clock down edge (1 −> 0)
.F.	requires user definition	Floating input or output
.K.	requires user definition	Clocked input (1−> 0−> 1)
.P.	requires user definition	Register preload
.SVn.	requires user definition	Super voltage ($2 \le n \le 9$)
.U.	requires user definition	Clock up edge (0 − > 1)
.X.	requires user definition	Don't care condition
.Z.	requires user definition	High impedance

In VHDL a constant is an object whose value may not be changed. The syntax for declaring a constant in VHDL is:

```
constant identifier_list : type [ :=expression];
```

An example of this is:

```
TYPE stvar is bit_vector (0 to 1);
constant State0 : stvar := "00";
```

This example declares a constant that is identified by the name State0, is of type stvar, which has been previously defined as a bit_vector subtype of length 2. This constant is given the value 00. Defining a constant of user-defined type called state variable (stvar) is useful when designing state machines in VHDL. See the State Machine section of this application note.

Special constants in ABEL are used for simulation vectors that are included in the source file (.ABL). *Warp* does not provide simulation support directly within the source file. So, the conversion recommendation for files containing simulation vectors is to delete or comment them out of the .VHD file. *Warp* provides simulation separately from the design file (.VHD). Simulation can take one of two forms. The first, functional waveform based design verification using NOVA. Second, full AC timing verification via VIEWSIM and VIEWTRACE. VIEWSIM and VIEWTRACE support both pattern files and waveforms. Both forms of *Warp* VHDL simulation exceed the capabilities of ABEL simulation.

VHDL-ABEL Dot Extension Cross Reference

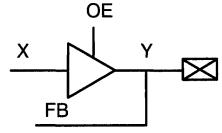
ABEL	VHDL	Function
.CLK	none	Clock input to flip-flop
.PIN	none	Pin feedback
.FB	none	Register feedback
.D	none	D-type flip-flop
.J	none	J input to JK flip-flop
.K	none	K input to JK flip-flop
.S	none	S input to SR flip-flop
.R	none	R input to SR flip-flop
.T	none	T-type flip-flop
.Q	none	Register feedback
.PR	none	Register preset
.RE	none	Register reset
.SP	none	Synchronous reg preset
.SR	none	Synchronous reg reset
.LE	none	Latch-enable input
.LH	none	Latch-enable input (HIGH)
.LD	none	Register load input
.CE	none	Clock enable input
.AP	none	Asynchronous preset
.AR	none	Asynchronous reset
.OE	none	Three-state output enable
.CLR	none	Synchronous clear
.ACLR	none	Asynchronous clear

VHDL-ABEL Dot Extension Cross Reference (continued)

ABEL	VHDL	Function
.SET	none	Synchronous set
.ASET	none	Asynchronous clear
.COM	none	Combinational feedback
.FC	none	Flip-flop mode control

Although VHDL is capable of supporting these constructs, it directly does not. Indirectly, through behavioral description, structural description, and an intelligent compiler, all of these constructs are supported. For example, *Warp* does provide predefined register transfer level (RTL) components (such as D- and T-type flip-flops). These RTL components can be structurally instantiated to model the ABEL extensions listed above. Specifically, to model the .OE ABEL extension, use an RTL component called *bufoe*. The syntax and port map (inputs and outputs) of a *bufoe* component is the following:

Label: BUFOE PORT MAP(X, OE, Y, FB)



X, OE, Y, and FB are sample signal names. The position each one occupies in the port map is the mechanism VHDL uses to correctly connect the signal names to the actual component in the architecture of the target device.

The behavioral equivalent of structurally instantiating a bidirectional buffer is called a behavioral three-state. Behavioral three-states are presently not supported, but will be in the future.

If the ABEL description equation is written in .T (T-type) flip-flop style, the recommended conversion method is to rewrite the equations as D-type (XOR the original equation with the flip-flop output signal name) and let *Warp* optimize the equation for either D- or T-type. See the real-world design conversion example in Appendix A called FLAGCTLR.

IS_TYPE Attribute Cross Reference

ABEL	VHDL	Description
'buffer'	none, may use RTL – buf	Macrocell has no inverter between reg and pin
'com'	none, may use RTL – buf	Signal is combinatorial
'invert'	none, NOT RHS of equation	Macrocell has an inverter between reg and pin
'neg'	none, NOT signal in equation	Complement Sum of Products
'pos'	none	Do not Complement Sum of Products
'reg'	none, place sig in process	Generic flip-flop
'reg_D'	none, may use RTL – dff	D-type flip-flop
'reg_T'	none, may use RTL – tff	T-type flip-flop
'reg_SR'	none, may use RTL – srff	SR-type flip-flop
'reg_JK'	none, may use RTL – jkff	JK-type flip-flop
'reg_G'	none	D-flip-flop w/gated clock
'xor'	none, may use RTL – xbuf	Target architecture has XOR

Operator Cross Reference

ABEL	Order of Precedence	VHDL	Order of Precedence	Operation
!	1	NOT	Context dependent	NOT (invert)
&	2	AND	1	AND
*	2	*	5	Multiplication
/	2	/	5	Division
%	2	mod	5	Modulus
<<	2	none		Shift left
>>	2	none		Shift right
+	3	+	3	Arithmetic addition
−	3	−	3	Arithmetic subtr.
$	3	XOR	1	XOR
!$	3	NOT XOR		XNOR
#	3	OR	1	OR
==	4	=	2	Equal
!=	4	/=	2	Not equal
<	4	<	2	Less than
<=	4	<=	2	Less than or equal
>	4	>	2	Greater than
>=	4	>=	2	Greater or equal
none		NAND	1	NAND
none		NOR	1	NOR
none		&	3	Concatenation
none		rem	5	Remainder
none		abs	6	Absolute Value
none		**	6	Exponentiation
?		+	4	Sign
?		−	4	Sign
= or :=		<=		Signal assignment
none		:=		Variable assignment
=		<=		Comb. assignment
:=		<=		Reg. assignment
none		=>		Association

The only ABEL operators without a direct VHDL counterpart are the >> and <<, (shift right and shift left). To directly (structurally) describe logic that performs an N-bit shift left or right function see the *Warp* design example titled advanced SHIFTN.VHD. To emulate (behaviorally describe) a shift function in VHDL use bit_vectors or arrays and index the elements using LOOPs. Another technique is to use *2 and /2 (multiply by 2 and divide by 2).

Keyword (Statement) Cross Reference

ABEL Keyword	VHDL Equivalent
CASE	CASE
DECLARATIONS	Note 1
DEVICE	ATTRIBUTE PART_NAME IS ...
ELSE	ELSE
ENABLE (Obsolete)	none
ENDCASE	END CASE
ENDWITH	Note 2
EQUATIONS	Note 3
FLAG (Obsolete)	none
FUSES	Note 4
GOTO	EXIT – Note 5
IF	IF
IN (Obsolete)	none
ISTYPE	Note 6
LIBRARY	USE
MACRO	FUNCTION – Note 7
MODULE	FUNCTION – Note 8
NODE	SIGNAL
OPTIONS	Note 9
PIN	Note 10
PROPERTY	none
STATE	Note 11
STATE_DIAGRAM	Note 11
TEST_VECTORS	Note 12
THEN	THEN
TITLE	Note 13
TRACE	Note 14
TRUTH_TABLE	Note 15
WHEN	WHEN
WITH	Note 16
ASYNC_RESET	Note 17
SYNC_RESET	Note 17
STATE_REGISTER	Note 18
XOR_FACTORS	Note 19

VHDL does not use the term keyword. Analogous to ABEL's use of the term keyword, VHDL uses the terms statement, reserved word, and identifier.

Notes

1. There is not a DECLARATIONS keyword in VHDL. However, the DECLARATIONS keyword is analogous to declaring an ENTITY in VHDL. Within the ENTITY construct inputs, outputs, and I/Os are declared with appropriate mode and type. Mode loosely refers to the pin drive direction, which can be IN, OUT, or INOUT. Refer to your language reference book for a more formal definition of the terms mode, IN, OUT, and INOUT.
2. ENDWITH is part of the WITH-ENDWITH transition structure used with IF-THEN-ELSE or CASE keywords. In VHDL conditional transition is handled via an IF-THEN-ELSE or CASE statement within a PROCESS. The process statement may or may not use a sensitivity list and instead use a WAIT UNTIL (condition) statement. See the application note titled "Designing State Machines with *Warp2* VHDL."
3. Equations in VHDL are listed within an architecture statement.
4. VHDL and *Warp* do not provide predefined fuse-level program specification.
5. VHDL does not have a GOTO keyword (statement). It provides an EXIT keyword for stopping execution of loops entirely.
6. The IS_TYPE keyword (statement) defines attributes and/or characteristics of pins and nodes. VHDL provides these attributes through behavioral specification. Additionally, *Warp* provides a set of predefined attributes and VHDL provides a mechanism for declaring new attributes. See the attribute table below.
7. VHDL provides function call and return capability. MACRO is more of a low-level substitution technique such that, wherever the MACRO_id occurs, the text associated with that macro will be substituted.
8. The MODULE ... END statement(s) are source file requirements in ABEL. In VHDL the ENTITY-ARCHITECTURE pair are the basic source file requirements. Both the ENTITY and ARCHITECTURE constructs require an END statement.
9. OPTION is a string of processing options that affect the way in which the ABEL source file is processed by the language processor. The analogous control in VHDL is not done in the source file. It in fact is not part of the VHDL language. It is simply a menu of compiler options that are set when using *Warp* to synthesize the design.
10. PIN is used to declare input and output signals that must be available on a device I/O pin. The analogous PIN specification is implied in VHDL via the port map list in the ENTITY construct. All signal names listed in the entity port map are input, output, and I/Os of the entity.
11. See application note titled "Describing State Machines with *Warp*2 VHDL."
12. Test vectors are not directly supported by VHDL. However, both behavioral simulation and full AC timing simulation are available for design verification.
13. The TITLE statement is used to give an ABEL MODULE a title that will appear as a header in both the programmer load file (the JEDEC file) and the documentation file. When compiling a VHDL design using *Warp*, the filename of the VHDL (.VHD) design file is passed through to the programmer load file (.JED) as well as the documentation file (.RPT).
14. The TRACE statement controls the display features of ABEL's simulator. There is not a similar keyword in VHDL because simulation is separate from the source file description.
15. The TRUTH_TABLE keyword is used in ABEL to specify outputs as functions of different input combinations in a tabular form. VHDL does not directly provide a TRUTH_TABLE keyword. However,

in the common library (directory) included in *Warp*, there is a file called LIBSTATE.VHD that contains a FUNCTION called TTF. TTF is a predefined truth table function that can be used for both combinatorial truth tables and for state transition tables. See the application note titled "Describing State Machines in *Warp*2 VHDL" regarding use of the TTF function.

16. WITH is part of the WITH-ENDWITH transition structure used with IF-THEN-ELSE or CASE statements. In VHDL, conditional transition is handled via an IF-THEN-ELSE or CASE statement within a PROCESS. The process statement may use a sensitivity list or may include a WAIT UNTIL (clock = '1') statement.
17. ASYNC_RESET and SYNC_RESET statements are used in Symbolic State descriptions. They symbolically specify what state the machine should asynchronously or synchronously reset to, based upon a signal or an expression. In VHDL, asynchronous and synchronous resets are best handled from a behavioral perspective the Resets and Presets section of this note for more detail.
18. STATE_REGISTER is a mechanism whereby specific states of a machine can be identified symbolically. See the State Machine section of this note for more detail.
19. XOR_FACTORS is a keyword that is useful for factoring logic designs that target a device which features XOR gates. There is not an analoguos keyword in VHDL. HOwever, the functional aspect of this keyword is part of the *Warp* Compiler Option menu. For more details see the *Warp* Compiler Options Documentation.

Predefined Attributes Supported by *Warp*

Value Attributes	'Left 'Right 'High 'Low 'Length
Function Attributes (types)	'Pos 'Val 'Leftof 'Succ 'Rightof 'Pred
Function Attributes (objects)	'Left 'Right 'High 'Low 'Length
Function Attributes (signals)	'Event
Type Attributes	'Base
Range Attributes	'Range 'Reverse_range

Other user-defined attributes include: Enum-encoding, Flip-flop-type, Order_code, Part_name, Pin_numbers, Polarity, State_encoding, and State_Variable. See *Warp* documentation for details.

Number Representations

ABEL	VHDL	Radix
^b	b" " or " " or ' '(default)[20]	Binary
^o	o" "	Octal
^d (default)	Note 21	Decimal
^h	x" "	Hexadecimal

Notes

20. The default number representation in ABEL is decimal. The default number representation in VHDL is binary.

21. The default number representation in VHDL is binary. Decimal representations of numbers in VHDL require the user to define a signal or variable with type integer or use an integer number and then type-convert it to bit_vector. This is easier than it sounds. In the common library directory within *Warp* there is a file called LIBINT.VHD that contains a predefined function called i2bv. This function takes an integer and returns a bit_vector. So, using a decimal number is not too difficult, but one must know that an integer must be used and then type-converted to bit_vector.

For example:

ABEL syntax	VHDL syntax	Description
^b1	'1'	binary 1
^b0	'0'	binary 0
^b101010000	"10101000"	binary 10101000
^hF	x"F"	hex F
^hF1	x"F1"	hex F1
^hAAA	x"AAA"	hex AAA
^oF0F0	o"F0F0"	octal F0F0
^d23	i2bv(23,5)	decimal 23
^d99	i2bv(99,7)	decimal 99

Polarity Conventions

VHDL does not know whether a signal name should be interpreted as an active HIGH or an active LOW. Therefore, a signal named SHIFT4 will be interpreted logically the same as one named L_SHIFT4, and as one named SHIFT4_NOT. In other words, the behavioral equations must test with the proper level and assert with the desired level.

During logic synthesis and optimization, the software may determine that by flipping the polarity of a function the logic required will be optimized.

Identifiers

VHDL is not case sensitive, so a signal named SHIFT0 is identical to one named sHIFT0.

Resets and Presets

Although there are a variety of ways to specify a reset or preset, the best method is behavioral specification. If the reset or preset is asynchronous, use the following:

Place an IF-THEN-ELSIF-ENDIF inside a process with a (CLK'EVENT and CLK='1') placed as the condition for the ELSIF. In the first IF, place your reset and preset condition test and your signal assignments. In other words, the first part of the IF contains the asynchronous or combinatorial logic description and the second part, the ELSIF, contains the clocked logic description. In the process statement use a sensitivity list that includes the clock, and reset/preset for the design. Don't forget that statements in a process are considered sequential and are only updated upon changes in signals listed in the sensitivity list. See the basic example called COUNTER2.VHD and the real-world converted design example called FLAGCTLR below.

If the reset or preset is synchronous, place the condition inside the clocked portion of the IF-THEN-ELSIF-ENDIF mentioned above and perform the appropriate signal assignments.

This methodology ensures that behavioral operation is preserved and no device specific attributes are required.

Groups

ABEL allows declaration of groups or sets. Sets are groupings of signals. For example a bus is a set of signals. To create a set of signals in VHDL use the bit_vector type declaration. To perform Boolean operations on these new sets use IF-THEN-ELSE and FOR LOOPs to index the individual elements. See the special type conversion function and the real-world examples below.

Special VHDL Type Conversion Function (Advanced)

VHDL is a strongly typed language. ABEL on the other hand is not a strongly typed language. ABEL allows a user to mix Boolean operations with relational operations on sets. To concisely convert ABEL equations that contain relational operations on sets (converted to VHDL type BIT_VECTOR) combined with Boolean operations on signals (converted to VHDL type BIT), use the following type-conversion function. All equations requiring this type-conversion function call can be modified easily with a SEARCH and REPLACE text editor.

```
------------------------------ cut here ------------------------------------

FUNCTION frbl_to_b (in1:Boolean)     RETURN bit IS
BEGIN
IF (in1=TRUE) THEN
    RETURN '1';
ELSE
    RETURN '0';END IF;
END frbl_to_b;
--  This type conversion function converts a signal or relational operation
--  result from type BOOLEAN to type BIT.  A Boolean can have a value of
--  either 'TRUE' or 'FALSE'.  A bit can have a value of either '0' or '1'.

------------------------------ cut here ------------------------------------
```

For example if you had an equation in ABEL such as:

```
ramwr = !addren & ba16 & !write & ((addr ==^h210)
    # (addr==^h212)
    # (addr==^h214)
    # (addr==^h216));
```

Where addr is a set of 16 address bits,

This equation could be converted to VHDL in at least two ways:

```
ramwr <= not addren and ba16 and not write and (fr_bl_to_b(addr =x"210")
     or fr_bl_to_b(addr=x"212")
     or fr_bl_to_b(addr=x"214")
     or fr_bl_to_b(addr=x"216"));
```

OR,

```
ramwr <= not addren and ba15 and not write and(
     (not addr(11) and not addr(10) and addr(9) and not addr(8) and not addr(7)
      and not addr(6) and not addr(5) and addr(4) and not addr(3) and not addr(2)
      not addr(1) and not addr(0))
   OR  (not addr(11) and not addr(10) and addr(9) and not addr(8) and not
      addr(7) and not addr(6) and not addr(5) and addr(4) and not addr(3) and not
      addr(2) and addr(1) and not addr(0))
   OR  (not addr(11) and not addr(10) and addr(9) and not addr(8) and not
     addr(7) and not addr(6) and not addr(5) and addr(4) and not addr(3) and
      addr(2) and not addr(1) and not addr(0))
   OR (not addr(11) and not addr(10) and addr(9) and not addr(8) and not addr(7)
     and not addr(6) and not addr(5) and addr(4) and not addr(3) and addr(2) and
     addr(1) and not addr(0)));
```

This example assumes all of the signals from the ABEL equations are converted to signals of type BIT except the set called 'addr', which is converted to type BIT_VECTOR.

This special type-conversion function has a obvious advantage and is well suited for use in converting descriptions to VHDL. By no means is it a requirement that descriptions use this function. It should be used for one reason only, to make a VHDL description concise. See the real-world design example in Appendix A called FLAGCTLR.

State Machines

See the *Warp* design examples titled intermediate TRAFFIC.VHD, intermediate DRINK.VHD and advanced TTF.VHD. See the application note titled "Describing State Machines in *Warp*2 VHDL." Also refer to pitfall numbers five and seven below.

Decoders

See the *Warp* design example titled basic DECODER.VHD and the special type-conversion function above.

Comparators

See the *Warp* design examples titled intermediate COMPARE.VHD and COMPARE2.VHD.

Counters

See the *Warp* design examples titled basic COUNTER.VHD, basic COUNTER2.VHD, intermediate COUNTER3.VHD, advanced COUNTER4.VHD, and advanced COUNTER5.VHD.

Multiplexers

Use the truth table function that is shown in the application note titled "Describing State Machines in *Warp*2 VHDL" or create a multiplexer using Boolean equations.

Shift Registers

See the *Warp* design example titled advanced SHIFTN.VHD. This example illustrates the use of the GENERATE statement.

Adders

See the *Warp* design examples titled basic ADDER1.VHD and basic ADDER2.VHD.

Repetitive Logic

The VHDL GENERATE statement lends itself to regular or repetitive logic structures. For example, n-bit registers, n-bit counters, n-bit shift registers, n-bit multiplexers, n-bit adders, and n-bit comparators may be concisely described by using the GENERATE statement. See the *Warp* design examples titled advanced SHIFTN.VHD and advanced COUNTER4.VHD.

Pitfalls

There are potential pitfalls. Some of the common mistakes made during conversion are:

1. Incorrect order of precedence of operators. For instance, all of the logical operators in VHDL have the same level of precedence. In other words, an equation that has both AND and OR operators requires parenthesis around the ANDed terms for proper logic synthesis. Refer to the cross reference and order of precedence table above.
2. Incomplete separation of clocked signals from combinatorial signals. Two simple ways to ensure proper logic synthesis of clocked signals and combinatorial signals are:
   a. Use a process for all signals, but use an IF-THEN-ELSIF-ENDIF within the process that groups all combinatorial signals under the IF, and groups all registered signals under the ELSIF. See the real-world design example in Appendix A called FLAGCTLR.
   b. Place all registered signals within a process (using a WAIT UNTIL CLOCK ='1') and place all combinatorial signals outside the process.
3. Using loops and variables outside of a process. VHDL requires that loops and variables be used inside a process. If there is more than one process, signals communicate between processes.
4. Using the incorrect mode for either output or bidirectional signals. Refer to your language reference book for a formal definition of mode.
5. Incomplete state specification for state machines. When designing a state machine, you **MUST** do one of the following:
   a. Specify all output values in each state of the machine.
   or
   b. Specify default values for all outputs at the beginning of the process.

The reason for this has to do with the way a process works. Each time a process is run (i.e., a clock event has occurred) the outputs that are specified in the particular pass through the process are updated. If a branch exists within the states of the machine that allows a pass through the process with one or more outputs not assigned a value, the logic synthesis engine either (a) assumes that the last statement for an unassigned output is valid and should be latched, or (b) that it is allowed to change with the clock. In other words, it is legal in VHDL to not specify all output values in each state of the machine, or not specify default values for all outputs at the beginning of the process, or not specify either one. If this subtle detail is overlooked, the design will compile and appear to synthesize successfully, but functional operation may not be correct. It is also possible that the logic synthesized will not be minimal. In other words, use defaults or specify the value of all outputs within each state of the machine.

6. Incorrect set or reset operation found in simulation. Polarity optimization settings used during logic synthesis and fitting can cause set and or reset operations to appear to operate inconsistently. During logic synthesis and fitting, the fitter can decide, by flipping the polarity of a function, the logic required will be minimized. This can have an adverse effect on the user selection of set or reset. (Note this pitfall only applies to 22V10s and FLASH370 where the polarity inversion is located between the output of the register and the pin.) See the polarity attribute in the *Warp* documentation for more details.

7. Failure to close, or complete, IF-THEN-ELSE-ENDIF and CASE statements. In other words, design descriptions that contain an IF must contain an ELSE, and descriptions containing a CASE-WHEN (condition), must contain a WHEN-OTHERS statement. This is required so that unnecessary implicit memory elements are not synthesized. See the application note titled "VHDL Techniques for Optimal Design Fitting" for more information.

Logic Synthesis

Proper logic synthesis is the goal of conversion. If the converted design compiles and synthesizes without errors, but the logic equations in the report file are not as expected (or simulation results are not as desired) consult the pitfalls section above. Also, consult your *Warp* - GALAXY compiler options documentation and *Warp* - NOVA user's guide. If all else fails, contact your local Cypress field application engineer.

Real-World Converted Designs

The designs in Appendix A originally were intended to fit into MACH 110s. However, due to product term and internal fanout requirements, MACH 210s were required. The designs were later converted to FLASH371s. Consult your Cypress data book for more information on the CY7C371's architecture.

Summary

Any design that has been described in Data I/O's ABEL language can be converted to VHDL. From an overall capability perspective, VHDL can be considered a superset of ABEL. Two designs documented in Appendix A were successfully converted using the cross reference tables and helpful hints contained within this application note.

Appendix A. Real-World Converted Designs

```
---------------------------------- cut here ----------------------------------

  Module FLAGCTLR
  Title 'Flag Controller 1 - Uxx_xx
  Revision 01'

  "ALGORITHM
  "
  "

  FLAGCTLR           device 'mach210a';

  "Inputs:

      R_40MHZ      pin ; "
      H_FEP_S0     pin ; "
      H_FEP_S1     pin ; "
      H_FEP_S2     pin ; "
      H_FEP_S3     pin ; "
      H_FEP_SET    pin ; "
      L_FEP_WE     pin ; "
      H_PPA_S0     pin ; "
      H_PPA_S1     pin ; "
      H_PPA_S2     pin ; "
      H_PPA_S3     pin ; "
      H_PPA_SET    pin ; "
      L_PPA_WE     pin ; "
      H_PPB_S0     pin ; "
      H_PPB_S1     pin ; "
      H_PPB_S2     pin ; "
      H_PPB_S3     pin ; "
      H_PPB_SET    pin ; "
      L_PPB_WE     pin ; "
      L_RESET      pin ; "

  "Outputs:

      H_FA0     pin  istype 'reg,buffer'; "
      H_FA1     pin  istype 'reg,buffer'; "
      H_FA2     pin  istype 'reg,buffer'; "
      H_FA3     pin  istype 'reg,buffer'; "
      H_FA4     pin  istype 'reg,buffer'; "
      H_FA5     pin  istype 'reg,buffer'; "
      H_FA6     pin  istype 'reg,buffer'; "
      H_FA7     pin  istype 'reg,buffer'; "

      H_FB0     pin  istype 'reg,buffer'; "
      H_FB1     pin  istype 'reg,buffer'; "
      H_FB2     pin  istype 'reg,buffer'; "
      H_FB3     pin  istype 'reg,buffer'; "
      H_FB4     pin  istype 'reg,buffer'; "

      H_AB0     pin  istype 'reg,buffer'; "
      H_AB1     pin  istype 'reg,buffer'; "
```

Appendix A. Real-World Converted Designs (continued)

```
    H_AB2      pin  istype 'reg,buffer'; "
    H_AB3      pin  istype 'reg,buffer'; "
    H_AB4      pin  istype 'reg,buffer'; "

Declarations

    X = .X.;
    C = .C.;
    Z = .Z.;

    FA =     [H_FA7,H_FA6,H_FA5,H_FA4,
              H_FA3,H_FA2,H_FA1,H_FA0];

    FB =     [H_FB4,H_FB3,H_FB2,H_FB1,H_FB0];
    AB =     [H_AB4,H_AB3,H_AB2,H_AB1,H_AB0];
    PPA_SEL  = [H_PPA_S3,H_PPA_S2,H_PPA_S1,H_PPA_S0];
    PPB_SEL  = [H_PPB_S3,H_PPB_S2,H_PPB_S1,H_PPB_S0];
    FEP_SEL  = [H_FEP_S3,H_FEP_S2,H_FEP_S1,H_FEP_S0];

Equations

  FA.CLK = R_40MHZ;
  FB.CLK = R_40MHZ;
  AB.CLK = R_40MHZ;

  FA.RE  = !L_RESET;
  FB.RE  = !L_RESET;
  AB.RE  = !L_RESET;


  H_FA0.T = (!H_FA0.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h0)
           #  H_FA0.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h0)
           # !H_FA0.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h0)
           #  H_FA0.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h0));

  H_FA1.T = (!H_FA1.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h1)
           #  H_FA1.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h1)
           # !H_FA1.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h1)
           #  H_FA1.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h1));


  H_FA2.T = (!H_FA2.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h2)
           #  H_FA2.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h2)
           # !H_FA2.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h2)
           #  H_FA2.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h2));

  H_FA3.T = (!H_FA3.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h3)
           #  H_FA3.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h3)
           # !H_FA3.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h3)
           #  H_FA3.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h3));


  H_FA4.T = (!H_FA4.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h4)
           #  H_FA4.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h4)
           # !H_FA4.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h4)
```

Appendix A. Real-World Converted Designs (continued)

```
          #  H_FA4.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h4));

H_FA5.T = (!H_FA5.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h5)
          #  H_FA5.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h5)
          # !H_FA5.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h5)
          #  H_FA5.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h5));


H_FA6.T = (!H_FA6.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h6)
          #  H_FA6.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h6)
          # !H_FA6.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h6)
          #  H_FA6.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h6));

H_FA7.T = (!H_FA7.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h7)
          #  H_FA7.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h7)
          # !H_FA7.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h7)
          #  H_FA7.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h7));


H_FB0.T = (!H_FB0.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h0)
          #  H_FB0.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h0)
          # !H_FB0.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h8)
          #  H_FB0.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h8));

H_FB1.T = (!H_FB1.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h1)
          #  H_FB1.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h1)
          # !H_FB1.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h9)
          #  H_FB1.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h9));


H_FB2.T = (!H_FB2.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h2)
          #  H_FB2.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h2)
          # !H_FB2.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^ha)
          #  H_FB2.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^ha));

H_FB3.T = (!H_FB3.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h3)
          #  H_FB3.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h3)
          # !H_FB3.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hb)
          #  H_FB3.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hb));


H_AB0.T = (!H_AB0.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h8)
          #  H_AB0.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h8)
          # !H_AB0.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h8)
          #  H_AB0.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h8));

H_AB1.T = (!H_AB1.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h9)
          #  H_AB1.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h9)
          # !H_AB1.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h9)
          #  H_AB1.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h9));


H_AB2.T = (!H_AB2.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^ha)
          #  H_AB2.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^ha)
          # !H_AB2.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^ha)
```

Appendix A. Real-World Converted Designs (continued)

```
            #  H_AB2.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^ha));

  H_AB3.T = (!H_AB3.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hb)
            #  H_AB3.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hb)
            # !H_AB3.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hb)
            #  H_AB3.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hb));


  H_FB4.T = (!H_FB4.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h4)
            #  H_FB4.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h4)
            # !H_FB4.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hc)
            #  H_FB4.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hc));


  H_AB4.T = (!H_AB4.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hc)
            #  H_AB4.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hc)
            # !H_AB4.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hc)
            #  H_AB4.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hc));

test_vectors ([R_40MHZ,L_RESET,
               L_FEP_WE, FEP_SEL, H_FEP_SET,
               L_PPA_WE, PPA_SEL, H_PPA_SET,
               L_PPB_WE, PPB_SEL, H_PPB_SET]
           -> [H_FA7, H_FA6, H_FA5, H_FA4, H_FA3, H_FA2, H_FA1, H_FA0,
               H_FB4, H_FB3, H_FB2, H_FB1, H_FB0,
               H_AB4, H_AB3, H_AB2, H_AB1, H_AB0])
[C,1,1,^h0,0,1,^h1,0,1,^h0,0]->[X,X,X,X,X,X,X,X, X,X,X,X,X, X,X,X,X,X];
[C,1,0,^h0,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h1,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h2,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h3,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h4,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h5,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h6,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h7,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h8,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h9,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^hA,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^hB,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^hC,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h0,1,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h1,1,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h2,1,1,^h1,0,1,^h0,0]->[0,0,0,0,0,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h3,1,1,^h1,0,1,^h0,0]->[0,0,0,0,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h4,1,1,^h1,0,1,^h0,0]->[0,0,0,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h5,1,1,^h1,0,1,^h0,0]->[0,0,1,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h6,1,1,^h1,0,1,^h0,0]->[0,1,1,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h7,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h8,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,0,0,1, 0,0,0,0,0];
[C,1,0,^h9,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,0,1,1, 0,0,0,0,0];
[C,1,0,^hA,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,1,1,1, 0,0,0,0,0];
[C,1,0,^hB,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,1,1,1,1, 0,0,0,0,0];
[C,1,0,^hC,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 0,0,0,0,0];
[C,1,1,^h7,1,0,^h0,0,1,^h0,0]->[1,1,1,1,1,1,1,0, 1,1,1,1,1, 0,0,0,0,0];
```

Appendix A. Real-World Converted Designs (continued)

```
  [C,1,1,^h7,1,0,^h1,0,1,^h0,0]->[1,1,1,1,1,1,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h2,0,1,^h0,0]->[1,1,1,1,1,1,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h3,0,1,^h0,0]->[1,1,1,1,1,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h4,0,1,^h0,0]->[1,1,1,1,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h5,0,1,^h0,0]->[1,1,0,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h6,0,1,^h0,0]->[1,0,0,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h7,0,1,^h0,0]->[0,0,0,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h0,0]->[0,0,0,0,0,0,0,0,0, 1,1,1,1,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h1,0]->[0,0,0,0,0,0,0,0,0, 1,1,1,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h2,0]->[0,0,0,0,0,0,0,0,0, 1,1,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h3,0]->[0,0,0,0,0,0,0,0,0, 1,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h4,0]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h8,0]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h9,0]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hA,0]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hB,0]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hC,0]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hC,1]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hB,1]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hA,1]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,1,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h9,1]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,1,1,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h8,1]->[0,0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h4,1]->[0,0,0,0,0,0,0,0,0, 1,0,0,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h3,1]->[0,0,0,0,0,0,0,0,0, 1,1,0,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h2,1]->[0,0,0,0,0,0,0,0,0, 1,1,1,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h1,1]->[0,0,0,0,0,0,0,0,0, 1,1,1,1,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h0,1]->[0,0,0,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h7,1,1,^h0,1]->[1,0,0,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h6,1,1,^h0,1]->[1,1,0,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h5,1,1,^h0,1]->[1,1,1,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h4,1,1,^h0,1]->[1,1,1,1,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h3,1,1,^h0,1]->[1,1,1,1,1,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h2,1,1,^h0,1]->[1,1,1,1,1,1,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h1,1,1,^h0,1]->[1,1,1,1,1,1,1,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h0,1,1,^h0,1]->[1,1,1,1,1,1,1,1,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h8,0,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,0];
  [C,1,1,^h7,1,0,^h9,0,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,0,0];
  [C,1,1,^h7,1,0,^hA,0,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,0,0,0];
  [C,1,1,^h7,1,0,^hB,0,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,0,0,0,0];
  [C,1,1,^h7,1,0,^hC,0,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^hC,1,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,0,0,0,0];
  [C,1,1,^h7,1,0,^hB,1,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,0,0,0];
  [C,1,1,^h7,1,0,^hA,1,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,0,0];
  [C,1,1,^h7,1,0,^h9,1,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,0];
  [C,1,1,^h7,1,0,^h8,1,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h0,1,1,^h0,1]->[1,1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1];

END FLAGCTLR;

------------------------------------ cut here ------------------------------------

--  Converted to IEEE 1076 VHDL

--  Module FLAGCTLR
```

Appendix A. Real-World Converted Designs (continued)

```
--  Title 'Flag Controller 1 - Uxx_xx
--  Revision 01'

use work.cypress.all;
use work.rtlpkg.all;
use work.int_math.all;

ENTITY FLAGCTLR IS PORT(
  R_40MHZ,H_FEP_SET,L_FEP_WE,H_PPA_SET,
  L_PPA_WE,H_PPB_SET,L_PPB_WE,L_RESET    : IN BIT;
  PPA_SEL,PPB_SEL,FEP_SEL                : IN BIT_VECTOR (3 downto 0);
  FA                                     : INOUT BIT_VECTOR (7 downto 0);
  FB,AB                                  : INOUT BIT_VECTOR (4 downto 0));
attribute part_name of eventflg: entity is "c371";
END FLAGCTLR;

ARCHITECTURE CONVERTED_ABL OF FLAGCTLR IS

FUNCTION frbl_to_b(in1:Boolean)     RETURN bit IS
BEGIN
IF (in1=true) THEN
     RETURN '1';
ELSE
     RETURN '0';
END IF;
END frbl_to_b;
--  This type conversion function converts a signal or relational operation
--  result from type BOOLEAN to type BIT.  A Boolean can have a value of either
-- 'TRUE' or'FALSE'.  A bit can have a value of either '0' or '1'.

BEGIN
PROCESS (R_40MHZ, L_RESET)
BEGIN

IF (L_RESET ='0') THEN

      FOR i IN 0 TO 4 LOOP
           FA(i) <= '0'; FB(i) <= '0'; AB(i) <= '0';
      END LOOP;

      FOR i IN 5 TO 7 LOOP
           FA(i) <= '0';
      END LOOP;

ELSIF (R_40MHZ'EVENT AND R_40MHZ ='1') THEN

  FA(0) <= FA(0) XOR
         ((NOT FA(0) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"0"))
      OR (FA(0) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"0"))
      OR (NOT FA(0) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"0"))
      OR (FA(0) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"0")));

  FA(1) <= FA(1) XOR
         ((NOT FA(1) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"1"))
      OR (FA(1) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"1"))
```

Appendix A. Real-World Converted Designs (continued)

```
    OR (NOT FA(1) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"1"))
    OR (FA(1) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"1")));

FA(2) <= FA(2) XOR
       ((NOT FA(2) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"2"))
    OR (FA(2) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"2"))
    OR (NOT FA(2) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"2"))
    OR (FA(2) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"2")));

FA(3) <= FA(3) XOR
       ((NOT FA(3) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"3"))
    OR (FA(3) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"3"))
    OR (NOT FA(3) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"3"))
    OR (FA(3) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"3")));

FA(4) <= FA(4) XOR
       ((NOT FA(4) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"4"))
    OR (FA(4) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"4"))
    OR (NOT FA(4) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"4"))
    OR (FA(4) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"4")));

FA(5) <= FA(5) XOR
       ((NOT FA(5) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"5"))
    OR (FA(5) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"5"))
    OR (NOT FA(5) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"5"))
    OR (FA(5) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"5")));

FA(6) <= FA(6) XOR
       ((NOT FA(6) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"6"))
    OR (FA(6) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"6"))
    OR (NOT FA(6) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"6"))
    OR (FA(6) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"6")));

FA(7) <= FA(7) XOR
       ((NOT FA(7) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"7"))
    OR (FA(7) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"7"))
    OR (NOT FA(7) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"7"))
    OR (FA(7) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"7")));

FB(0) <= FB(0) XOR
       ((NOT FB(0) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"0"))
    OR (FB(0) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"0"))
    OR (NOT FB(0) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"8"))
    OR (FB(0) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"8")));

FB(1) <= FB(1) XOR
       ((NOT FB(1) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"1"))
    OR (FB(1) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"1"))
    OR (NOT FB(1) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"9"))
    OR (FB(1) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"9")));

FB(2) <= FB(2) XOR
       ((NOT FB(2) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"2"))
    OR (FB(2) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"2"))
    OR (NOT FB(2) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"A"))
```

Appendix A. Real-World Converted Designs (continued)

```
      OR (FB(2) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"A")));

  FB(3) <= FB(3) XOR
         ((NOT FB(3) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"3"))
      OR (FB(3) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"3"))
      OR (NOT FB(3) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"B"))
      OR (FB(3) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"B")));

  AB(0) <= AB(0) XOR
         ((NOT AB(0) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"8"))
      OR (AB(0) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"8"))
      OR (NOT AB(0) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"8"))
      OR (AB(0) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"8")));

  AB(1) <= AB(1) XOR
         ((NOT AB(1) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"9"))
      OR (AB(1) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"9"))
      OR (NOT AB(1) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"9"))
      OR (AB(1) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"9")));

  AB(2) <= AB(2) XOR
         ((NOT AB(2) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"A"))
      OR (AB(2) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"A"))
      OR (NOT AB(2) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"A"))
      OR (AB(2) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"A")));

  AB(3) <= AB(3) XOR
         ((NOT AB(3) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"B"))
      OR (AB(3) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"B"))
      OR (NOT AB(3) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"B"))
      OR (AB(3) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"B")));

  FB(4) <= FB(4) XOR
         ((NOT FB(4) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"4"))
      OR (FB(4) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"4"))
      OR (NOT FB(4) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"C"))
      OR (FB(4) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"C")));

  AB(4) <= AB(4) XOR
         ((NOT AB(4) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"C"))
      OR (AB(4) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"C"))
      OR (NOT AB(4) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"C"))
      OR (AB(4) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"C")));

END IF;
END PROCESS;
END CONVERTED_ABL;

---------------------------------- cut here ----------------------------------

Module CONVERTER
Title 'Converter
Revision 01'

" This device converts a 32-bit floating point word from one format to another.
```

Appendix A. Real-World Converted Designs (continued)

```
CONVERTER                       device       'MACH210A';

"Control Inputs:

CLK             PIN 35;" Clock
OE              PIN 10;" Low Active Output Enable
WE              PIN 11;" Low Active Write Enable
MODE            PIN 13;" Shift Mode

"Data I/O BITS:

D31             PIN 43   ISTYPE    'REG,BUFFER';
D30             PIN 42   ISTYPE    'REG,BUFFER';
D29             PIN 41   ISTYPE    'REG,BUFFER';
D28             PIN 40   ISTYPE    'REG,BUFFER';
D27             PIN 39   ISTYPE    'REG,BUFFER';
D26             PIN 38   ISTYPE    'REG,BUFFER';
D25             PIN 37   ISTYPE    'REG,BUFFER';
D24             PIN 36   ISTYPE    'REG,BUFFER';
D23             PIN 31   ISTYPE    'REG,BUFFER';
D22             PIN 30   ISTYPE    'REG,BUFFER';
D21             PIN 29   ISTYPE    'REG,BUFFER';
D20             PIN 28   ISTYPE    'REG,BUFFER';
D19             PIN 27   ISTYPE    'REG,BUFFER';
D18             PIN 26   ISTYPE    'REG,BUFFER';
D17             PIN 25   ISTYPE    'REG,BUFFER';
D16             PIN 24   ISTYPE    'REG,BUFFER';
D15             PIN 21   ISTYPE    'REG,BUFFER';
D14             PIN 20   ISTYPE    'REG,BUFFER';
D13             PIN 19   ISTYPE    'REG,BUFFER';
D12             PIN 18   ISTYPE    'REG,BUFFER';
D11             PIN 17   ISTYPE    'REG,BUFFER';
D10             PIN 16   ISTYPE    'REG,BUFFER';
D09             PIN 15   ISTYPE    'REG,BUFFER';
D08             PIN 14   ISTYPE    'REG,BUFFER';
D07             PIN  9   ISTYPE    'REG,BUFFER';
D06             PIN  8   ISTYPE    'REG,BUFFER';
D05             PIN  7   ISTYPE    'REG,BUFFER';
D04             PIN  6   ISTYPE    'REG,BUFFER';
D03             PIN  5   ISTYPE    'REG,BUFFER';
D02             PIN  4   ISTYPE    'REG,BUFFER';
D01             PIN  3   ISTYPE    'REG,BUFFER';
D00             PIN  2   ISTYPE    'REG,BUFFER';

H,L,C,Z,X   =   1,0,.C.,.Z.,.X.;

DIN         =   [D19.PIN,D18.PIN,D17.PIN,D16.PIN,
                 D15.PIN,D14.PIN,D13.PIN,D12.PIN,
                 D11.PIN,D10.PIN,D09.PIN,D08.PIN,
                 D07.PIN,D06.PIN,D05.PIN,D04.PIN,
                 D03.PIN,D02.PIN,D01.PIN,D00.PIN];

DOUT        =   [D31,D30,D29,D28,D27,D26,D25,D24,
                 D23,D22,D21,D20,D19,D18,D17,D16,
```

Appendix A. Real-World Converted Designs (continued)

```
                D15,D14,D13,D12,D11,D10,D09,D08,
                D07,D06,D05,D04,D03,D02,D01,D00];

DBIDI       =   [D19,D18,D17,D16,D15,D14,D13,D12,
                D11,D10,D09,D08,D07,D06,D05,D04,
                D03,D02,D01,D00];

DOUTFB      =   [L,L,L,L,L,L,L,L,L,L,L,L,
                D19.FB,D18.FB,D17.FB,D16.FB,
                D15.FB,D14.FB,D13.FB,D12.FB,
                D11.FB,D10.FB,D09.FB,D08.FB,
                D07.FB,D06.FB,D05.FB,D04.FB,
                D03.FB,D02.FB,D01.FB,D00.FB];

M0_SHIFT6   =   !WE & !MODE &
                ((D19.PIN == H) # (D18.PIN == H) # (D17.PIN == H));

M0_SHIFT3   =   !WE & !MODE &
                (((D19.PIN == L) & (D18.PIN == L) & (D17.PIN == L)) &
                ((D16.PIN == H) # (D15.PIN == H) # (D14.PIN == H)));

SHIFT0      =   !WE &
                (((D19.PIN == L) & (D18.PIN == L) & (D17.PIN == L)) &
                ((D16.PIN == L) & (D15.PIN == L) & (D14.PIN == L)));

M1_SHIFT6   =   !WE & MODE &
                ((D19.PIN == H) # (D18.PIN == H));

M1_SHIFT4   =   !WE & MODE &
                (((D19.PIN == L) & (D18.PIN == L)) &
                ((D17.PIN == H) # (D16.PIN == H)));

M1_SHIFT2   =   !WE & MODE &
                (((D19.PIN == L) & (D18.PIN == L) &
                (D17.PIN == L) & (D16.PIN == L)) &
                ((D15.PIN == H) # (D14.PIN == H)));

EQUATIONS

DOUT.CLK    =   CLK;

DOUT.OE     =   !OE;
DOUT        :=  M0_SHIFT6  &
                        [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,L,D19.PIN,D18.PIN,
                             D17.PIN,D16.PIN,D15.PIN,D14.PIN,D13.PIN,D12.PIN,
                             D11.PIN,D10.PIN,D09.PIN,D08.PIN,D07.PIN,D06.PIN]

              #

                M0_SHIFT3  &
                        [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,D16.PIN,D15.PIN,
                             D14.PIN,D13.PIN,D12.PIN,D11.PIN,D10.PIN,D09.PIN,
                             D08.PIN,D07.PIN,D06.PIN,D05.PIN,D04.PIN,D03.PIN]

              #

                SHIFT0      &
                        [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,D13.PIN,D12.PIN,
```

Appendix A. Real-World Converted Designs (continued)

```
                    D11.PIN,D10.PIN,D09.PIN,D08.PIN,D07.PIN,D06.PIN,
                    D05.PIN,D04.PIN,D03.PIN,D02.PIN,D01.PIN,D00.PIN]

        #

          M1_SHIFT6  &
                   [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,H,D19.PIN,D18.PIN,
                    D17.PIN,D16.PIN,D15.PIN,D14.PIN,D13.PIN,D12.PIN,
                    D11.PIN,D10.PIN,D09.PIN,D08.PIN,D07.PIN,D06.PIN]

        #

          M1_SHIFT4  &
                   [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,L,D17.PIN,D16.PIN,
                    D15.PIN,D14.PIN,D13.PIN,D12.PIN,D11.PIN,D10.PIN,
                    D09.PIN,D08.PIN,D07.PIN,D06.PIN,D05.PIN,D04.PIN]

        #

          M1_SHIFT2  &
                   [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,D15.PIN,D14.PIN,
                    D13.PIN,D12.PIN,D11.PIN,D10.PIN,D09.PIN,D08.PIN,
                    D07.PIN,D06.PIN,D05.PIN,D04.PIN,D03.PIN,D02.PIN]

        #

          WE & DOUTFB;

Test_Vectors

([CLK,OE,WE,MODE,                DBIDI]          ->       DOUT)
 [C,H,L,X,                      ^b00000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000000101101110110;"Read,Shift0
 [C,H,L,L,                      ^b00000110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000100110101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00001010101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101010101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00010010101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000110010101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00011010101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111010101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00010110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000110110101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00011110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101110101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00011110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111110101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00111110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000001000111110101101;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b01011110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000001001011110101101;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b10011110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000001010011110101101;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b10100110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000001010100110101101;"Read,Shift6/Mode0
```

Appendix A. Real-World Converted Designs (continued)

```
 [C,H,L,L,                      ^b01100110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000100110011010110 1;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b11000110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101100011010110 1;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b11111110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101111111010110 1;"Read,Shift6/Mode0
 [C,H,L,L,H,                    ^b00000100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101001011011101;"Read,Shift2/Mode1
 [C,H,L,L,H,                    ^b00001000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000110001011011101;"Read,Shift2/Mode1
 [C,H,L,L,H,                    ^b00001100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111001011011101;"Read,Shift2/Mode1
 [C,H,L,L,H,                    ^b00010000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000100100001011011 1;"Read,Shift4/Mode1
 [C,H,L,L,H,                    ^b00100000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101000001011011 1;"Read,Shift4/Mode1
 [C,H,L,L,H,                    ^b00110000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101100001011011 1;"Read,Shift4/Mode1
 [C,H,L,L,H,                    ^b00111100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101111001011011 1;"Read,Shift4/Mode1
 [C,H,L,L,H,                    ^b10000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111000000010110 1;"Read,Shift6/Mode1
 [C,H,L,L,H,                    ^b01000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000110100000010110 1;"Read,Shift6/Mode1
 [C,H,L,L,H,                    ^b11000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111100000010110 1;"Read,Shift6/Mode1
 [C,H,L,L,H,                    ^b11010100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111010100101101;"Read,Shift6/Mode1
 [C,H,L,L,H,                    ^b11101000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111110100010110 1;"Read,Shift6/Mode1
 [C,H,L,L,H,                    ^b11111100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111111110010110 1;"Read,Shift6/Mode1

End CONVERTER;

---------------------------------- cut here ----------------------------------

-- CONVERTED TO IEEE 1076 VHDL

-- Module CONVERTER
-- Title 'Converter
-- Revision 01'

-- This device converts a 32-bit floating point word from one format to another.

-- Control Inputs

use work.cypress.all;
use work.rtlpkg.all;
use work.int_math.all;

ENTITY CONVERTER IS PORT(
CLK,OE,WE,MODE                                  : IN BIT;
D                                               : INOUT X01Z_VECTOR (0 TO 31));
```

Appendix A. Real-World Converted Designs (continued)

```
attribute part_name of CONVERTER: entity is "c371";
END CONVERTER;

ARCHITECTURE CONVERTED_ABL OF CONVERTER IS
SIGNAL SHIFT2_TMP, SHIFT1_TMP, SHIFT0_TMP, SHIFT2,
       SHIFT1, SHIFT0, M0_SHIFT6, M0_SHIFT3,
       M_SHIFT0, M1_SHIFT6, M1_SHIFT4, M1_SHIFT2      : BIT;
SIGNAL D_TMP, D_FB                                    : BIT_VECTOR (0 TO 31);
BEGIN

P1: PROCESS
 BEGIN
  WAIT UNTIL CLK ='1';

   FOR i IN 0 TO 13 LOOP
   D_TMP(i) <=(D_FB(i+6) AND      SHIFT2 AND      SHIFT1 AND NOT SHIFT0)
           OR (D_FB(i+4) AND      SHIFT2 AND NOT SHIFT1 AND NOT SHIFT0)
           OR (D_FB(i+3) AND NOT SHIFT2 AND      SHIFT1 AND      SHIFT0)
           OR (D_FB(i+2) AND NOT SHIFT2 AND      SHIFT1 AND NOT SHIFT0)
           OR (D_FB(i+0) AND NOT SHIFT2 AND NOT SHIFT1 AND NOT SHIFT0)
           OR (WE AND D_TMP(i));
   END LOOP;

   D_TMP(14) <= ('0' AND M0_SHIFT6) OR ('1' AND M0_SHIFT3)
           OR ('0' AND  M_SHIFT0) OR ('1' AND M1_SHIFT6)
           OR ('0' AND M1_SHIFT4) OR ('1' AND M1_SHIFT2)
           OR (WE AND D_TMP(14));

   D_TMP(15) <= ('1' AND M0_SHIFT6) OR ('0' AND M0_SHIFT3)
           OR ('0' AND  M_SHIFT0) OR ('1' AND M1_SHIFT6)
           OR ('1' AND M1_SHIFT4) OR ('0' AND M1_SHIFT2)
           OR (WE AND D_TMP(15));

   FOR i IN 16 TO 28 LOOP
   D_TMP(i) <='0';
   END LOOP;

END PROCESS P1;

M0_SHIFT6  <= (NOT WE AND MODE) AND
              (D_FB(19) OR D_FB(18) OR D_FB(17));

M0_SHIFT3  <= (NOT WE AND NOT MODE AND NOT D_FB(19) AND NOT D_FB(18)
               AND NOT D_FB(17)) AND (D_FB(16) OR D_FB(15) OR D_FB(14));

M_SHIFT0   <= NOT WE AND NOT D_FB(19) AND NOT D_FB(18) AND NOT
              D_FB(17) AND NOT D_FB(16) AND NOT D_FB(15) AND NOT D_FB(14);

M1_SHIFT6  <= (NOT WE AND MODE) AND (D_FB(19) OR D_FB(18));

M1_SHIFT4  <= (NOT WE AND MODE AND NOT D_FB(19) AND NOT D_FB(18))
               AND (D_FB(17) OR D_FB(16));

M1_SHIFT2  <= (NOT WE AND MODE AND NOT D_FB(19) AND NOT D_FB(18) AND
               NOT D_FB(17) AND NOT D_FB(16)) AND (D_FB(15) OR D_FB(14));
```

Appendix A. Real-World Converted Designs (continued)

```
SHIFT2_TMP <= M0_SHIFT6
              OR M1_SHIFT6
              OR M1_SHIFT4;

SHIFT1_TMP <= M0_SHIFT6
              OR M0_SHIFT3
              OR M1_SHIFT6
              OR M1_SHIFT2;

SHIFT0_TMP <= M0_SHIFT3;

-- Mapping for the Bidirectional buffers
-- D_TMP is the internal signal which drives the output buffer
-- OE    is the signal for output enable (active high)
-- D     is the pin name, matches name in port assignment
-- D_FB  is the signal from pin that feeds back and drives the internal
--       structure
G1: FOR i IN 0 TO 28 GENERATE
      B1:BUFOE PORT MAP(D_TMP(i), OE,D(i), D_FB(i));
END GENERATE;

B2: BUF PORT MAP(SHIFT0_TMP,SHIFT0); -- Forces logic synthesis to "split
B3: BUF PORT MAP(SHIFT1_TMP,SHIFT1); -- sums" into SHIFT codes that are
B4: BUF PORT MAP(SHIFT2_TMP,SHIFT2); -- encoded and placed on outputs D29-31

B5: BUFOE PORT MAP(SHIFT0, OE, D(29), open);
B6: BUFOE PORT MAP(SHIFT1, OE, D(30), open);
B7: BUFOE PORT MAP(SHIFT2, OE, D(31), open);

END CONVERTED_ABL;

---------------------------------- cut here ----------------------------------
```

Abel™ –HDL vs. IEEE–1076 VHDL

Abstract

Currently there exist several popular Hardware Description Languages (HDLs) that allow designers to describe the function of complex logic circuits textually, as opposed to schematically. One of the most widely used of these languages is Data I/O's Abel–HDL. Abel–HDL, as a language, can be used to describe the behavior of logic circuits that can be fitted to a wide variety of PALs, PLDs, PROMs, and FPGAs from a variety of programmable logic IC manufacturers. IEEE–1076 VHDL (VHSIC Hardware Description Language) has recently been gaining widespread support. VHDL is an open, portable language defined and standardized by the IEEE that can be used to describe the behavior of an entire system from the highest levels of functionality all the way down to the logic-gate level. A majority of CAE vendors, programmable IC manufacturers, and third-party software vendors already have or are planning tools that support VHDL logic synthesis, logic modeling, and/or VHDL simulation.

The purpose of this application note is to compare and contrast the complexity and basic features of Abel–HDL with those of IEEE–1076 VHDL. Both of these languages are very robust in their support of different types of constructs that can be used to describe the same functionality at different levels of abstraction. It is beyond the scope of this document to exhaustively describe these possibilities or to present a complete tutorial for writing code in either language because of the great variety of constructs and syntax available with which to describe the functionality of a given circuit. Rather, a simple example design that contains a mixture of synchronous and asynchronous logic circuits will be shown. Sample code is written in both Abel–HDL and VHDL that describes the example's functionality and synthesizes to create functionally identical hardware. The code written here represents a typical level of abstraction that balances readability with compactness. With experience, designers can develop their own preferences for style. For instance, state machines can be described in a number of ways: state tables, IF-THEN-ELSE statements, CASE-WHEN statements, or explicitly using a combination of Register-Transfer-Level (RTL) code (individually describe each gate/register as a component with its inputs and outputs) and/or Boolean equations.

Example Description

The following example is a circuit that creates a 50% duty cycle clock with programmable frequency. *Figure 1* shows the block diagram of this Programmable Clock Generator. The output of the circuit is CLK_OUT, whose period is equal to clock(ns)*incnt*2. To program the device, LD_CNT is used to latch the value present at the INCNT(3:0) inputs into the 4-Bit Input Register. The output of this register is ENDCNT(3:0). The clock input is used to clock the 4-bit Up/Down Counter, whose output is COUNT(3:0). The 4-bit Comparator is an asynchronous comparator that compares the values of COUNT and ENDCNT. Its outputs are endeqcnt (ENDCNT = COUNT), endltcnt (ENDCNT < COUNT), endeq0 (ENDCNT = 0), and cnteq0 (COUNT = 0). Note that it is possible for ENDCNT to be less than COUNT if a new value for ENDCNT is loaded into the input registers that is less than the current value of COUNT. The cnt_state State Machine controls the CLK_OUT signal and is clocked and enabled by the clock and en inputs, respectively.

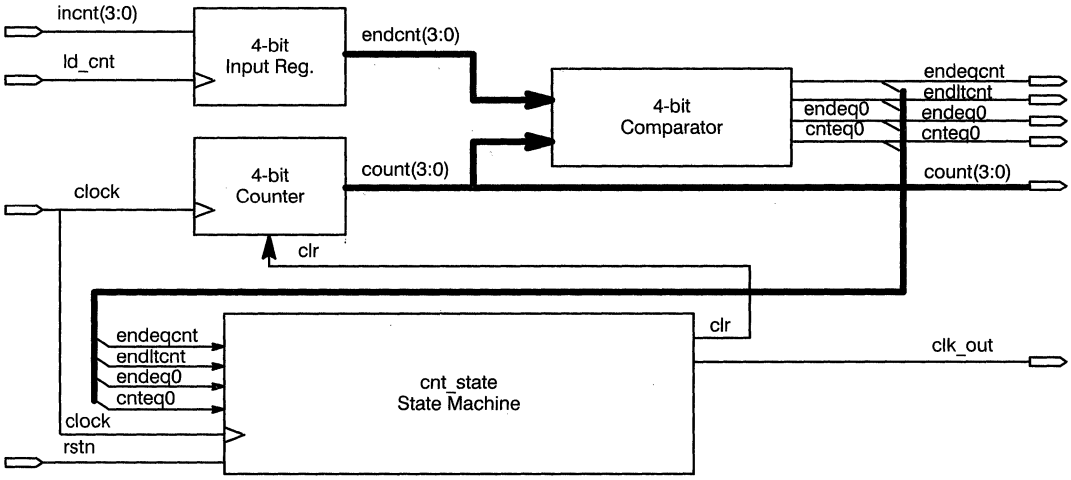


Figure 1. Block Diagram

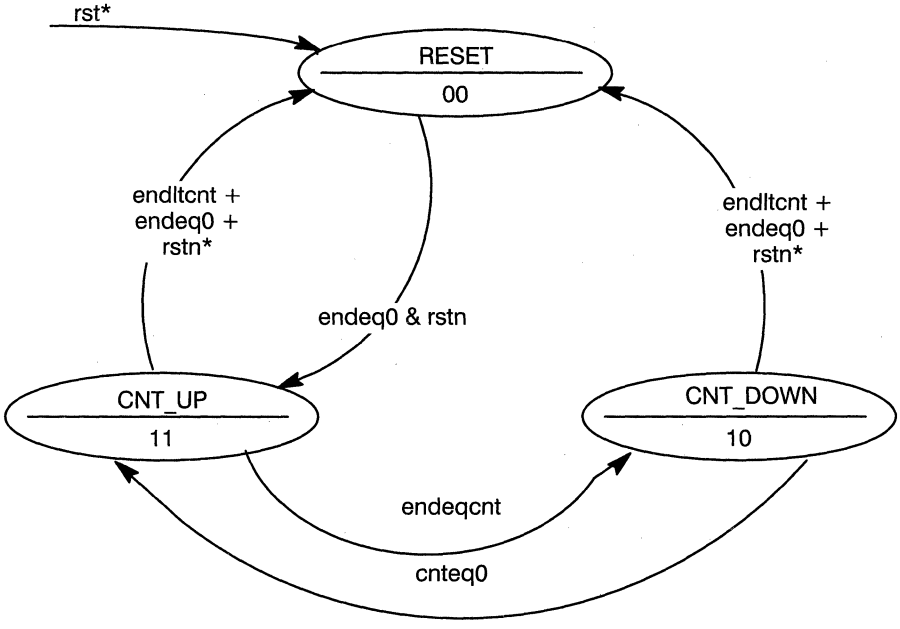


Figure 2. State Diagram

Figure 2 shows the state diagram for cnt_state. The state machine consists of three states: *reset*, *cnt_up*, and *cnt_down*. Two state bits are used to describe this Mealy-type state machine. The state machine powers-up to the reset state. It also synchronously enters the *reset* state from any state if RSTN=0. Once RSTN=1, the state machine will count up until COUNT equals ENDCNT, at which point it will begin to count down until COUNT equals 0 and then repeat the cycle. If at any time ENDCNT = 0, cnt_state will return to *reset*. Also, if ENDCNT is ever less than COUNT, thus signifying an invalid condition, cnt_state will go to *reset*.

The PLD targeted is the CY7C335. It was chosen because it can be used to illustrate a variety of features contained in some PLDs such as input registers, multiple clocks, buried registers, and synchronous reset/preset. However, any PLD with sufficient resources could be targeted. This is one of the main advantages of using a HDL. High-level languages, by design, allow a designer to write generic code that can be targeted to different devices/architectures with little or no modification. Going one step further, VHDL allows simulation and debugging of the logic from the source code. This can greatly reduce the overall design cycle time by allowing functional verification of the logic prior to targeting a specific physical device. Once the logic has been verified, the designer can then compile and fit the same design into a variety of devices. From here, the designer can decide which implementation best suits his requirements.

Abel−HDL vs. VHDL

In general, the constructs used to describe logic function in both ABEL−HDL an IEEE−1076 VHDL are very similar. Each can accept Boolean equations, truth tables, state descriptions (IF-THEN and CASE-WHEN), and signal assignments that are quite similar in appearance. However, differences do exist in syntax and in the overhead sections. These are the declarative sections which define hierarchical organization, design libraries, etc. As we shall see, some of these statements found in the VHDL code have no direct counterpart in Abel−HDL. This is because Abel−HDL was created with a single purpose in mind, logic synthesis for PALs and PLDs. Therefore, with Abel−HDL, assumptions can be made by the compiler which can simplify the overhead syntax needed. Because VHDL is a one-language-fits-all standard which applies to synthesis, modeling, and system definition at all levels, some syntax overhead is necessary to fully describe a design in the proper context. For example, the use of standard and user-definable packages and libraries allows many designs to share commonly used definitions, components, macros, functions, etc. Fortunately for the logic designer, these statements are used in most designs so that familiarity with them comes quickly and easily. This commonality also allows ample "cutting-and-pasting" from design to design.

The following sections compare and contrast the source code files for Abel−HDL and VHDL on a logical section-by-section basis. Both of these files, when compiled, create functionally identical logic. Copies of the source code files in their entirety can be found in the Appendices.

Design and I/O Declarations

The basic structure of both Abel−HDL and VHDL source files allow for one or more design units to be defined within it. Each design unit is a complete logic description. Multiple design units may be combined hierarchically in a single top-level source file which binds them together. In the example, we have used a single design unit for simplicity. The first section of code chosen for comparison contains the design declaration and device I/O declarations. In this example, the pin numbers have been fixed. This optional in either language. The declaration of the target device is also optional in the source code itself. The targeted device need not be declared until it's time to compile and fit the logic.

Figure 3 contains the Abel−HDL code that declares the module name (line 01), the device (line 03), and the input and output pin numbers and types (lines 04−09). Line 45 is the end statement that completes the design module.

The corresponding VHDL code is shown in *Figure 4*. VHDL requires a slightly different structure. A design unit consists of an Entity section and an Ar-

```
01: module clk_gena;

02: declarations
03: device 'p335';

    "Inputs
04:  clock, ld_cnt, rstn                pin 3,1,7;
05:  incnt3, incnt2, incnt1, incnt0     pin 6,5,4,2;

    "Outputs
06:  endeqcnt, endltcnt                 pin 25,23 istype 'com';
07:  endeq0, cnteq0                     pin 24,27 istype 'com';
08:  clk_out                            pin 17 istype 'reg_d';
09:  count3, count2, count1, count0     pin 19,15,28,26;
.
.
.
45: end clk_gena;
```

Figure 3. Abel−HDL Design and I/O Declarations

chitecture section. By themselves, each of these is considered a separate design component. The Entity section defines the component name (line 01). The port statement (lines 02−06) declares the I/O of the entity. For each signal, a mode (in, out, buffer) defines the direction of the pin (buffer signifies output with feedback). The signal type is also defined here. The type of a signal defines size and possible values which that signal can take on. Type bit defines a one-bit signal that can have the values of "0" and "1". Type bit_vector (0 to 3) declares that the signal is a 4-bit vector, each bit of which can take on the values of "0" or "1". Line 07 declares the target device and is optional. Lines 08−12 declares the fixed pin assignments and is also optional. Note that this line could be written as a single line terminated with a ";". The "&" in this context signifies a continuation from the previous line. Line 13 is the end statement that terminates the *clk_genv* entity.

Lines 14 and 15 are statements that call out other libraries and packages making them visible to this design. These libraries and packages may be predefined in the VHDL language or may be user defined. They may contain components, functions, procedures, declarations, etc., which may then be used by the current design. For instance, *work.int_math.all* enables all functions contained in the package *int_math* (integer math), which is found in the work library. These functions describe the operation of the "+" and "−" operators used in the up and down counter logic descriptions. The package *rtlpkg* contains the definition of the Global Synchronous Set (gss) statement used on line 27.

The second part of a complete design unit is the Architecture section. In this section is where we find the description of the behavior of the black box defined in the Entity section. Associated with the Architecture statement are begin and end statements. Line 16 declares the Architecture name, behave, for the following statements which describe the functionality of the Entity *clk_genv*. The Entity and Architecture sections are separated because VHDL allows multiple architectures to be defined for a given entity. Only one architecture can be associated with an entity in a given design. This feature allows multiple versions of an architecture to be saved in a library. The Configuration statement is used to select a specific architecture (see Reference 3).

```
01: entity clk_genv is
02: port(clock, ld_cnt, rstn      :in bit;
03:      incnt                    :in bit_vector(3 downto 0);
04:      count                    :buffer bit_vector(3 downto 0);
05:      endeqcnt, endltcnt, clk_out :buffer bit;
06:      endeq0, cnteq0           :buffer bit);

07: attribute part_name of clk_genv : entity is "c335";
08: attribute pin_numbers of clk_genv : entity is
09:  "clock:3 ld_cnt:1 rstn:7 clk_out:17 "
10:& "incnt(3):6 incnt(2):5 incnt(1):4 incnt(0):2 "
11:& "count(3):19 count(2):15 count(1):28 count(0):26 "
12:& "endeqcnt:25 endltcnt:23 endeq0:24 cnteq0:27";
13: end clk_genv;

14: use work.int_math.all;
15: use work.rtlpkg.all;

16: architecture behave of clk_genv is
.
.
.
22: begin
.
.
.
60: end behave;
```

Figure 4. VHDL Desing and I/O Declarations

Internal Signal Declarations

In both Abel−HDL and VHDL, internal signals (nodes) may be defined. These signals do not connect directly to an input or an output pin and may result from buried logic or may simply represent a wire to transfer data. Both languages are similar in that the signal name and type are declared.

As can be seen in *Figure 5*, lines 10 and 11 of the Abel−HDL code declare the signals RST_CTR and ENDCNT3...0. Each is defined as type node, as opposed to type pin, which would mean that the signal

```
10:  rst_ctr                              node istype 'reg_d';
11:  endcnt3,endcnt2,endcnt1,endcnt0      node;
12:  incnt     = [incnt3,incnt2,incnt1,incnt0];
13:  count     = [count3,count2,count1,count0];
14:  endcnt    = [endcnt3,endcnt2,endcnt1,endcnt0];
15:  outputs = [count,endeqcnt,endltcnt,endeq0,cnteq0,clk_out];
16:  cnt_state = [rst_ctr,clk_out];
```

Figure 5. Abel−HDL Signal Declarations

```
17:  signal endcnt : bit_vector(3 downto 0);
18:  signal cnt_state : bit_vector(0 to 1);
```

Figure 6. VHDL Signal Declarations

would be connected to an I/O pin. An optional signal attribute may be used with the keyword istype to define the signal's characteristics more explicitly. Lines 12 – 16 show the groupings of signals into sets. Defining sets allows a group of signals to be referenced by one name. Any operation performed on the set name will be performed on each member of the set.

VHDL allows signal names that represent, among others, bit vectors such as the one shown in line 17 in *Figure 6*. Here ENDCNT is equivalent to [ENDCNT(3), ENDCNT(2), ENDCNT(1), ENDCNT(0)]. As seen earlier in the entity declaration, INCNT has been defined as a port (I/O pins) and is a 4-bit vector similar to ENDCNT. Individual signals cannot be declared and grouped into sets as with Abel−HDL. Rather, groups are declared initially as bit vectors. The individual members of the set can then be operated upon separately or as a group (line 58 of the VHDL source code shows cnt_state(1) being assigned to the output CLK_OUT).

State-Machine State Definitions

The section where state register assignments are declared is very similar for Abel−HDL and VHDL. Both languages require assignment of a constant value to a name which gets compared to the current value of the state bits (cnt_state) in the actual state machine implementation (IF-THEN-ELSE, CASE-WHEN). Shown in this document is one method of designing a state machine. Both Abel−HDL and VHDL allow a variety of ways in which to create a

```
17:  reset      = [0,0];
18:  cnt_up     = [1,1];
19:  cnt_down   = [1,0];
```

Figure 7. Abel−HDL State Machine Definition

state machine. For large state machines, a more compact implementation might be with a Truth-Table in which inputs and outputs are described in a tabular form. This method is more compact but some may find it less "readable" than other methods. Both languages also support Mealy, Moore, and one-hot (one register per state) state machine implementations. *Figure 7* shows the Abel−HDL state assignment code.

Figure 8 shows the VHDL code needed to make state assignments. Note here the increased verbosity relative to Abel−HDL. This, again, is due to the fact that VHDL is a more general-purpose language and that statements must be more explicit. Here, a constant is defined to be a certain type (bit_vector) and then is assigned a initial value using the ":=" operator. Note that VHDL's usage of the ":=" operator is different than its meaning as a registered signal assignment operator in Abel−HDL.

Combinatorial Logic Equations

Both Abel−HDL and VHDL allow combinatorial Boolean) logic equations. As *Figures 9* and *10* show, the syntax is quite similar. Combinatorial statements in Abel−HDL are signified by a "=" operator. Use of the *istype reg* attribute in the signal declaration section and/or use of the appropriate explicit

```
19:  constant reset : bit_vector(0 to 1) := "00";
20:  constant cnt_up : bit_vector(0 to 1) := "11";
21:  constant cnt_down : bit_vector(0 to 1) := "10";
```

Figure 8. VHDL State Machine Definition

```
20: equations
21:    endeqcnt = ((endcnt.fb - 1) == count.fb);
22:    endltcnt = (endcnt.fb < count.fb);
23:    endeq0 = (endcnt.fb == 0);
24:  cnteq0 = (count.fb == 1);
25:    outputs.sp = !rstn;
26:    count.clk = clock;
```

Figure 9. Abel−HDL Combinatorial Logic Equations

```
23: endeqcnt <= '1' when (count = (endcnt-1)) else '0';
24: endltcnt <= '1' when (endcnt < count) else '0';
25: endeq0   <= '1' when (endcnt = "0000") else '0';
26: cnteq0   <= '1' when (count = "0001") else '0';
27: gss <= NOT(rstn);
```

Figure 10. VHDL Combinatorial Logic Equations

signal extensions (.c, .q, .d, etc...) and the ":=" operator signifies registered logic. In VHDL, the syntax for both a combinatorial and registered signal assignment is the same, "<=". The difference being determined by where in the code the statement appears. It is treated as a registered signal only if the signal assignment statement occurs inside of a clocked process. (See References 1 and 3 for full explanation of processes.)

Since the CY7C335 used in this example, like many other devices, contains special features, such as global or individual resets, presets, or OEs, there needs to be a means to expressly access them. Lines 25 and 27 of the Abel−HDL and VHDL, respectively, show how to access the available global synchronous preset of the device. In the Abel−HDL code we have defined a set to be all of the output signals (see Line 14). By simply using the .sp extension, !rstn is assigned to the global synchronous preset signal. Note that it is not necessary to define a set. Since in this device the preset is global, by assigning !rstn to the preset of one output register, all are automatically connected. Line 26 assigns the signal clock to the counter registers.

In VHDL, since extensions are not allowed in signal assignments, special functions are created and placed in a standard package which access these specific device features. In this case, the gss (global synchronous set) function is found in the package *rtlpkg*. When using a function (or other statement) found in a package, a use statement must be added (Line 15) so that the contents of the package are visible to this design. This requirement may seem cumbersome on the surface, it in fact represents one of the most powerful advantages of using VHDL. It gives the ability to save and reuse commonly used statements in standard or user defined packages which can then be accessed by any design. These packages can in turn be placed in libraries which can be organized by function, project, etc.

Input Register Logic Definition

The CY7C335 was chosen for this example because it can be used to illustrate a variety of features that may be found in other programmable logic devices. In this section, we are making use of the registered inputs. The inputs are defined as INCNT(3:0). Once registered, these signals become ENDCNT(3:0) and are assigned to internal nodes in the device.

In the Abel_HDL code shown in *Figure 11*, Line 27 assigns the signal LD_CNT to be the clock for the ENDCNT registers. Line 28 uses the registered assignment operator, ":=", to create ENDCNT from INCNT.

```
27:  endcnt.clk = ld_cnt;
28:  endcnt := incnt;
```

Figure 11. Abel−HDL Input Register Definition

In VHDL, to create registered logic, a process must be used. This highlights a key concept in VHDL, the notion of concurrent vs. sequential statements. All concurrent statements are continuously and simultaneously evaluated, creating combinatorial logic. Sequential statements, as the name implies, are evaluated in order. The IF-THEN-ELSE construct is a classic example of a sequential statement. In VHDL, only statements found within a process are sequential. The processes themselves are concurrent and are continuously evaluated at the same time as all other statements between the begin and end of an Architecture section. A process is awakened (i.e., evaluated) when a change occurs in the value of a signal that is sensitive to that process. Sensitive signals are defined by the use of a sensitivity list or a wait statement at the beginning of the process. In our example we have used the wait statement to awaken a process when the clock signal for the associated logic sees a rising edge (Lines 28−31 of *Figure 12*). Here, the *in_reg* process is evaluated when a rising edge occurs on LD_CNT. Inside the process statements are evaluated sequentially. In this process there is a single statement which, when evaluated, causes the value of INCNT to be transferred to ENDCNT. This effectively creates a register clocked by LD_CNT.

State-Machine Description

The description of the simple three-state state machine (see *Figure 2*) is shown next for both Abel−HDL and VHDL in *Figures 13* and *14*, respectively. In general, both languages allow a variety of state machine definition methods. Included are truth tables, IF-THEN-ELSE statements, and CASE-WHEN statements. Both implementations require a state machine name declaration, clock declaration, and state descriptions.

```
28: in_reg : process begin
29:     wait until (ld_cnt = '1');
30:      endcnt <= incnt;
31: end process;
```

Figure 12. VHDL Input Register Definition

```
29:  cnt_state.clk = clock;
30:  state_diagram cnt_state
31:       state reset:
32:                 count := 0000;
33:            if (endeq0) then reset;
34:            else cnt_up;

35:       state cnt_up:
36:                 count := (count.fb + 1);
37:            if (endltcnt # endeq0) then reset;
38:            else if (endeqcnt) then cnt_down;
39:            else cnt_up;

40:       state cnt_down:
41:                 count := (count.fb - 1);
42:            if (endltcnt # endeq0) then reset;
43:            else if (cnteq0) then cnt_up;
44:            else cnt_down;
```

Figure 13. Abel−HDL State Machine Equations

```
32: counter : process begin
33:  wait until (clock = '1');
34:       case cnt_state is
35:       when reset =>
36:         count <= "0000";
37:       if (endeq0 = '0') then cnt_state <= cnt_up;
38:       else cnt_state <= reset;
39:       end if;

40:       when cnt_up =>
41:      count <= count +1;
42:       if (endltcnt='1' OR endeq0='1') then
43:      cnt_state <= reset;
44:       elsif (endeqcnt = '1') then cnt_state <= cnt_down;
45:       else cnt_state <= cnt_up;
46:       end if;

47:       when cnt_down =>
48:      count <= count - 1;
49:       if (endltcnt='1' OR endeq0='1') then
50:      cnt_state <= reset;
51:       elsif (cnteq0 = '1') then cnt_state <= cnt_up;
52:       else cnt_state <= cnt_down;
53:       end if;

54:       when others =>
55:      count <= "0000";
56:      cnt_state <= reset;

57:       end case;
58: end process;
59:  clk_out <= cnt_state(1);
```

Figure 14. VHDL State Machine Equations

In the Abel−HDL code of *Figure 13*, line 29 declares the signal clock to be the clock source for the state registers. Line 30 declares the following state descriptions to be for the state machine cnt_state. Lines 31−44 are the descriptions for each of the three states. Within each state description can be found signal assignments and IF-THEN-ELSE statements defining the conditional next-state assignments.

Similar to Abel−HDL, the VHDL code of *Figure 14* contains a clock declaration on line 33 (the wait until statement implies clock is the register clock in this process), and a state machine declaration on line 34 (the case statement defines cnt_state as the state machine under evaluation). Lines 35−57 contain the individual state descriptions. Lines 32 and 58 declare the beginning and end of the process called counter. This explicit declaration of the beginning and end of processes is necessary because of the VHDL distinction between concurrent statements and sequential (within a process) statements. Note the addition of the "when others" statement. This is added to insure that the state machine can recover from an invalid (undefined) state. Lastly, line 59 as-

signs the value of the state bit cnt_state(1) to the output signal CLK_OUT. Had this statement been placed inside the process it would have been treated as a sequential statement and, therefore, would be registered. This would have caused a registered, or pipelined, delay to be added to CLK_OUT.

Summary

In summary, as design languages, Abel−HDL and IEEE−VHDL are really quite similar in complexity. Many experienced Abel−HDL users may perceive VHDL to be unnecessarily complicated. This may be true if one is limited to the smaller playing field covered by Abel−HDL. VHDL, on the other hand, covers a broader set of applications, such as full system-level description and simulation. The extra verbosity is minimal when compared to the extra functionality provided. For instance, VHDL allows true source code simulation, an easy migration path to ASICs (standard, portable language), and design with different types such as integers, enumerated types, records, etc. It also is truly device independent. For instance, falling edge clocks and XORs can be described behaviorally in VHDL whereas in Abel−HDL, the target device must be declared and specific fuses programmed to make use of these special features.

References

1. Cypress Semiconductor, Warp2 *User's Manual*, 1993.
2. Data I/O, *Abel Design Software User Manual*, 1990.
3. S. Mazor and P. Langstraat, *A Guide To VHDL*, Kluwer Academic Publishers, Norwell, MA, 1992.
4. Douglas L. Perry, VHDL Second Edition, McGraw-Hill Series, Computer Engineering, 1994.

Appendix A. VHDL Design File for Prog. Clock Generator

```
01: entity clk_genv is
02: port(clock, ld_cnt, rstn   :in bit;
03:      incnt              :in bit_vector(3 downto 0);
04:      count              :buffer bit_vector(3 downto 0);
05:      endeqcnt, endltcnt, clk_out  :buffer bit;
06:      endeq0, cnteq0     :buffer bit);
07: attribute part_name of clk_genv : entity is "c335";
08: attribute pin_numbers of clk_genv : entity is
09:    "clock:3 ld_cnt:1 rstn:7 clk_out:17 "
10: & "incnt(3):6 incnt(2):5 incnt(1):4 incnt(0):2 "
11: & "count(3):19 count(2):15 count(1):28 count(0):26 "
12: & "endeqcnt:25 endltcnt:23 endeq0:24 cnteq0:27";
13: end clk_genv;

14: use work.int_math.all;
15: use work.rtlpkg.all;

16: architecture behave of clk_genv is

17:  signal endcnt : bit_vector(3 downto 0);
18:  signal cnt_state : bit_vector(0 to 1);
19:  constant reset : bit_vector(0 to 1) := "00";
20:  constant cnt_up : bit_vector(0 to 1) := "11";
21:  constant cnt_down : bit_vector(0 to 1) := "10";

22: begin

23:  endeqcnt <= '1' when (count = (endcnt-1)) else '0';
24:  endltcnt <= '1' when (endcnt < count) else '0';
25:  endeq0   <= '1' when (endcnt = "0000") else '0';
26:  cnteq0   <= '1' when (count = "0001") else '0';
27:  gss <= NOT(rstn);

28: in_reg : process begin
29:     wait until (ld_cnt = '1');
30:        endcnt <= incnt;
31: end process;

32: counter : process begin
33:    wait until (clock = '1');
34:    case cnt_state is
35:    when reset =>
36:        count <= "0000";
37:      if (endeq0 = '0') then cnt_state <= cnt_up;
38:      else cnt_state <= reset;
39:      end if;
```

Appendix A. VHDL Design File for Prog. Clock Generator (continued)

```
40:     when cnt_up =>
41:       count <= count +1;
42:        if (endltcnt='1' OR endeq0='1') then
43:       cnt_state <= reset;
44:        elsif (endeqcnt = '1') then cnt_state <= cnt_down;
45:        else cnt_state <= cnt_up;
46:        end if;

47:  when cnt_down =>
48:       count <= count - 1;
49:        if (endltcnt='1' OR endeq0='1') then
50:       cnt_state <= reset;
51:        elsif (cnteq0 = '1') then cnt_state <= cnt_up;
52:        else cnt_state <= cnt_down;
53:        end if;

54:    when others =>
55:       count <= "0000";
56:       cnt_state <= reset;

57:     end case;
58: end process;
59:  clk_out <= cnt_state(1);
60: end behave;
```

Appendix B. Abel−HDL Design File for Prog. Clock Generator

```
01: module clk_gena;

02: declarations
03: device 'p335';

    "Inputs
04:  clock, ld_cnt, rstn        pin 3,1,7;
05:  incnt3, incnt2, incnt1, incnt     pin 6,5,4,2;

    "Outputs
06:  endeqcnt, endltcnt       pin 25,23 istype 'com';
07:  endeq0, cnteq0         pin 24,27 istype 'com';
08:  clk_out           pin 17 istype 'reg_d';
09:  count3, count2, count1, count0    pin 19,15,28,26;

10:  rst_ctr           node istype 'reg_d';
11:  endcnt3,endcnt2,endcnt1,endcnt    node;

12:  incnt    = [incnt3,incnt2,incnt1,incnt0];
13:  count    = [count3,count2,count1,count0];
14:  endcnt   = [endcnt3,endcnt2,endcnt1,endcnt0];
15:  outputs = [count,endeqcnt,endltcnt,endeq0,cnteq0,clk_out];

16:  cnt_state = [rst_ctr,clk_out];
17:  reset     = [0,0];
18:  cnt_up    = [1,1];
19:  cnt_down  = [1,0];

20: equations

21:  endeqcnt = ((endcnt.fb - 1) == count.fb);
22:  endltcnt = (endcnt.fb < count.fb);
23:  endeq0 = (endcnt.fb == 0);
24:  cnteq0 = (count.fb == 1);
25:  outputs.sp = !rstn;
26:  count.clk = clock;
27:  endcnt.clk = ld_cnt;
28:  endcnt := incnt;

29:  cnt_state.clk = clock;
30:  state_diagram cnt_state
31:       state reset:
32:                 count := 0000;
33:            if (endeq0) then reset;
34:            else cnt_up;
```

Appendix B. Abel−HDL Design File for Prog. Clock Generator (continued)

```
35:       state cnt_up:
36:                 count := (count.fb + 1);
37:            if (endltcnt # endeq0) then reset;
38:            else if (endeqcnt) then cnt_down;
39:            else cnt_up;

40:       state cnt_down:
41:                 count := (count.fb - 1);
42:            if (endltcnt # endeq0) then reset;
43:            else if (cnteq0) then cnt_up;
44:            else cnt_down;
45: end clk_gena;
```

The FLASH370™ Family Of CPLDs and Designing with *Warp2*™

This application note covers the following topics: (1) a general discussion of complex programmable logic devices (CPLDs), (2) an overview of the CY7C370 family of CPLDs, and (3) using the *Warp2* VHDL Compiler for the CY7C370 family.

Overview of CPLDs

CPLDs extend the concept of the PLD to a higher level of integration to improve system performance, use less board space, improve reliability, and reduce cost. Instead of making the PLD bigger with more input terms and product terms, a CPLD architecture is composed of multiple PLDs or logic blocks (LABs) connected together with a programmable interconnect matrix (PIM). Multiple Logic Array Blocks (LABs) provide comparable speed to a PLD because the basic propagation path is through one LAB and each LABs product term array is comparable to a PLD array. Multiple LABs provide the higher integration. The number of LABs in a CPLD is typically between 2 for the smaller CPLDs and 16 for the larger ones. In addition to LABs interconnected by the PIM, are the input/output macrocells and the dedicated input macrocells. *Figures 1* and *2* show the CPLD generic block diagram and the logic block diagram respectively.

The architectural components of the LAB are: (1) the product term array, (2) the product term allocator, and (3) the macrocell. The product term array is the same in the CPLD as in the PLD except that the inputs into the array can now also come from the PIM. The product term allocator is a new concept in the CPLD where product terms are not fixed to a macrocell with its associated input/output pin but

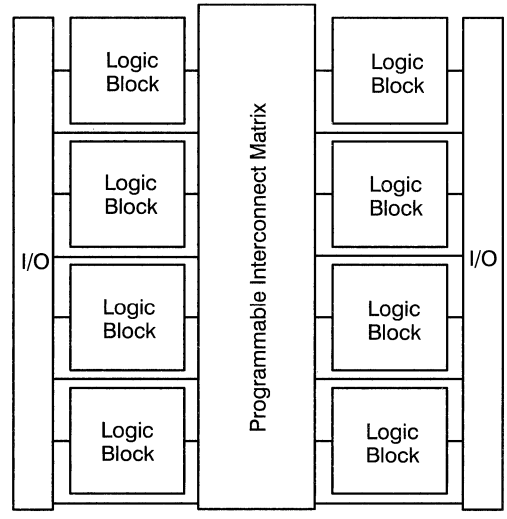


Figure 1. Generic Block Diagram

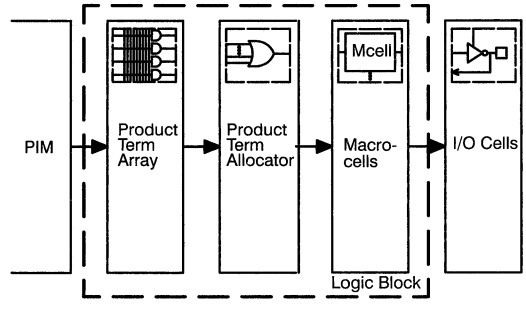


Figure 2. Logic Block Diagram

can be routed to different macrocells depending on where they are needed. The result is a more efficient allocation of product terms and higher integration. Implementation of the product term allocator varies across CPLD vendors which is more fully discussed in the section describing the features of the CY7C370 family.

The macrocell accepts the single output of the product term allocator which is the ORing of a variable number of product terms. In some macrocells this input feeds into a two input XOR gate with the other input potentially carrying the Q feedback. This configures the D flip flop to a T flip flop which can provide an improvement in capacity for certain designs such as counters. After the XOR gate, the macrocell is configurable as registered, combinatorial, and in some cases latched. There are two kinds of macrocells which are input/output dedicated and buried. Dedicated macrocells output to the input/output macrocell and also provide feedback into the product term array. Buried macrocells only provide feedback into the product term array.

The function of the PIM is to distribute the needed fraction of the total available resources, all outputs from the LAB and possibly also dedicated inputs and inputs/outputs, to the appropriate LAB. There are two common methods of PIM implementation: array based interconnect and mux based interconnect.

Figure 3 shows the data path of communication between two LABs using the array based interconnect. In the array based interconnect, each output of the LAB can potentially connect to any number of PIM input terms through a memory element. Each PIM input term is assigned to a specific LAB and functions as an input term into the LABs product term array. In this example only four PIM input terms are shown two going to LAB1 and two going to LAB2. There is a sense amp per input term to detect the logic level, buffer the signal, and drive it into the LAB. The true and complement of the PIM signal feed into the product term array (not shown in the figure). Since every LAB output can connect to any PIM input, the interconnect is considered 100 percent routable. It never limits the ability of the device to fit logic. A macrocell output can connect to one or multiple PIM input terms. The major drawback from using a memory element as an interconnect is the slower propagation delay than the muxed based interconnect.

Figure 4 shows the data path of communication between two LABs using the muxed based interconnect. In the muxed based interconnect a mux chooses one of a number of potential PIM input terms into the LAB. The PIM input terms differ from the array based interconnect in that they are output from a 1 of n (where "n" is the number of inputs of the mux) mux instead of the output of a wired nor memory array. The inputs into the muxes are all the outputs of the LABs as well as dedicated inputs and input/output pins. *Figure 3* shows two PIM input terms output from two 4-to-1 muxes. In this example, macrocell 2 from LAB1 and macrocell 2 from LAB2 both show 2 chances to route into the muxes with other inputs having only 1 chance The wider the mux (the number of inputs into the mux) the more likely all desired inputs into each LAB will be successfully routed and the more chances each signal gets to route into a LAB. The disadvantage of larger muxes is a larger slower propagation delay through the PIM and increased die size. Implementations of mux-based interconnect vary in the size of the mux.

Features of the FLASH370 CPLDs

The FLASH370 family of CPLDs offers densities from 2 to 16 LABs . *Figure 5* shows the block diagram of the CY7C374/5 with 8 LABs. The even numbers of the family (372,374,376) bury half of the macrocells for maximum integration with the same pinout as the (371,373,375) respectively. The 377 does not have a corresponding equivalent pinout with buried macrocells. *Table 1* shows the family members offered.

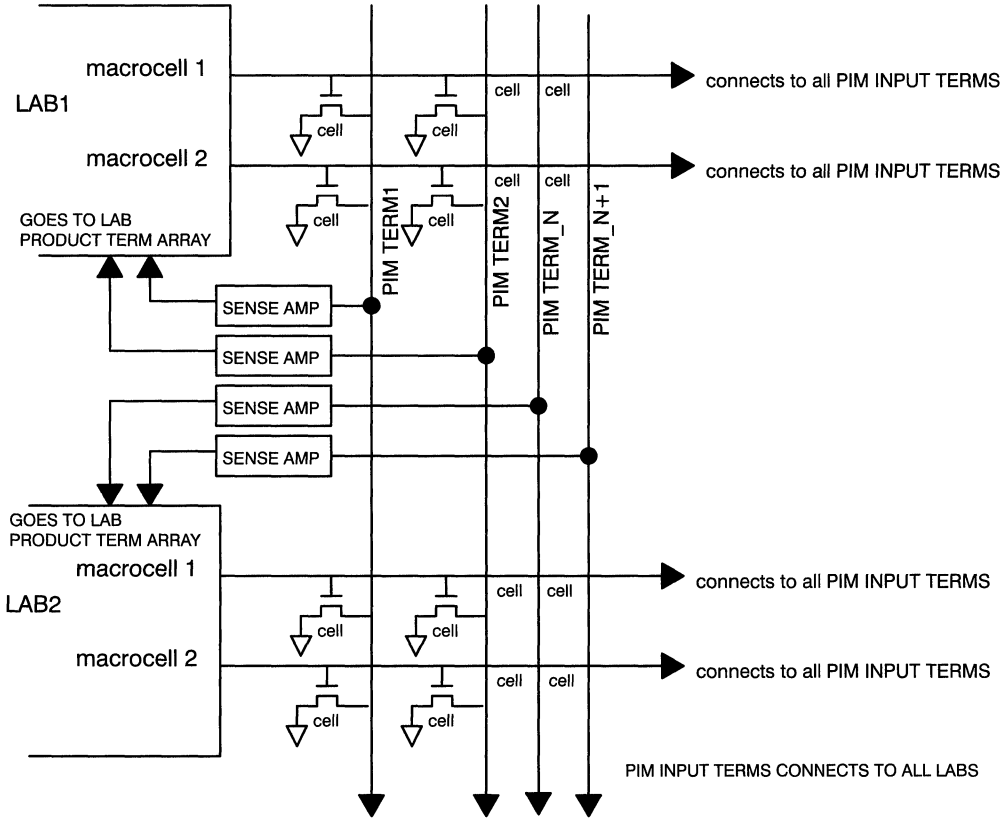


Figure 3. Array-Based Interconnect

Table 1. FLASH370 Family Members

Feature	CY7C371	CY7C372/3	CY7C374/5	CY7C376/7
Macrocells	32	64	128	256
Dedicated Inputs	6	6	6	6
I/O pins	32	32/64	64/128	128/256
Dedicated Inputs Usable as Clocks	2	2/4	4/4	4/4
Speed (t_{PD})	8.5 ns	10 ns	12 ns	15 ns
Primary Packages	44-PLCC	44/84-PLCC 100-TQFP	84-PLCC 100/160-TQFP	160-TQFP 289-BGA

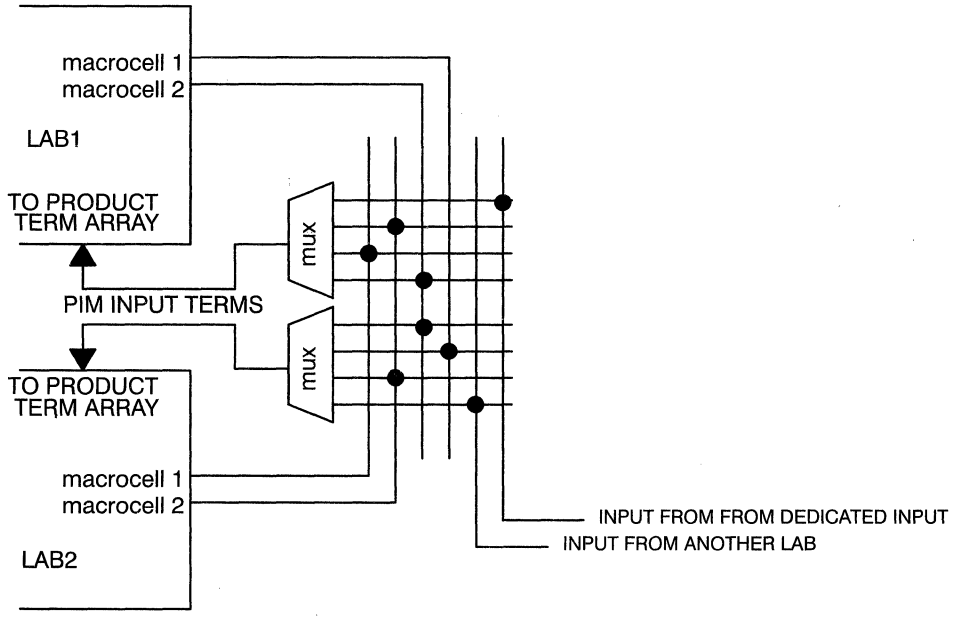


Figure 4. Mux Based Interconnect

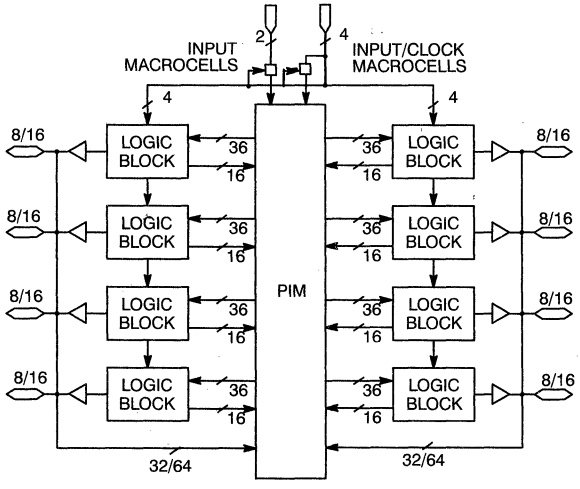


Figure 5. CY7C374/5 Block Diagram

Figures 6 and *7* show the product term array, product term allocator, macrocells, and input/output macrocells for the CY7C370 family. Each LAB features 36 inputs, which can adequately handle 32-bit operations plus control signals with one pass through the LAB. The product term array features the true and complement polarities of each PIM output signal for a total of 72 inputs. 80 standard product terms are provided to the product term allocator which allocates from 0 to 16 product terms to each of the 16 macrocells. Additionally, 6 special product terms are also generated in the product term array. They are an asynchronous preset, asynchronous reset, and two groups of 2 bank output enable product terms.

The output macrocell (*Figure 8*) provides a selection of four output controlling options: (1) control from one output enable, (2) control from a second output enable, (3) permanently enabled, or (4) permanently disabled. Each LAB contains 4 output enable product terms, 2 for the upper 8 macrocells and 2 for the lower 8 macrocells.

The state macrocell (*Figure 8*) contains options to register, latch, or send data through combinatorially. For the input/output macrocell there is an additional output polarity mux to improve capacity before the signal goes to the input/output macrocell. For buried macrocells there is an additional mux which can configure the state register as an input register. If the buried macrocell is configured as an input, zero product terms will be allocated from the array. In *Figure 8* architecture bit C7 can choose the feedback from the input/output pin as the input into the register instead of from the product term array.

There is one asynchronous preset and reset product term for each LAB. There are polarity muxes for the clocks, preset and reset. Each macrocell can choose among two clocking options for the CY7C371/372 and four clocking options for the CY7C373/374/375/376/377. All macrocells in a LAB receive the same polarity of the clock, set and reset. Polarities are configurable per LAB. *Figure 8* shows the input/output macrocell and input/output plus buried macrocell.

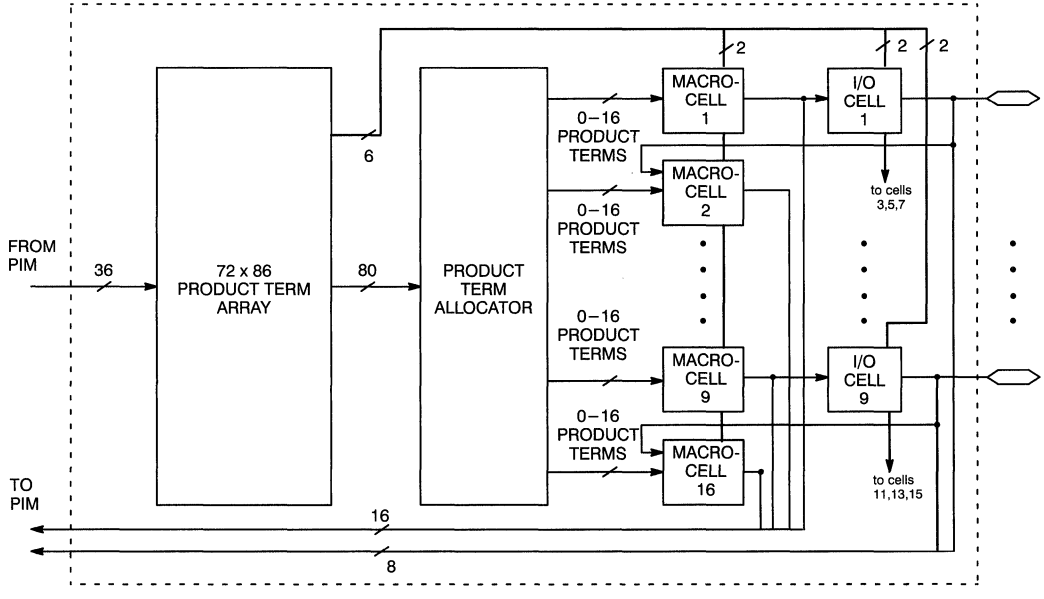


Figure 6. Logic Block for CY7C372, CY7C374, and CY7C376 (Register Intensive)

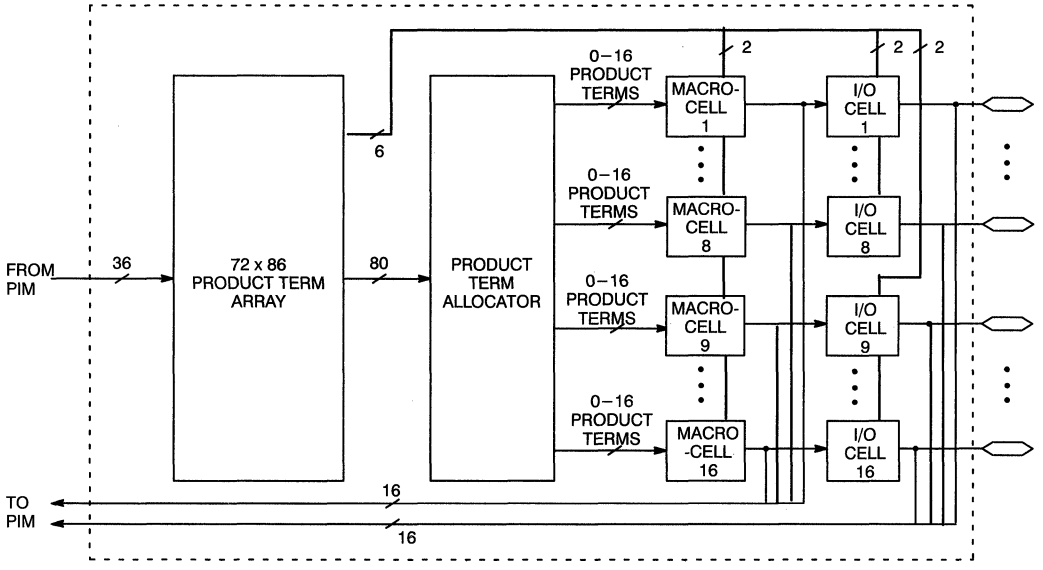


Figure 7. Logic Block for CY7C371, CY7C373, CY7C375, and CY7C377 (I/O Intensive)

Figure 9 and *10* show the input/clock and input macrocells. The input macrocell provides the flexibility to let the input enter combinatorially, latched, single registered, or double registered (for maximum metastability performance). For the 371/372 there are two input/clocks pins and four input pins. For the CY7C373/374/375/376/377 there are four input/clock pins and two input pins. For added flexibility, each clock can be configurable for either positive or negative polarity.

In order to fully understand the operation of the CY7C370 product term allocator, two important aspects of product term allocator design need to be introduced: product term steering and product term sharing. Steering refers to the assignment of a product term resource to a macrocell. In the traditional PLD there is no steering flexibility. Each macrocell has assigned product terms that can only be used by that macrocell. In many designs each macrocell requires a different number of product terms putting an emphasis on the ability to allocate product terms individually on an as needed basis. Product term sharing refers to a product term being used by multiple macrocells. The logic equations for different macrocells sometimes contain the same minterm. Instead of generating this same minterm multiple times, it is generated on only one product term and shared across macrocells, thereby improving capacity.

Figure 11 is a conceptual representation of the CY7C370 product term allocator. The product term allocator functions like a segmented OR array by ORing from 0 to 16 product terms for each macrocell. Product terms can be steered and shared on an individual basis. This architecture has several advantages over other implementations that steer product terms away from one macrocell to serve another.

Figure 12 is a conceptual representation of the MACH™ product term allocator. It shows no ability to share product terms across macrocells. Each cluster of four product terms can route to only one macrocell. The product terms are routed in groups of four which is a much higher granularity of product term allocation and not as efficient.

To demonstrate this inefficiency, consider a macrocell that needs five product terms to implement its logic. Two product term clusters with a total of eight

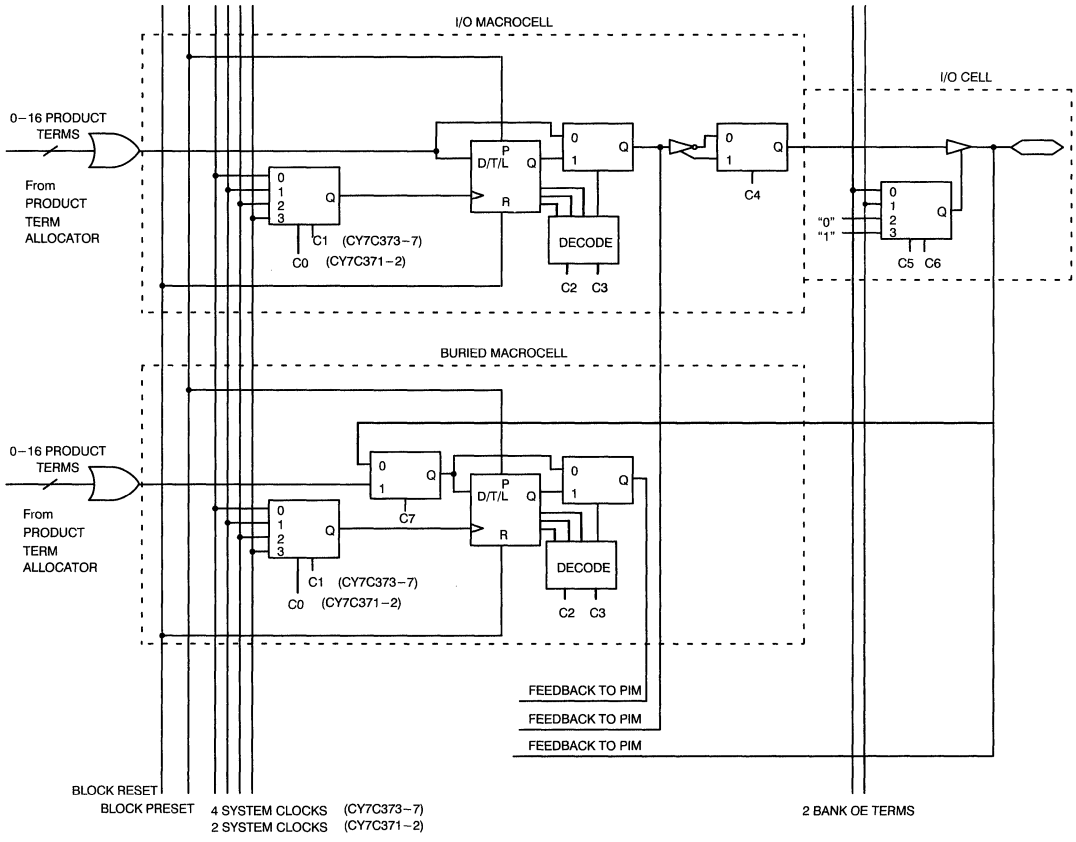


Figure 8. I/O and Buried Macrocells

available product terms are needed. This wastes the resources of three product terms from the borrowed cluster since these product terms can not be re-routed to another macrocell.

The MAX7000™ product term allocator representation (*Figure 13*) shows the use of expander terms. Expander terms allow two passes through the array which can produce very high capacity. These expanders are also shared among all product terms in the LAB. The problem with using the expanders is in the additional propagation delay of two passes through the array. This complicates the timing model and links the performance of the device to the use of the expander product terms. As with the MACH product term allocator, the MAX7000 allocator also has five product term clusters. It therefore suffers from the same problem of product term wasting when more than one cluster is routed to a macrocell.

The CY7C370 product term allocator provides the most effective method of steering and sharing product terms. The propagation of signals through the product term allocator is independent of the number of product terms allocated to each macrocell. Additionally the flexibility of this product term allocator, with the PIM, enables a change in the design without a modification to the external pinout of the device. There is no need for input and output switch matrices, which add extra delay and degrade performance.

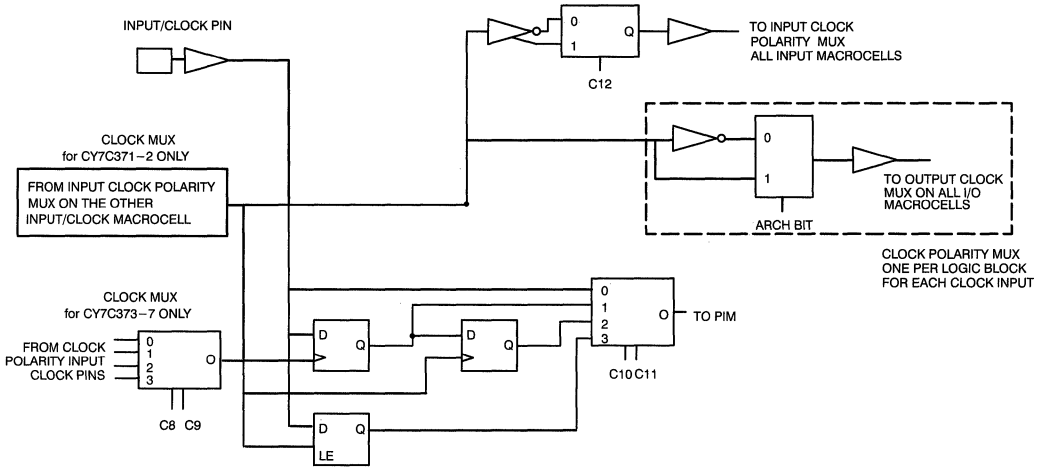


Figure 9. Input/Clock Pins

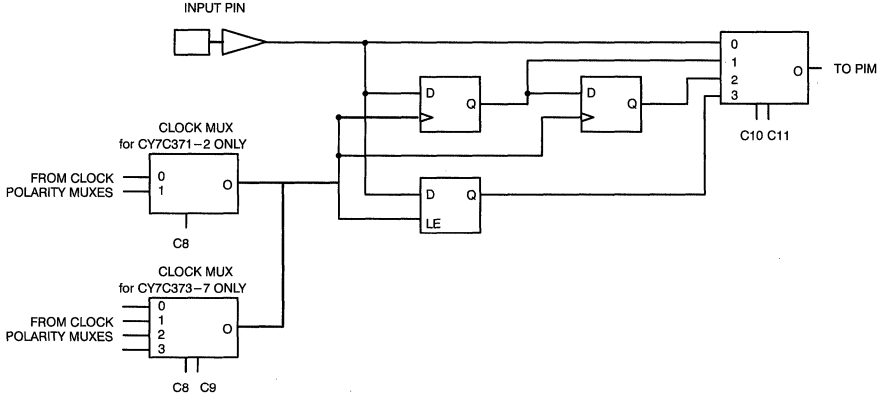


Figure 10. Input Pins

The timing model of the CY7C370 family is far simpler than for other CPLD solutions for two reasons. First, all input signals into the LAB pass through the PIM. This includes all input/outputs, feedbacks from macrocell outputs, and dedicated inputs. Secondly, the propagation time through the product term allocator is independent of the number of product terms allocated to a macrocell. As a result, there are no expander delays, no dedicated versus input/output pin delays, no penalties for using up to 16 product terms, or no delay penalties for steering and or sharing product terms. The CY7C370 family of products provides timing as predictable as PLDs like the 22V10.

The PIM in the CY7C370 was designed to approach the 100 percent routability of the array based interconnect but not made so wide that performance and die size suffered.

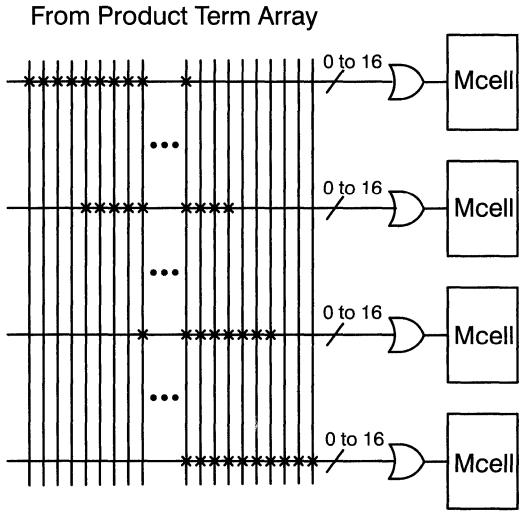


Figure 11. CY7C370 Product Term Allocator Representation

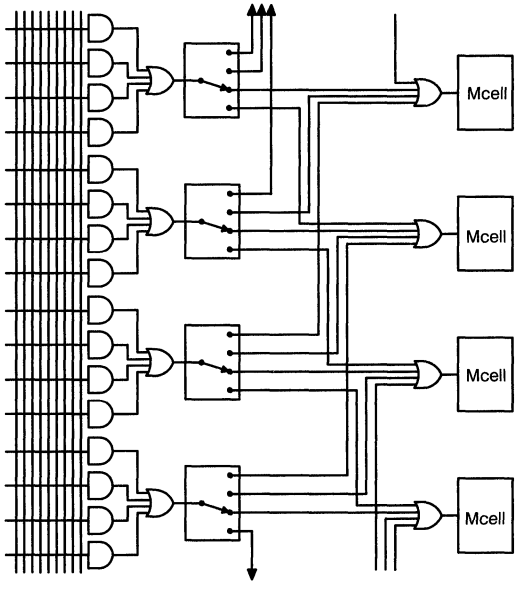


Figure 12. MACH Product Term Allocator Representation

Using *Warp* ™ to Design with the CY7C370

Development software is extremely important for ease of use and efficiency of resource allocation when designing with CPLDs. Cypress offers two software packages that will fully support the CY7C370 family of products as well as all other PLDs, FPGAs, and state machine PROMs. *Warp2* provides full VHDL language support which is becoming the industry standard for describing hardware design. A functional simulator is also provided. *Warp3* additionally includes schematic capture and exact timing simulation capability.

The simplified timing model of the CY7C370 often makes exact timing simulation unnecessary because performance can be predicted directly from the datasheet. Therefore the functional simulator of *Warp2* may be a cost effective design solution. With *Warp* no manual intervention for fitting the designs into the devices are necessary. In addition to *Warp*, customers also have third party support from a variety of vendors.

Warp products take in VHDL designs and automatically fit them into the chosen device. The following

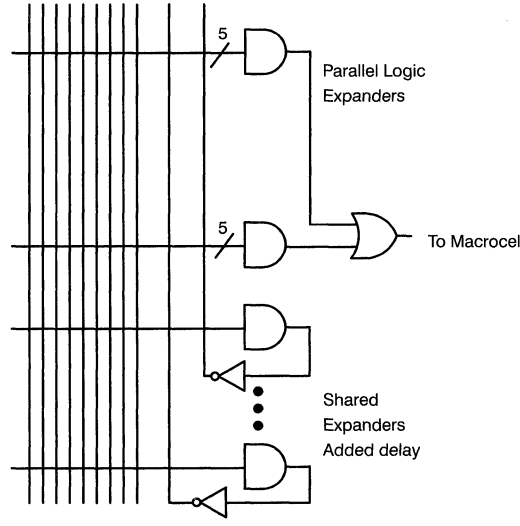


Figure 13. MAX Product Term Allocator Representation

section explains how to exploit the special features of the CY7C370 with VHDL. A thorough treatment of VHDL constructs is found in the *Warp2* Reference Manual. Topics covered here are: (1) using the single/double registered options for the dedicated inputs, and registering signals from the IO pins, (2) using the clock polarity mux feature, (3) describing registered versus latched versus combinatorial outputs, (4) using the output enable feature, (5) using the asynchronous preset/reset feature, and (6) Using the buried registers as for the (372/4/6).

To register the dedicated inputs one or two signals must be defined to represent the additional nodes for one and two registers respectively. Appendix A demonstrates how to use single and double registered inputs for a 4 bit loadable counter. In *proc2*, RESET1 and RESET2 are the outputs of the first and second registers. It requires 2 passes through *proc2* to activate RESET2. Signal RESET2 is then used in *proc1* to perform the reset. *Proc2* additionally registers the data to be loaded with the statement `regin <=temp.dat`. The signal REGIN is then used in process *Proc1* to load the counter with the statement `temp.cnt <= regin`. If the same clock is used for the inputs as for the state registers, then the statements in process *proc2* could be incorporated into *proc1* and only one process is needed. The assignment of the entity output pins is handled by the instantiation of the *bufoe* component (called in the statement `use work.rtlpkg.all`), which takes the signal TEMP.CNT as input and transfers it to the output (in this case called COUNT) when the output enable control (called OUTEN) is HIGH. Registering the inputs from the input/output pins is better suited for the 372/374/376 members of the family since the signal does not need to go through the PIM and logic block.

Clocking on the falling instead of the rising edge of the clock is simply done by changing the statement `wait until (clk = '1')` to `wait until (clk = '0')`. Events occurring on the rising and falling edge of a clock can be incorporated into the same design by defining a separate process for the event, provided that sufficient logic blocks are available.

VHDL describing combinatorial and registered outputs is identical to other part implementations as with the CY7C370. The registered equations must be inserted inside a process and after a `wait until clock=` statement.

Appendix B shows an example of how to implement the combinatorial macrocell option with maximum usage of output enable flexibility for the CY7C371. A total of eight different input signals control the output enable functionality. The entire function is handled by the *bufoe* component where the input into the buffer is the external input pin. No signals are necessary.

The latch option is unique to the CY7C370 family. Appendix C shows an example of how to latch a signal using the IF-THEN-ELSE construct. In this example the signal is latched when the clock is HIGH by setting the signal value to itself with the statements `signala <= signala` and `signalb <= signalb`. When the clock is LOW the path is combinatorial and the signal value gets the input. This is handled in the code `if clk='0' then signala <= inputa; signalb <= inputb`. Two signals are defined, SIGNALA and SIGNALB, to latch the data when the clock is in the right polarity (in this case HIGH).

Appendix D shows the full registered configuration. As in Appendix C, the signals SIGNALA and SIGNALB are defined and the function of the register is defined within a process. On the rising edge of the clock, SIGNALA gets INPUTA and SIGNALB gets INPUTB.

Appendix E uses latches for the output enable control. Signals need to be generated from the array and are passed as the output enable parameter into the *triout* component. This function behaves similarly to the *bufoe* but does not include the feedback parameter.

Appendix F shows how to use the buried registers to implement the least significant bits in a counter. A bit vector signal is defined to represent all the register states. Those states that are needed as outputs are assigned to the entity output pins outside of the process with the statement `count (0 to 11) <= fullcnt (4 to 15)`. If output enable control is desired then this last statement is omitted and

the signal to output assignment is handled with the *bufoe* component.

Appendix G is the same as Appendix F except that the registers are reset asynchronously. The format of the process is much different from Appendix F but functions exactly the same except for the asynchronous instead of synchronous reset. The process uses a "sensitivity list" that includes all the parameters that will activate the process. The synchronous part of the process is initiated by the statement `clk'event and clk='1'` instead of `wait until clk='1'`. The asynchronous preset/reset is similar to other Cypress PLDs except for the additional polarity mux feature that enables active HIGH or LOW. To specify clock polarity, the VHDL construct for active HIGH is `if reset = '1' then` and for active LOW is `if reset = '0' then`.

Appendix A. inregcnt

```
-- The bufoe port map parameters are:
-- bufoe port map(signal going to the input of the tristateable buffer,
--     tristate control signal,
--     the output signal that is the entity output pin,
--     the feedback signal from the entity input/output pin)
-- In this example the last entry is "open" meaning no feedback.

USE work.bv_math.all;          -- necessary for inc_bv();
USE work.rtlpkg.all;           -- necessary for bufoe

ENTITY inregcnt IS
       PORT   (clk, clkin, reset, load, outen: IN bit;
         count: INOUT x01z_VECTOR(0 TO 3));
END inregcnt;

ARCHITECTURE behavior OF inregcnt IS
TYPE bufRec IS                 -- record for bufoe
       RECORD                  -- inputs and feedback
            cnt: bit_vector(0 TO 3);
            dat: bit_vector(0 TO 3);
END RECORD;
SIGNAL temp: bufRec;
SIGNAL regin: bit_vector(0 to 3);-- for registering input loaded data
SIGNAL reset1, reset2:bit;      -- for registering the reset input
CONSTANT counterSize: integer := 3;
BEGIN
g1:    FOR i IN 0 TO counterSize GENERATE
            bx: bufoe PORT MAP(temp.cnt(i), outen, count(i), temp.dat(i));
       END GENERATE;
proc1: PROCESS
       BEGIN
            WAIT UNTIL (clk = '1');
            IF reset2 = '1' THEN -- uses the double registered signal
                 temp.cnt <= "0000";
            ELSIF load = '1' THEN
                 temp.cnt <= regin; -- uses the single registered signal
            ELSE
                 temp.cnt <= inc_bv(temp.cnt);  -- increment bit vector
            END IF;
       END PROCESS;
-- Proc2 single registers the load operation and double registers the reset
-- operation.  Note the two clkin's are needed for the double register.
proc2: PROCESS
       BEGIN
         WAIT UNTIL (clkin = '1');
          regin <= temp.dat; --single register for data load
          reset1 <= reset; --single register the reset signal
          reset2 <= reset1;--double register the reset signal
       END PROCESS;
END behavior;
```

Appendix B. usecomb

```
--uses the full functionality of the oe features of the 371.
--macrocell is in combinatorial mode

USE work.rtlpkg.all;

ENTITY usecomb IS
       PORT (outen1, outen2, outen3, outen4, outen5, outen6, outen7,
            outen8; IN bit; inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END usecomb;

ARCHITECTURE behavior OF usecomb IS
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(inputa(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(inputa(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(inputa(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(inputa(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(inputb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(inputb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(inputb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(inputb(i), outen8, outb(i+6), open);
       END GENERATE;
END behavior;
```

Appendix C. uselatch

```
--uses the full functionality of the oe features of the 371.
--macrocell in latched mode

USE work.rtlpkg.all;

ENTITY uselatch IS
       PORT   (clk, outen1, outen2, outen3, outen4, outen5, outen6, outen7,
            outen8: IN bit;
            inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END uselatch;

ARCHITECTURE behavior OF uselatch IS
SIGNAL signala, signalb: bit_vector(0 to 1);
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(signala(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(signala(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(signala(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(signala(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(signalb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(signalb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(signalb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(signalb(i), outen8, outb(i+6), open);
       END GENERATE;--the clk input is an active low latch enable
--the if then construct must be within a process.
PROCESS
  BEGIN
    IF clk='0' then
      signala <= inputa;
      signalb <= inputb;
    ELSE
      signala <= signala;
      signalb <= signalb;
    END IF;
  END PROCESS;
END behavior;
```

Appendix D. usereg

```
--macrocell in registered mode

ENTITY usereg IS
       PORT   (clk, outen1, outen2, outen3, outen4, outen5, outen6, outen7,
            outen8: IN bit; inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END usereg;

ARCHITECTURE behavior OF usereg IS
SIGNAL signala, signalb: bit_vector(0 to 1);
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(signala(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(signala(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(signala(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(signala(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(signalb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(signalb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(signalb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(signalb(i), outen8, outb(i+6), open);
       END GENERATE;   --the clk input is a rising edge triggered clock for
                       --the register
--the wait until construct must be within a process.
PROCESS
  BEGIN
    WAIT UNTIL clk='1';
    signala <= inputa;
    signalb <= inputb;
  END PROCESS;
END behavior;
```

Appendix E. uselatch2

```
--This file shows the use of the triout component to perform the
--output enable function.

--COMPONENT triout
-- port (
--    x: IN bit; -- input to buffer
--    oe: IN bit; -- output enable
--    y: OUT bit); -- output
--END component

--The oe control is a function of the dedicated inputs and is latch
--controlled.

USE work.rtlpkg.all;          --to instantiate triout component

ENTITY uselatch2 IS
  PORT (clk1, clk2, in_oe1, in_oe2: IN bit;
        inputa, inputb: IN bit_vector(0 to 1);
         outa,outb: INOUT x01z_vector(0 to 7));
END uselatch2;

ARCHITECTURE behavior OF uselatch2 IS
SIGNAL signala, signalb: bit_vector(0 to 1);
SIGNAL sig_en1, sig_en2, sig_en3, sig_en4: bit;
BEGIN
g1:  FOR i IN 0 TO 1 GENERATE
       bx1: triout PORT MAP(signala(i), sig_en1, outa(i));
       bx2: triout PORT MAP(signala(i), sig_en2, outa(i+2));
       bx3: triout PORT MAP(signala(i), sig_en3, outa(i+4));
       bx4: triout PORT MAP(signala(i), sig_en4, outa(i+6));
       bx5: triout PORT MAP(signalb(i), sig_en1, outa(i));
       bx6: triout PORT MAP(signalb(i), sig_en2, outa(i+2));
       bx7: triout PORT MAP(signalb(i), sig_en3, outa(i+4));
       bx8: triout PORT MAP(signalb(i), sig_en4, outa(i+6));
     END GENERATE;

--The clock latches the data when high and is combinatorial when low
oecontrol: PROCESS
           BEGIN
             IF clk1= '0' then
               sig_en1 <= not(in_oe2) and not(in_oe1);
               sig_en2 <= not(in_oe2) and in_oe1;
               sig_en3 <= in_oe2 and not(in_oe1);
               sig_en4 <= in_oe2 and in_oe1;
             ELSE
               sig_en1 <= sig_en1;
               sig_en2 <= sig_en2;
               sig_en3 <= sig_en3;
```

Appendix E. uselatch2 (continued)

```
              sig_en4 <= sig_en4;
            END IF;
          END PROCESS;

latch: PROCESS
          BEGIN
            IF clk2= '0' then
              signala <= inputa;
              signalb <= inputb;
            ELSE
              signala <= signala;
              signalb <= signalb;
            END IF;
          END PROCESS;
END behavior;
```

Appendix F. buriedreg

```
-- The purpose of this example is to show how to use the
-- buried registers to create a 16 bit counter.  The 12
-- most significant bits are assigned to i/o registers
-- and the 4 least significant bits go to the buried registers.

USE work.bv_math.all;                -- necessary for inc_bv();

ENTITY buriedreg IS
       PORT   (clk, reset: IN BIT;
            count: INOUT bit_vector(0 TO 11));
END buriedreg;

ARCHITECTURE behavior OF buriedreg IS
SIGNAL fullcnt : bit_vector(0 to 15);
BEGIN
PROCESS
     BEGIN
       WAIT UNTIL (clk = '1');
            IF reset = '1' THEN        -- synchronous reset
                 FOR i IN 0 TO 15 LOOP
                      fullcnt(i) <= '0';
                 END LOOP;
            ELSE
                 fullcnt <= inc_bv(fullcnt);
            END IF;
     END PROCESS;
count(0 to 11) <= fullcnt(4 to 15);
END behavior;
```

Appendix G. buriedreg2

```
-- The purpose of this example is to show how to use the
-- buried registers to create a 16 bit counter.  The 12
-- most significant bits are assigned to i/o registers
-- and the 4 least significant bits go to the buried registers.
-- This example also demonstrates how to do an asynchronous reset.

USE work.bv_math.all;                -- necessary for inc_bv();

ENTITY buriedreg2 IS
  PORT   (clk, reset: IN BIT;
    count: inout bit_vector(0 TO 11));
END buriedreg2;

ARCHITECTURE behavior OF buriedreg2 IS
SIGNAL fullcnt : bit_vector(0 to 15);
BEGIN
   PROCESS(clk,reset)--sensitivity list
        BEGIN
          IF reset = '1' THEN
            fullcnt <= x"0000";-- asychronous reset, the x stands for hex
          ELSIF (clk'event and clk = '1') then
            fullcnt <= inc_bv(fullcnt);-- synchronous count
          END IF;
        END process;
   count(0 to 11) <= fullcnt(4 to 15);  -- assigns signals to entity outputs
                                        -- and defines buried registers
END behavior;
```

Implementing a Reframe Controller for the CY7B933 HOTLink™ Receiver in a CY7C371 CPLD

Introduction

This application note describes a reframe controller for the Cypress CY7B933 HOTLink Receiver. The primary function of the controller is to monitor the Receive Violation Symbol output, RVS, from the CY7B933 in order to detect framing errors and, under the correct conditions, assert the Reframe signal, RF, to the CY7B933. The controller function is designed with a state machine, a few counters, and some decode logic. All are implemented in VHDL and fit into a Cypress CY7C371 32-macrocell FLASH CPLD. The exact implementation in this application note makes several assumptions about the next-higher-level controller that may not be universally applicable. However, the source code for the design is provided in Appendix A at the end of this application note so that modification and customization for other interfaces is easily possible.

Why Reframing is Necessary

The CY7B923 and CY7B933 HOTLink Transmitter and Receiver are a pair of chips for high-speed point-to-point serial data communication. The CY7B923 is the transmitter, and the CY7B933 is the receiver. The CY7B923 takes in an 8-bit byte at a frequency between 16 and 33 MHz, encodes it into 10 bits, does a parallel-to-serial conversion, and then transmits the serial data at ten times the byte-rate clock (about 160 to 330 Megabits per second (Mbps)). At the other end of the link, the CY7B933 receives the serial data, does a serial-to-parallel conversion, unencodes the data back into its original form, and shifts the 8-bit parallel data out at the same byte-rate clock frequency used by the transmitter. (Note: the chips can also transmit and receive 10 bits of unencoded data. For a full description of the encoding and decoding functions, see the CY7B923/933 datasheet.)

The key element in the data-and-clock-recovery circuit on the receiver is the PLL, i.e., phase-locked loop, on the chip. It is triggered by the transitions in the incoming data stream, and it is used to both separate the data stream into individual bits and to generate the byte-rate clock going out of the chip. Once the PLL achieves synchronization with the incoming serial data stream and is receiving bits properly, the receiver must be given a reference point that will set the byte boundaries in the bit stream. This is done by the framing circuitry. Whenever the receiver's RF (reframe) input is asserted, the receiver's framing logic will check the incoming bit stream for the special pattern that defines a byte boundary. When this is found, the receiver logic sets a reference point and simply counts bits from that point on so it can properly execute the serial-to-parallel conversion on subsequent byte boundaries, and properly align the byte-rate clock rising edge.

Thus, framing is always required when the receiver begins receiving data for the first time, either at power-up or after switching from one transmitter source to another. Periodic reframing may also be necessary, however, due to other conditions. If the PLL goes out of lock—that is, if it loses its synchronization with the incoming serial bit stream for any reason, the recovered data will be erroneous and the

framing boundary information will be lost. Once the PLL gets back into synchronization with the incoming bit stream, it will be necessary to force the receiver to reframe in order to re-establish the proper byte boundary point.

Using RVS to Know When to Reframe

The PLL out-of-lock condition can be detected by the behavior of the RVS output of the CY7B933 receiver. The CY7B933 asserts RVS when it detects an error in the bit stream. Infrequent errors, due to random noise in the environment or attenuation by the transmission medium, for example, are expected and do not necessarily mean that the PLL is out of lock or that the data needs to be reframed. Too many errors in too short a time indicates that the PLL has lost lock and reframing is necessary. The benchmark chosen in this controller is 16 errors occurring in a period of 64 bytes. If the controller counts RVS asserted 16 times during a 64-byte period, it will assume the PLL has lost lock and will assert RF to the receiver to force it to reframe.

The 16-out-of-64 benchmark is somewhat arbitrarily chosen, but it is justified by the fact that when the PLL is in lock, you would normally expect to see significantly fewer errors. The fact that 16 out of 64 is the criteria used does not mean that 15 out of 64, or 14 out of 64, etc., are acceptable error rates and that the PLL is not out of lock in these cases as well. But, it is fairly certain that if the PLL does go out of lock, you will get at least 16 errors in 64 byte-times, very quickly. Furthermore, there are counters inside the HOTLink Receiver that detect this same condition (16 errors in a 64-byte period) and when this detection occurs inside the CY7B933, it forces the PLL to re-lock onto the serial input data stream. Even if the PLL is out of lock, if fewer than 16 errors are detected in a 64-byte period, the PLL will not be forced to re-synchronize with the data stream and will stay out-of-lock until that condition is detected. Therefore, for consistency, the same criteria was selected for the reframe controller.

Additional Functionality of the Reframe Controller

The reframe controller itself interfaces to a higher-level controller that controls the entire receiver system. That higher-level controller can force the reframe controller to initiate framing in the CY7B933, regardless of any errors. There are two ways to do this. The first is with the DO_REFRAME signal, which the higher-level controller asserts when it wants the reframe controller to go through the same procedure it goes through to initiate framing when an out-of-lock condition occurs. If the reframe controller sees this signal asserted, it acts just like it had detected an out-of-lock condition. The other way the higher-level controller can force a reframe is by asserting its FORCE_RF output. This simply forces the reframe controller's RF output HIGH and does not cause the internal logic or state machine to change. The reframe controller's RF output will stay asserted as long as its FORCE_RF input remains asserted.

The higher-level controller will normally assert DO_REFRAME on power-up or when the transmitter source is switched on in order to find the initial byte-boundary, as described above. The FORCE_RF signal could be used for any reason depending on specific system requirements. The most likely reason to use it is to force multibyte framing. When the receiver does multibyte framing, instead of looking for a single byte-boundary-indicating character, the receiver looks to detect two of these special characters within any four-byte sequence. This is a more reliable way of finding the byte boundary, simply because it causes the framing circuitry to verify its first find with another one. This may be useful in particularly noisy environments. To cause the receiver to do multibyte framing, you must assert its RF input for 2048 consecutive cycles; this is something the reframe controller would not ordinarily do. The higher-level controller can cause this to happen by asserting FORCE_RF to the reframe controller for 2048 cycles, thus causing its RF output to be asserted for the same length of time.

The reframe controller also implements a basic handshake with the higher-level controller to make sure the two controllers' operations stay consistent

after forced reframes. Whenever the higher-level controller uses the DO_REFRAME signal to force the reframe controller to initiate framing, it will keep that signal asserted until the reframe controller asserts RFDONE_HS. This signal from the reframe controller indicates that the receiver has finished its reframing. The higher-level controller will then assert RFDONE_ACK, which acknowledges receipt of RFDONE_HS, and both the reframe controller and the higher-level controller will return to the state it normally returns to following a reframe.

In addition to the operations described above, the reframe controller also provides a decoding function. When the HOTLink Receiver detects a data error and asserts RVS, it also puts the code for the type of error on its eight data outputs, D7 – D0. The reframe controller decodes these signals and asserts one of two outputs, UNDEF_CHAR or RDISP_ERR, depending on the exact type of error decoded. The two types of errors are an undefined-character error and a running-disparity error. A running-disparity error means that the character received had too many consecutive 1s or 0s to be a valid byte of data (the purpose of the eight-bit-to-ten-bit encoding mentioned earlier is to encode the data in such a way as to minimize the imbalance of 1s and 0s in the bit stream). If the reframe controller detects the code for a running-disparity error, it will assert the RDISP_ERR output. If the received character has the correct running disparity but is not a valid code for any character, then it is an undefined-character error, and the reframe controller will assert the UNDEF_CHAR output instead.

Design and Implementation

The out-of-lock detection, RF control, higher-level controller interface, and error-type decoding are implemented with a simple state machine, a few internal counters, and some decoding logic, and it is all fit into a 32-macrocell CY7C371 FLASH CPLD (for more information on this CPLD, please refer to other application notes in the PLD section of this handbook and to the CY7C371 datasheet). The design was done in VHDL and compiled with Cypress' *Warp*™ PLD/FPGA design tool. The receiver system, the reframe controller's interface, and the details of the design of the internal state machine, counters, and logic are described in detail in the rest of this section.

Receiver System

Figure 1 shows where this reframe controller fits into the overall system. The CY7B933 receiver connects (through a physical connector) to the actual transmission medium, which can be either twisted pair, coaxial cable, or fiberoptic cable. The reframe controller interfaces to the receiver, and it also interfaces to the higher-level system controller.

Controller Interface

The complete set of reframe controller inputs and outputs is shown in *Figure 2*, and their source or destination, polarity, and functionality are described below.

Inputs

RF_ENABLE. Overall enable. It comes from a higher-level controller. When asserted (HIGH), reframe controller is enabled. When deasserted, reframe controller is disabled and does not operate.

CLK. Clock signal to the reframe controller that comes from the recovered byte-rate-clock output, RCLK, of the CY7B933, and is also used in the rest of the system as the system clock.

RESET. Resets the state machine and the internal counters and status registers (HIGH = asserted).

RVS. Received Violation Symbol. It comes from RVS output of the CY7B933 (HIGH = asserted).

FORCE_RF. When asserted, this forces the RF output to also be asserted regardless of other conditions. It comes from a higher-level controller (HIGH = asserted).

DO_REFRAME. When asserted, it causes internal state machine to initiate framing in the receiver just as if it had detected an out-of-lock condition. It comes from higher-level controller (HIGH = asserted).

RFDONE_ACK. Handshake signal from the higher-level controller acknowledging that it received confirmation that the reframe controller completed

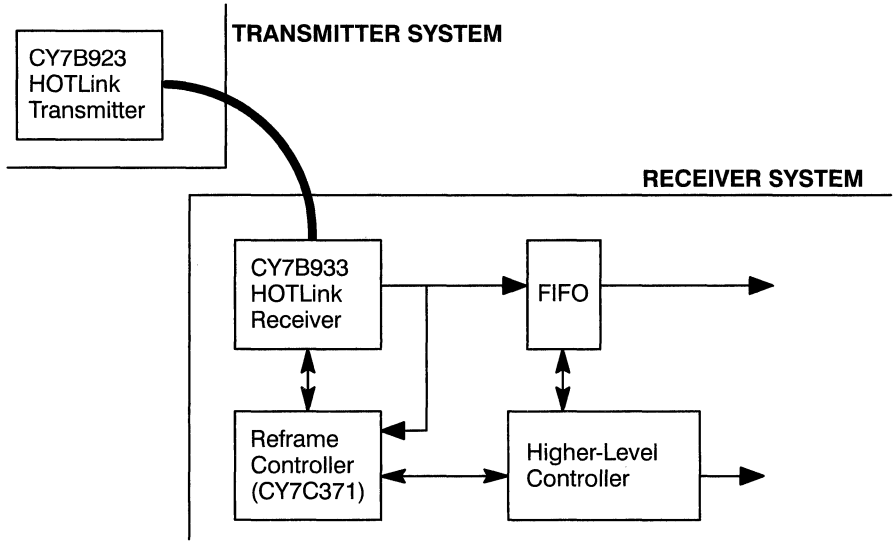


Figure 1. Block Diagram

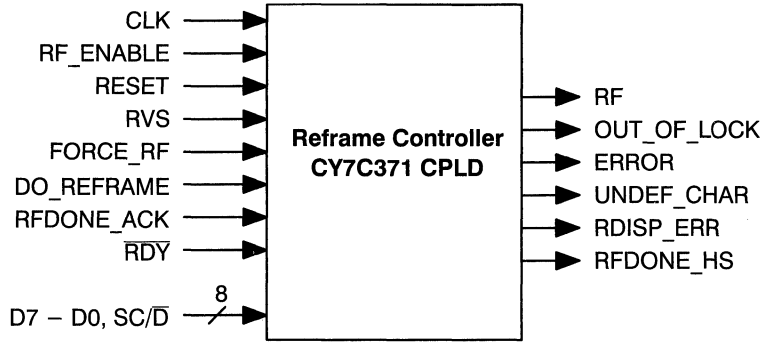


Figure 2. Controller Inputs and Outputs

the framing procedure. The handshake is only done when the framing was triggered by the DO_REFRAME signal, not by an out-of-lock condition (HIGH = asserted).

$\overline{\text{RDY}}$. Ready signal. It comes from the CY7B933 $\overline{\text{RDY}}$ output and indicates to the reframe controller that the receiver has completed the reframe operation (LOW = asserted).

D[7:0], SC/$\overline{\text{D}}$. Eight-bit data byte and control/data indicator bit from the CY7B933 receiver. The information on these lines can be decoded during a receive violation to determine the error type.

Outputs

RF. Reframe output. It goes to the RF input of the CY7B933 receiver and causes the HOTLink Re-

ceiver to begin a framing operation on the incoming data stream (HIGH = asserted).

RFDONE_HS. This is the handshake signal to the higher-level controller telling it that the reframe it requested with the DO_REFRAME signal has been completed (HIGH = asserted).

OUT_OF_LOCK. This signal indicates that the HOTLink Receiver's PLL has gone out of lock with the incoming serial bit stream. This is inferred by counting sixteen or more RVS assertions in a single 64-byte period. Once asserted, it remains asserted until the PLL regains lock and reframing has been accomplished (HIGH = asserted).

ERROR. When asserted (HIGH) it indicates to the higher-level controller that an error of some type (as indicated by the RVS signal from the receiver) has occurred.

UNDEF_CHAR. This is an undefined-character-error signal, one of two types of errors that can be decoded from the D7 – D0, SC/$\overline{D}$ inputs during receive violations. This signal is only valid when the ERROR output is also asserted, and it can only be asserted when RDISP_ERR is deasserted (HIGH = asserted).

RDISP_ERR. Running-disparity-error signal. This is the other of the two types of errors that can be decoded from the D7 – D0, SC/$\overline{D}$ inputs during data-receive violations. This signal is only valid when the ERROR output is also asserted, and it can only be asserted when UNDEF_CHAR is deasserted (HIGH = asserted.)

Counters

The primary function of the controller, which is to detect the out-of-lock condition by monitoring RVS and initiate a reframe when necessary, is implemented through the use of two counters. The VHDL for this function is shown in *Figure 3*. The first counter, rcvdbyts_count, is a seven-bit counter that counts the number of bytes received (0 to 64) and the second counter, error_count, is a five-bit counter that counts the number of times that RVS is asserted. If error_count reaches 16 before rcvdbyts_count reaches 64, then the out-of-lock condition will be declared. If rcvdbyts_count reaches 64 before error_count reaches 16, then fewer than 16 errors occurred in the given 64-byte window and out-of-lock is not declared. If rcvdbyts_count reaches 64 before error_count reaches 16, both rcvdbyts_count and error_count are set back to zero and a new 64-byte window begins. If the out-of-lock condition is declared (error_count = 16 and rcvdbyts_count $\leq$ 64), then the out-of-lock flip-flop is set to HIGH and a reframe operation is initiated. The out-of-lock flip-flop stays HIGH until the receiver successfully reframes. At that point, the out-of-lock flip-flop is set back to LOW and the search for the out-of-lock condition is started again.

State Machine

The state machine is described by the diagram in *Figure 4*, and the VHDL code that implements it is shown in *Figure 5*.

IDLE state

The normal, quiescent state of the state machine, and the state it enters upon reset, is IDLE. In this state, the RF output is deasserted and the state machine waits for either a DO_REFRAME input from the outside or for the counters to set the out-of-lock flip-flop. If neither of these conditions occur, the state machine simply stays in the IDLE state. Once either one of these conditions occurs, the state machine must initiate a reframe, so it will go to the START_REFRAME state.

START_REFRAME state

In the START_REFRAME state, RF is asserted, and the state machine unconditionally transitions to the COUNT_2_CLOCKS state.

COUNT_2_CLOCKS state

The COUNT_2_CLOCKS state enables a two-bit counter to start counting incoming clock cycles. After two clock cycles have been counted, the state machine transitions to the LOOK_FOR_xRDY state. Two clock cycles must be counted before looking for the $\overline{RDY}$ signal from the outside because a total of three clocks must pass after RF is asserted until the value of $\overline{RDY}$ can be guaranteed valid (see the "HOTLink CY7B933 $\overline{RDY}$ Pin Description" ap-

```
-- relevant VHDL code for counter functions

use work.bv_math.all;
use work.int_math.all;

signal count2: bit_vector(0 to 1);          -- 2-bit counter
signal error_count: bit_vector(0 to 4);     -- 5-bit counter
signal rcvdbyts_count: bit_vector(0 to 6);  -- 7-bit counter

counters: process (CLK, RVS, reset, rcvdbyts_count, error_count,
       out_of_lock) begin

  if (clk'event and clk = '1') then

    if (reset = '1') then
      fb_out_of_lock    <= '0';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    elsif (error_count = "10000") then
      fb_out_of_lock    <= '1';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    elsif (rcvdbyts_count = "1000000") then
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    else
      rcvdbyts_count  <= rcvdbyts_count + 1;
      if (RVS = '1') then
        error_count <= error_count + 1;
      end if;
    end if;

    if (current_state = LOOK_FOR_xRDY) and (xRDY = '0') then
      fb_out_of_lock <= '0';
    end if;

    if (current_state = COUNT_2_CLOCKS) then
      count2 <= count2 + 1;
    else
      count2 <= "00";
    end if;

  end if;

end process; --counters
```

Figure 3. VHDL for Counter Functions

plication note for more details on this). One clock cycle passed during the START_REFRAME state, so the COUNT_2_CLOCKS state is used to count two more clock cycles to get to the requirement of three. RF is asserted throughout this state.

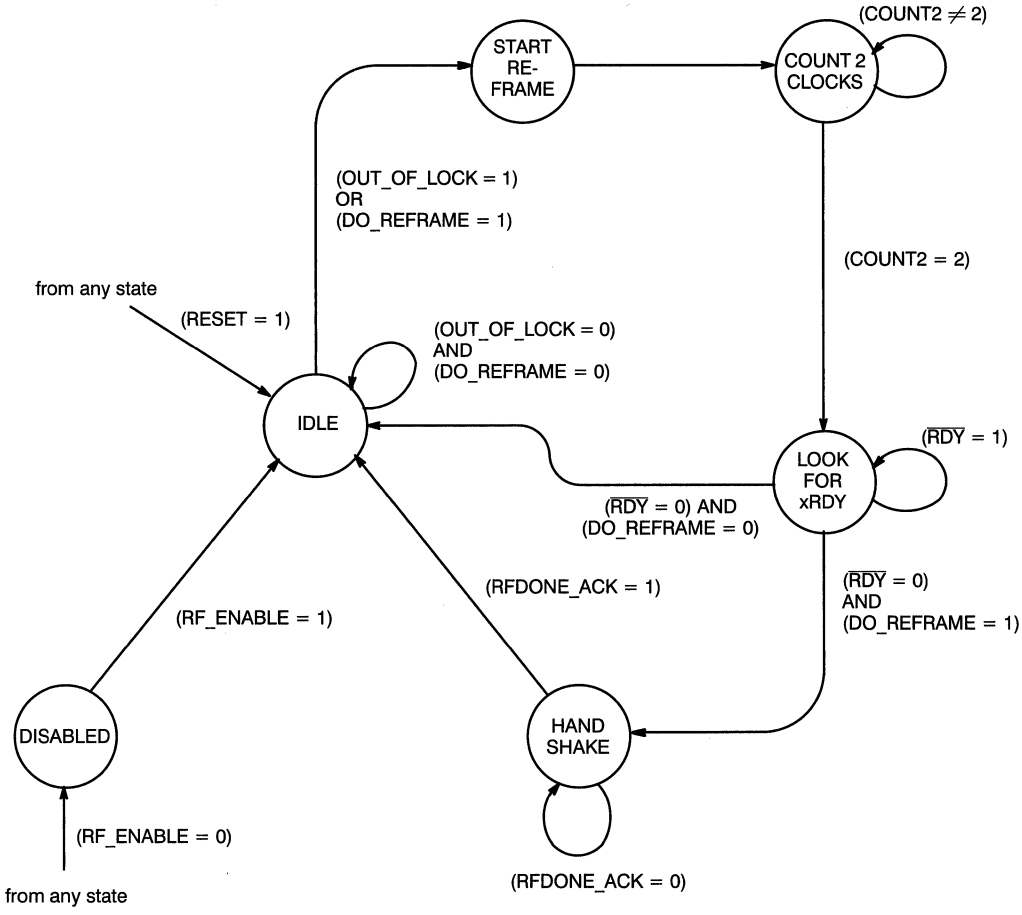


Figure 4. State Diagram

LOOK_FOR_xRDY state

On the fourth clock cycle from the start of RF, the value of $\overline{RDY}$ is guaranteed to be valid and the state machine, in the LOOK_FOR_xRDY state, continues to assert RF and waits until the HOTLink Receiver asserts $\overline{RDY}$. Once the receiver asserts RDY, it has successfully reframed and is ready to resume normal receiver operation. Thus, once an asserted $\overline{RDY}$ is detected in the LOOK_FOR_xRDY state, the state machine exits that state and goes back to the IDLE state. If the reframe was started by an out-of-lock detection, the transition back to the IDLE state is immediate; if the reframe was started by the DO_REFRAME input, then the state machine goes to the HANDSHAKE state first.

HANDSHAKE state

The HANDSHAKE state is used to make sure the reframe controller and the higher-level controller are consistent with each other. The only way this state will ever be entered is if the higher-level con-

```
-- Relevant VHDL code for state machine

subtype StateType is bit_vector(0 to 2);      -- State Type
constant       DISABLED: StateType := b"111"; -- State Defns.
constant           IDLE: StateType := b"000";
constant  START_REFRAME: StateType := b"001";
constant COUNT_2_CLOCKS: StateType := b"010";
constant  LOOK_FOR_xRDY: StateType := b"011";
constant      HANDSHAKE: StateType := b"100";
signal current_state, next_state : StateType; --State declaration

-- State Machine Description

if (RESET = '1') then
  next_state <= IDLE;
elsif (RF_ENABLE = '0') then
  next_state <= DISABLED;
else
  case current_state is

  when IDLE =>
    if (fb_OUT_OF_LOCK = '1') or (DO_REFRAME = '1') then
      next_state <= START_REFRAME;
    else
      next_state <= current_state;
    end if;

  when START_REFRAME =>
    next_state <= Count_2_Clocks;

  when COUNT_2_CLOCKS =>
    if (count2 = "10") then
      next_state <= LOOK_FOR_xRDY;
    else
      next_state <= current_state;
    end if;

  when LOOK_FOR_xRDY =>
    if (xRDY = '0') and (DO_REFRAME = '1') then
      next_state <= HANDSHAKE;
    elsif (xRDY = '0') and (DO_REFRAME = '0') then
      next_state <= IDLE;
    else
      next_state <= current_state;
    end if;
```

Figure 5. VHDL Code for State Machine

```
  when HANDSHAKE =>
    if (RFDONE_ACK = '1') then
      next_state <= IDLE;
    else
      next_state <= current_state;
    end if;

  when DISABLED =>
    if (RF_ENABLE = '0') then
      next_state <= current_state;
    else
      next_state <= IDLE;
    end if;

  end case;
end if;

if (clk'event and clk = '1') then
    current_state <= next_state;
end if;
```

Figure 5. VHDL Code for State Machine (continued)

troller initiated a reframe by asserting DO_REFRAME to the reframe controller. Once that reframe has been completed by the receiver, the reframe controller communicates this to the higher-level controller by asserting RFDONE_HS. Once the higher-level controller acknowledges this assertion and is ready to proceed with normal receiving operation, it will assert RFDONE_ACK as confirmation to the reframe controller. It will simultaneously deassert DO_REFRAME so that once the state machine goes back to the IDLE state, that input is deasserted and does not erroneously cause another immediate pass into the reframe procedure. Once the state machine detects the RFDONE_ACK assertion, it exits the HANDSHAKE state and returns to the IDLE state. The RF operation is deasserted throughout the HANDSHAKE state.

DISABLED state

There is one more state, the DISABLED state, which is treated separately. As long as RF_ENABLE, the overall controller enable, is asserted, the state machine will never enter this state. If RF_ENABLE gets deasserted, the state machine will transition to the DISABLED state no matter what state it was in, and it will stay there until RF_ENABLE is once again asserted. Once RF_ENABLE is reasserted, the state machine goes to the IDLE state and resumes normal operation.

It was mentioned previously that the out-of-lock flip-flop is set when the out-of-lock condition is detected, and it stays set until the reframe has been completed. The exact time when the OUT_OF_LOCK flip-flop gets cleared is at the rising clock edge when the state machine exits the LOOK_FOR_xRDY state. This is because that is the exact point where the receiver has signalled to the controller, with $\overline{\text{RDY}}$, that it has successfully competed the reframe.

Decode Logic

The error-decode logic is very straightforward, and the VHDL code for it is shown in *Figure 6*. The ERROR output is a registered version of the RVS input. The RDISP_ERR and UNDEF_CHAR outputs are decoded from the D7 – D0, SC/$\overline{\text{D}}$ inputs. These outputs are also registered.

When the receiver asserts RVS, it will also put a code for the error type on its eight data outputs. If this code is E4, E2, or E1 (hex), it indicates the error is a running-disparity error, (explained earlier), and the RDISP_ERR output is asserted. If it is any other hex code, the receiver has detected some kind of illegal or undefined character, and the UNDEF_CHAR output will be asserted instead. These outputs are mutually exclusive, i.e., if one is asserted, the other must be deasserted. However, it is only meaningful to decode the data outputs when an error condition is detected, so the ERROR signal must be examined by the higher-level controller as well. If ERROR is not asserted, the output from RDISP_ERR and UNDEF_CHAR is no longer valid.

VHDL, CY7C371 Utilization, and CY7C371 Speed Considerations

The complete VHDL description for this design is given in Appendix A. The full source code consists of the fragments shown throughout this application note along with the other code necessary to mesh it together, (process declarations, signal declarations, and package-entity declarations). As the fragments and complete source file show, VHDL is a very simple, efficient way for describing PLD designs. For example, the counter functions are simply bit vectors that are used in the manner: COUNT <= COUNT + 1. Upper limits for the counters, clearing functions, resets, and presets are all implemented with a few simple IF-THEN-ELSE statements. The entire state machine is implemented with a CASE statement and IF-THEN-ELSE statements that have a straightforward, natural, one-to-one correspondence with the bubble diagram shown in *Figure 4*. The entire set of decode logic is implemented in a single IF-THEN-ELSE clause. Furthermore, the VHDL code provided is easy to understand and can be very easily modified. For example, it can be modified to interface to different higher-level-controller interfaces than the one assumed in this application note, or it could be incorporated into the higher-level controller design, with that design consisting of other VHDL code and implemented in a larger FLASH370™ CPLD, a pASIC™ FPGA, or even a gate array.

```
-- relevant VHDL code for Decode Logic

if (clk'event and clk = '1') then

  if (RVS = '1') then
    ERROR <= '1';
    if (D = x"E4" or D = x"E2" or D = x"E1") then
      UNDEF_CHAR <= '0';
      RDISP_ERR  <= '1';
    else
      UNDEF_CHAR <= '1';
      RDISP_ERR  <= '0';
    end if;
  else
    ERROR <= '0';
    UNDEF_CHAR <= '0';
    RDISP_ERR  <= '0';
  end if;

end if;
```

Figure 6. VHDL Code for Decode Logic

This design used all 32 of the CY7C371's macrocells and 37 of its 38 I/O and input pins. It could have used fewer pins if necessary, by making the various counters be internal counters only. The outputs of the counters were brought out to output pins in this example, however, for easier simulation and debugging. The speed of the CY7C371 ranges from 66 MHz (with a 15-ns combinatorial propagation delay and a 12-ns clock-to-output time) to 143 MHz (with a 8.5-ns combinatorial propagation delay and a blazing 6-ns clock-to-output time). For this application, the maximum byte-rate clock of the CY7B933 is 33-MHz, and this and the corresponding set-up and hold times on the CY7B933 make the CY7C371-66 quite sufficient. The higher-level controller may have tighter timing requirements, but there is plenty of speed to be gained by going to the faster speed bins of the CY7C371. The design can, thus, easily meet much faster system timing requirements.

Conclusion

The serial data received by the CY7B933 needs to be framed, i.e., aligned to the proper byte boundaries. This must always be done when the serial communication first begins, and it must always be redone if the PLL loses lock on the incoming serial bit stream. This application note described a controller that will manage this operation and provided some guidelines for determining when the periodic reframing is necessary. It assumed a particular interface to a higher-level controller, but the design was done in VHDL, which is provided in the appendix, to make it very easily modifiable and adaptable to any other specific interface. The controller itself is implemented in a CY7C371 32-macrocell CPLD, which had sufficient resources and routability to implement this fairly substantial function. It was able to do this exceeding system speed requirements even in its slowest speed bin.

Appendix A. VHDL Description

```
-- Application Note
-- Using a CY7C371 as a HOTLink Reframe Controller
-- Cypress Semiconductor

use work.bv_math.all;
use work.int_math.all;

entity CONTROLLER is port (
    CLK, RVS, RESET, xRDY, DO_REFRAME, FORCE_RFOUT, RFDONE_ACK,
      RF_ENABLE : in bit;
    D           : in bit_vector(0 to 7);
    curr_st     : out bit_vector (0 to 2);
    rb_cntr     : out bit_vector (0 to 6);
    err_cntr    : out bit_vector (0 to 4);
    RF, RFDONE_HS, OUT_OF_LOCK, UNDEF_CHAR, RDISP_ERR, ERROR : out bit
);
end CONTROLLER;

architecture CNTRL933 of CONTROLLER is

    subtype StateType is bit_vector(0 to 2);        -- State Type
    constant       DISABLED: StateType := b"111";   -- State Definitions
    constant           IDLE: StateType := b"000";
    constant  START_REFRAME: StateType := b"001";
    constant COUNT_2_CLOCKS: StateType := b"010";
    constant  LOOK_FOR_xRDY: StateType := b"011";
    constant      HANDSHAKE: StateType := b"100";

    signal current_state, next_state : StateType;
    signal fb_OUT_OF_LOCK : bit;

    signal count2: bit_vector(0 to 1);          -- 2-bit counter
    signal error_count: bit_vector(0 to 4);     -- 5-bit counter
    signal rcvdbyts_count: bit_vector(0 to 6);  -- 7-bit counter

begin

counters: process (CLK, RVS, reset, rcvdbyts_count, error_count,
       out_of_lock) begin

  if (clk'event and clk = '1') then

    if (reset = '1') then
      fb_out_of_lock    <= '0';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
```

Appendix A. VHDL Description (continued)

```
    elsif (error_count = "10000") then
      fb_out_of_lock    <= '1';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    elsif (rcvdbyts_count = "1000000") then
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    else
      rcvdbyts_count  <= rcvdbyts_count + 1;
      if (RVS = '1') then
        error_count <= error_count + 1;
      end if;
    end if;

    if (current_state = LOOK_FOR_xRDY) and (xRDY = '0') then
      fb_out_of_lock <= '0';
    end if;

    if (current_state = COUNT_2_CLOCKS) then
      count2 <= count2 + 1;
    else
      count2 <= "00";
    end if;

  end if;

end process; --counters

next_st_comb:  process (CLK, fb_OUT_OF_LOCK, DO_REFRAME, FORCE_RFOUT, xRDY,
                        RFDONE_ACK, RESET, RF_ENABLE, current_state) begin

  if (RESET = '1') then
    next_state <= IDLE;
  elsif (RF_ENABLE = '0') then
    next_state <= DISABLED;
  else
    case current_state is

    when IDLE =>

      if (fb_OUT_OF_LOCK = '1') or (DO_REFRAME = '1') then
        next_state <= START_REFRAME;
      else
        next_state <= current_state;
      end if;

    when START_REFRAME =>
```

Appendix A. VHDL Description (continued)

```
      next_state <= Count_2_Clocks;

    when COUNT_2_CLOCKS =>

      if (count2 = "11") then
        next_state <= LOOK_FOR_xRDY;
      else
        next_state <= current_state;
      end if;

    when LOOK_FOR_xRDY =>

      if (xRDY = '0') and (DO_REFRAME = '1') then
        next_state <= HANDSHAKE;
      elsif (xRDY = '0') and (DO_REFRAME = '0') then
        next_state <= IDLE;
      else
        next_state <= current_state;
      end if;


    when HANDSHAKE =>

      if (RFDONE_ACK = '1') then
        next_state <= IDLE;
      else
        next_state <= current_state;
      end if;

    when DISABLED =>

      if (RF_ENABLE = '0') then
        next_state <= current_state;
      else
        next_state <= IDLE;
      end if;

    end case;
  end if;
end process; --next_st_comb

outp_comb:  process (current_state, FORCE_RFOUT) begin

  if (FORCE_RFOUT = '1') then
    RF <= '1';
  else
    case current_state is
```

Appendix A. VHDL Description (continued)

```
    when IDLE =>
      RF        <=  '0';
      RFDONE_HS <=  '0';

    when START_REFRAME =>
      RF        <=  '1';
      RFDONE_HS <=  '0';

    when COUNT_2_CLOCKS =>
      RF        <=  '1';
      RFDONE_HS <=  '0';

    when LOOK_FOR_xRDY =>
      RF        <=  '1';
      RFDONE_HS <=  '0';

    when HANDSHAKE =>
      RF        <=  '0';
      RFDONE_HS <=  '1';

    end case;
  end if;
end process; --outp_comb

seq_assgnmnt:  process (clk) begin

  if (clk'event and clk = '1') then

    current_state <= next_state;

    if (RVS = '1') then
      ERROR <= '1';
      if (D = x"E4" or D = x"E2" or D = x"E1") then
        UNDEF_CHAR <= '0';
        RDISP_ERR  <= '1';
      else
        UNDEF_CHAR <= '1';
        RDISP_ERR  <= '0';
      end if;
    else
      ERROR <= '0';
      UNDEF_CHAR <= '0';
      RDISP_ERR  <= '0';
    end if;

  end if;

end process; --seq_assgnmnt
```

Appendix A. VHDL Description (continued)

```
-- concurrent assignment statements
-- outputs and local feedback signals made the same

curr_st     <=  current_state;
rb_cntr     <=  rcvdbyts_count;
err_cntr    <=  error_count;
OUT_OF_LOCK <=  fb_out_of_lock;

end CNTRL933;  -- end architecture
```

Implementing a 128Kx32 Dual-Port RAM Using the FLASH370™

Introduction

More and more communication systems require the use of very deep, high-speed dual-port memories to provide a common storage area for use between processors. System designers are looking for dual-port memories of 128 KByte and larger in size. These same systems are using 32-bit buses. These larger dual-port memories are not readily available as monolithic devices. As a result, the designer is left with the task of implementing these devices using discrete components. A full-featured implementation would include some static RAM combined with external support logic, arbitration, and control functions. This application note describes how to implement a 128K x 32-bit-wide dual-port memory or larger, using high-speed 1M SRAMs and a Cypress CPLD, the CY7C371. The CPLD, or Complex Programmable Logic Device, will be used to implement the memory control functions of the dual-port system and will be coded using VHDL.

Dual-Port Block Diagram

A good reference for the function and operation of a dual-port memory can be found in the application note in the Cypress *Applications Handbook* titled "Understanding Dual-Port RAMs." To reiterate, the block diagram of a standard dual-port memory is shown in *Figure 1*. This block diagram indicates the various blocks associated with a dual-port. There are four major blocks: the memory array, the

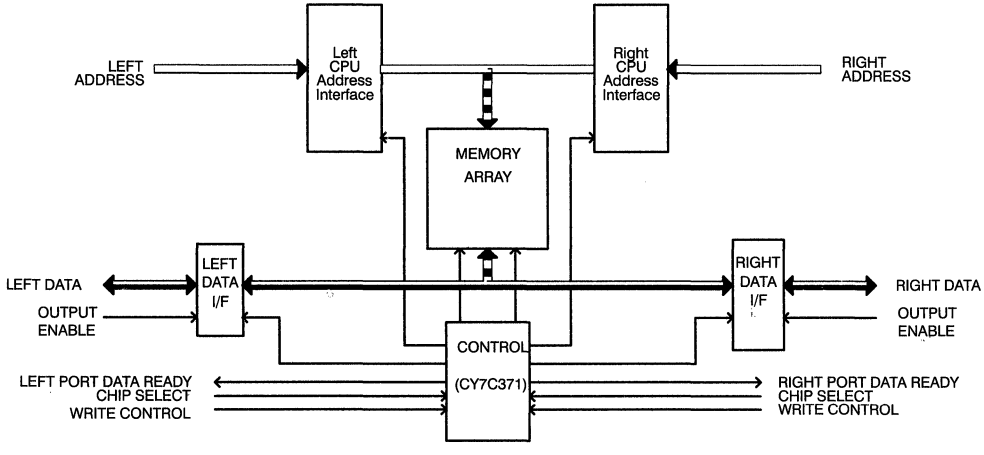


Figure 1. Dual-Port Memory Array Block Diagram

arbitration/control function, the right port or interface, and the left port or interface.

As can be seen from the block diagram in *Figure 1*, there are a series of signals that are required both internal and external to this system. The external signals are the normal signals that a monolithic dual-port chip would have. These are the signals that are labeled in the block diagram. The other signals are the internal signals that are used to allow the pieces of this dual-port system to communicate with one another. These are the address output enables for the address interface logic, the data output enable and the latch enable for the data interface logic, and the RAM output enable and write enable. These will be discussed in detail later.

The memory array consists of a single, standard SRAM or group of SRAMs to make up the overall array size. This array can be expanded in depth and width as needed. The arbitration/control logic accepts asynchronous read or write requests from each port or interface and sequences through a series of internal states that perform the read or write operation on the memory array. A CPLD is used in this example to implement this logic. The control logic must arbitrate between requests as well as synchronize the inputs to the internal clock frequency of the control function. The address buffers are used to isolate the address bus of the memory array from the left and right address ports. This allows the control-logic CPLD to select the correct address at the proper time. The bidirectional, latched data path allows data to be written to or read from the memory array. The data is also held in the latch during the remainder of the access.

Use of SRAM for Dual-Port

A 128Kx8 SRAM (like the Cypress CY7C109, 25-ns SRAM, as used in this note) was chosen here to implement a 128K x 32 sized array. Appendix A shows the schematic representation of the design. The array can be any size; this note shows this configuration because it depicts how to expand in the width direction. Cascading devices to expand the depth of the array is just as easily implemented. In either case, the contents of the control logic CPLD remain the same. The array could also be implemented with a single SRAM device if the array size warrants it.

A Brief Description of the CY7C371

The CY7C371 is a complex PLD with 32 macrocells, 32 I/O pins and 6 dedicated input pins (including 2 clock pins). The macrocells are grouped into two Logic Blocks of 16 macrocells each. There is a programmable interconnect matrix or PIM that connects the two logic blocks to the inputs and to each other. The macrocells themselves contain a register that can be configured as a T flip-flop, a D flip-flop, a level-triggered latch, or can be bypassed for combinatorial product terms. Each macrocell can support up to 16 product terms. For more detailed information on the CY7C371 and the whole FLASH370 family of CPLDs, please consult the application note "The FLASH370™ Family Of CPLDs and Designing with *Warp2*™" in the *Cypress Applications Handbook*.

The CY7C371 is well suited to this application. The dedicated inputs can be configured with a double registering mechanism to synchronize asynchronous signals so that they can be used synchronously inside the CPLD. The double registering will also dramatically reduce the chance of a metastable condition. The CPLD architecture is optimal for state machine designs and this arbiter requires three state machines to define it. The double-registered input configuration will be used in this example to resync the asynchronous chip select and write control inputs from both ports.

State Machine Design

The finite state machine that controls the dual-port memory array is really comprised of three "dependent" state machines operating concurrently as shown in *Figure 2*. Dependent state machines monitor or depend on the state of another state machine in order to change state. The first two machines, called "leftside" and "rightside," are identical. Their primary task is to monitor the interface of both ports. When the chip select input ($\overline{R\_CS}$ or $\overline{L\_CS}$) goes active (logic LOW), the appropriate machine advances from the Ready state to the

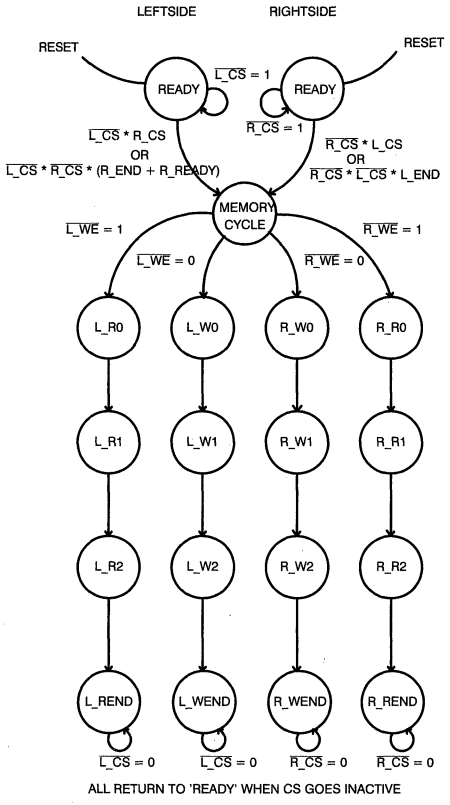


Figure 2. Memory Control Function State Machine

Memory Cycle state. The Memory Cycle state will start one of the memory access sequences. The length of each memory sequence (i.e., the number of state machine cycles) can be "tuned" to the access time of the SRAMs in the memory array. The memory cycle state machine will cycle back to the Ready state at the same time the memory access sequence ends and the select input goes inactive. It will either wait for a new request or start another memory access depending on the state of the other state machine ("leftside" or "rightside"). In the case where two requests are pending or appear at the same time, the left port gets priority. This means that the memory access for the left port is performed first. A READY signal (L_READY and R_READY) indicates when data is available on either port, it can only be active when either select input is active.

State Machine Implementation

The actual implementation of the state machines in the CY7C371 is done using VHDL. The structure of VHDL allows for simplification in coding these dependent state machines; the use of multiple processes and the CASE statement prove to be very powerful and efficient ways to perform this task.

Upon reset, both rightside and leftside state machines enter the Ready state and wait for a memory access. The leftside state machine will be used as an example. Both sides are identical at this point. Once a request is detected [for example $\overline{L\_CS}$ goes active (=0)], the leftside state machine transitions into a memory cycle. A priority scheme favoring the left port is encoded into the process for both state machines. If two accesses occur simultaneously, the left one is performed first. If one port request is detected before the other, it is completed while the other is held off. This extends the overall access time of the memory, but allows for "fair" operation. Each memory access sequence, Left Read, Left Write, Right Read, and Right Write, is comprised of four states. The four states (R0, R1, R2, REND or W0, W1, W2, WEND) run sequentially, one per clock cycle. They are there to allow the proper timing for the generation of control signals to the various components in the dual-port system. The REND or WEND state indicates the end of a memory cycle and is also a hold state if the CS is still active for that particular port. Once the REND or WEND state is reached and the CS is inactive, the state machine returns to the READY state and another access can be initiated.

CY7C371 Signals

A total of ten outputs are required to control the memory array and both the left and right ports. Refer to Appendix A for the 128K x 8 dual-port memory array schematic. The SRAM in the array is controlled by $\overline{RAM\_OE}$ and $\overline{RAM\_WE}$. The $\overline{RAM\_OE}$ signal is created when either port executes a read successfully. Therefore, the $\overline{RAM\_OE}$

signal is enabled during either read sequence only during the R0 through R2 cycles. Writes to the SRAM are controlled by the write state machine for either port. The $\overline{\text{RAM\_WE}}$ is generated for either port during the W1 and W2 cycles of a write access only. The port address inputs are isolated from the memory array by a set of 74FCT244Ts. The left port is controlled by $\overline{\text{L\_ADD\_OE}}$ and is generated during the left memory access sequence states 0 through 2 for either a read or a write to the left port. The right port address is controlled in the same manner, by using the right memory access sequence states 0 through 2. The data buffer functions are implemented using 74FCT543Ts with the "B" (HIGH current) side interfaced to the outside and the "A" side interfaced to the memory array. During reads, the latch enables ($\overline{\text{L\_LAT\_EN}}$, $\overline{\text{R\_LAT\_EN}}$) are used to hold the data read from the array in the latches. The output enables ($\overline{\text{L\_OE}}$, $\overline{\text{R\_OE}}$) are then driven directly to access the read data. During writes, the output enables ($\overline{\text{L\_DAT\_OE}}$, $\overline{\text{R\_DAT\_OE}}$) are used to allow the data to pass from the outside into the memory array. These output enables and latch enables are controlled by the OR of the appropriate memory access sequence states. Mealy outputs are used for the L_READY and R_READY signals. These outputs are active whenever the respective state machine is in state 2 and the CS is active. Using Mealy outputs here allows the ready signal to go inactive as soon as the CS input ($\overline{\text{L\_CS}}$ or $\overline{\text{R\_CS}}$) goes inactive instead of waiting for the state machine to transition back to the READY state.

VHDL Code for Controller in 371

Appendix B contains the VHDL code used for the CY7C371 in this design. This code was compiled with the Cypress *Warp2* tool and targeted for the CY7C371 to generate the programming (JEDEC) and simulation file(s). The Nova simulator in the *Warp2* tool was used to verify the design. For details on these tools please refer to the *Warp2 User's Guide*. Furthermore, a thorough explanation of VHDL constructs can be found in the *Warp2 Reference Manual*.

The code in Appendix B starts out by defining the inputs and outputs and the internal signals required. The first process is for the Chip Select and Write Enable resync. This is where the double registering occurs, as mentioned in the description of the CY7C371 earlier in this application note. The next process is where the state machine definitions start. It begins by defining the rightside state machine and uses a separate process to define the leftside state machine. Buried within each of these processes is the Memory Cycle state machine for the READ and WRITE cycles of each port. The next process is used to define the RAM_OE and RAM_WE for the memory array control. This is a simple IF-THEN-ELSE clause. The last process is used to generate the signal which gets used in the Mealy equations for the leading edge of the L_READY and R_READY signals. Lastly, the L_READY and R_READY signals are defined outside of a process by gating state2 with the CS input.

Performance Evaluation

To evaluate the performance of this dual-port system, three different timing scenarios were looked at. The first scenario is for an unarbitrated access from either port. This assumes that both port state machines are in the Ready state and only one access occurs. The second scenario involves the right port being granted access shortly before the left port, forcing the left port to wait. The third involves simultaneous accesses from each port. In this case the left side has priority (by design) and the right side is held off. These cases are shown in the following three timing diagrams (*Figures 3*, *4*, and *5*). From these it is possible to determine the timing of each access by counting the number of clock cycles for each scenario.

Table 1 lists the number of clock cycles for each of the three cases of *Figures 3*, *4*, and *5*. These numbers reflect the worst case situations for Case #2 and #3 where the maximum possible delay is assumed.

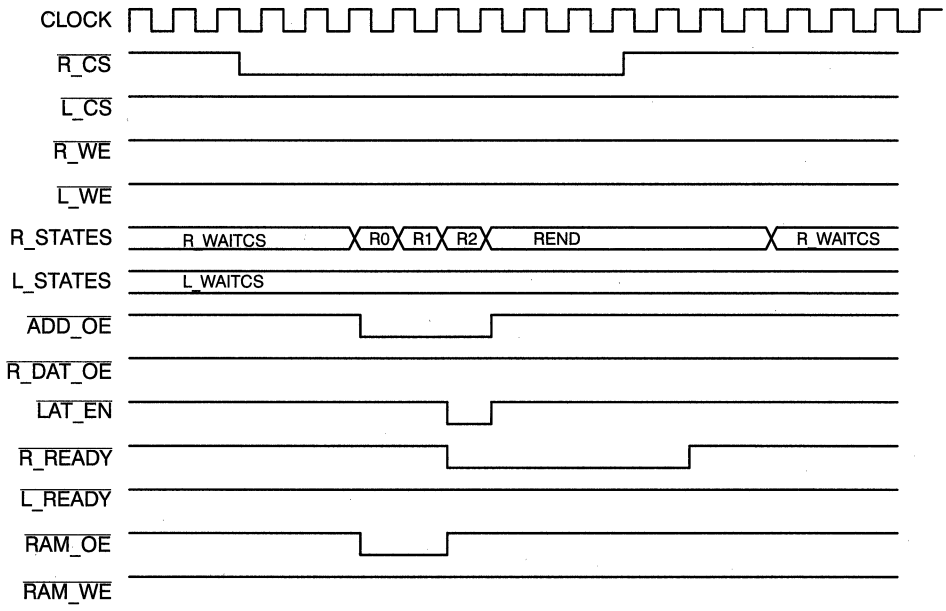


Figure 3. Timing Diagram—Unarbitrated Access From Right Port

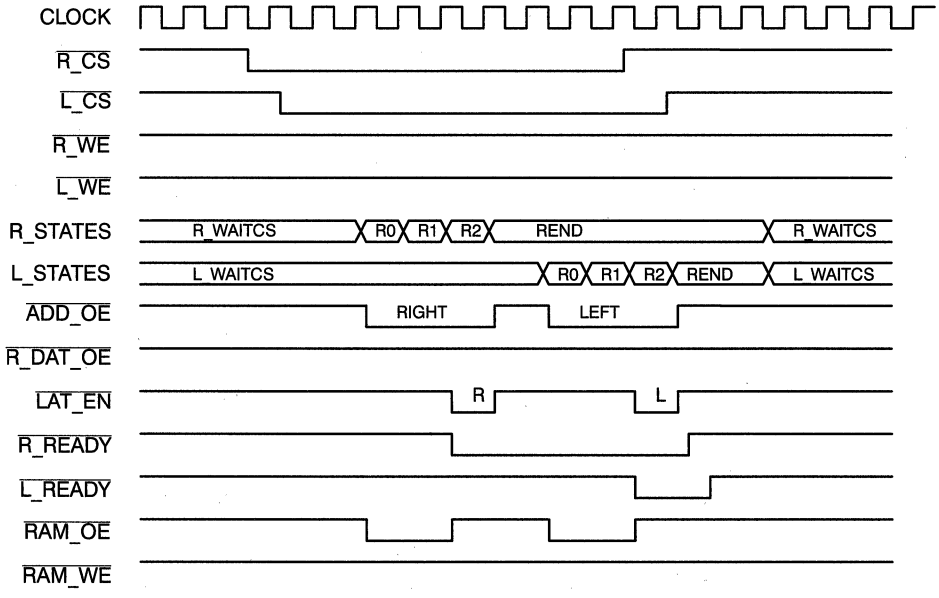


Figure 4. Timing Diagram—Right Port Access Before Left Port

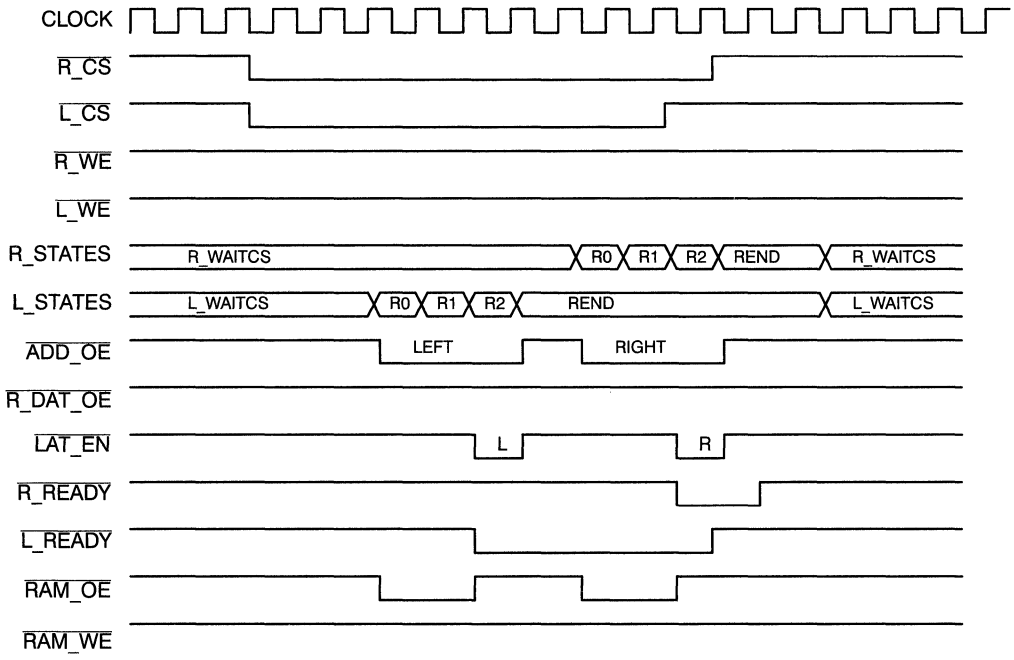


Figure 5. Timing Diagram—Simultaneous Access

Table 1. Access Time in Clock Cycles

Timing Parameter	Case #1	Case #2	Case #3
LEFT			
Input Set-Up Timing	2 clocks	Note 1	2 clocks
Arbitration Cycle	1 clock	Note 1	1 clock
Memory Access	3 clocks	3 clocks	3 clocks
Latch Hold Cycle	1 clock	1 clock	1 clock
Total Number of Clock Cycles	7 clocks	11 clocks	7 clocks
RIGHT			
Input Set-Up Timing	N/A[2]	2 clocks	Note 1
Arbitration Cycle	N/A	1 clock	Note 1
Memory Access	N/A	3 clocks	3 clocks
Latch Hold Cycle	N/A	1 clock	1 clock
Total Number of Clock Cycles	N/A	7 clocks	11 clocks

Notes:

1. Worst case input set-up timing and arbitration cycle assumes 7 clock access delay on opposite port.
2. N\A means No Activity on this port.

To calculate the access time in nanoseconds, the following formula is applied:

$t_{ACC} = t_{IS371} + [t_{CYC371} \times \#clocks] + t_{PD543}$

Where:

t_{ACC} = total access time

t_{IS371} = CY7C371 input register set-up time = 2 ns

t_{CYC371} = clock cycle of CY7C371 = 7 ns

#clocks = number of clocks from *Table 1*

t_{PD543} = 74FCT543CT transparent to latched propagation delay = 7 ns

Since the CY7C371 inputs are double registered, two clock cycles are required to resync the Chip Select and Write Enable inputs. If the input set-up timing can be guaranteed, this internal delay of two cycles can be eliminated by using single- or non-registered inputs.

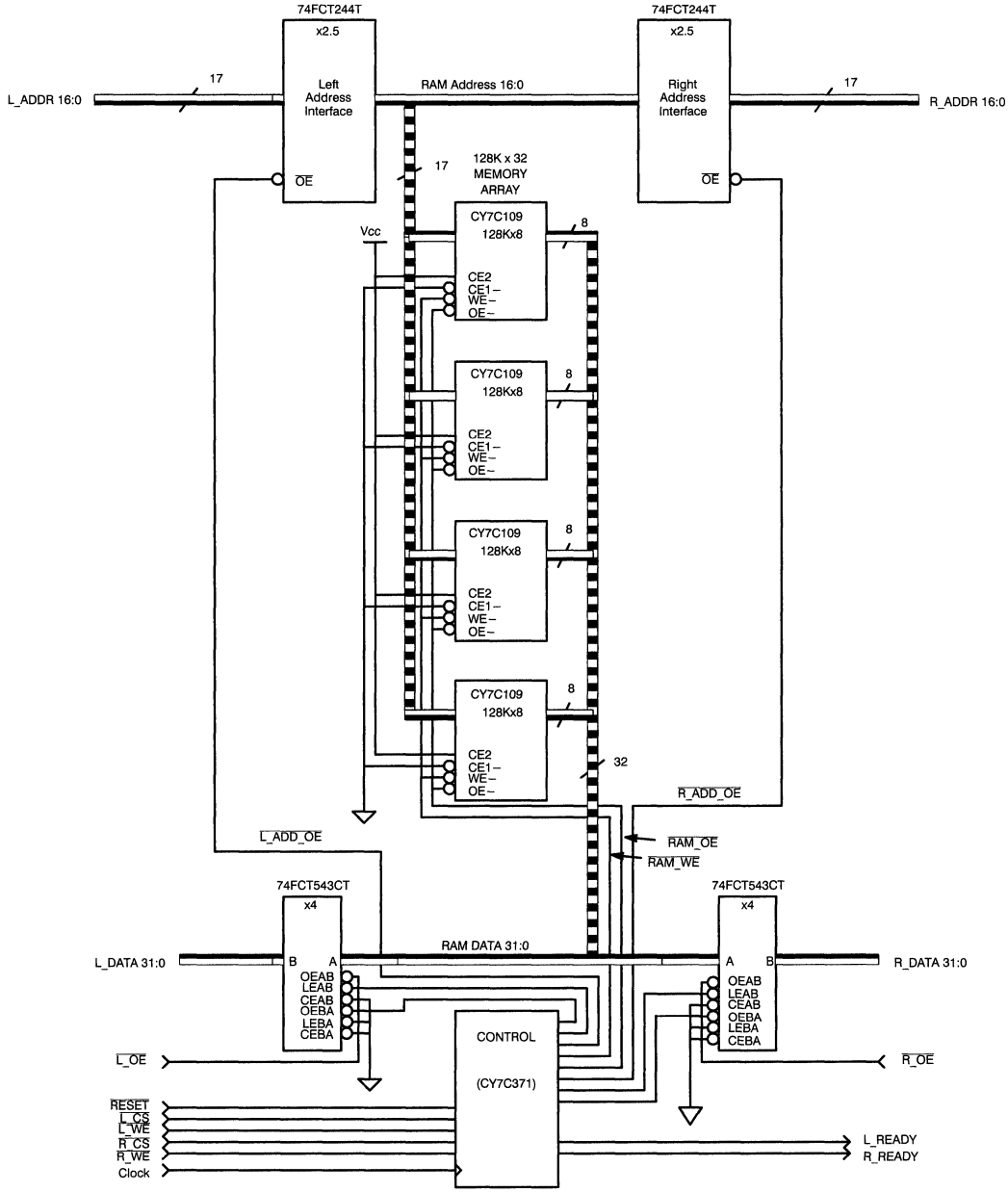
Memory Expansion

The example used here shows that an array of any size can be easily implemented. The addition of memories and associated address buffers makes depth expansion easy. The width may also be increased by cascading memories and adding additional buffers. Both techniques would be utilized to expand in depth and width. These enhancements are possible without making any changes to the CY7C371 Control Function PLD design. Likewise this design could implement a smaller array than shown here, again without revising the CY7C371.

Summary

This application note has demonstrated the implementation of a large asynchronous dual-port memory array by utilizing standard memory and logic devices and the CY7C371. The performance of this design is limited by various factors. The access time of the SRAM and the clock speed of the CY7C371 used are two factors that could improve performance without changing the VHDL code for the CY7C371. Another option would require some design changes, though minor. Making one or both ports synchronous with respect to the CPU would eliminate the two-clock delay associated with the resync function of the CY7C371. The implementation of these improvements offers the designer a few options to tailor the design to fit specific system requirements and achieve the desired level of performance.

Appendix A. Schematic



Appendix B. VHDL Code for Controller

```
-- Dual-port memory controller

ENTITY dpram IS
        PORT (clock, r_we_n, r_cs_n, l_we_n, l_cs_n, reset_n: IN BIT;     --INPUTS
              ram_oe_n, ram_we_n : OUT BIT;                               --OUTPUTS
              r_ready, r_add_oe, r_dat_oe, r_lat_en : OUT BIT;
              l_ready, l_add_oe, l_dat_oe, l_lat_en : OUT BIT
              );
END dpram;

-----------------------------------------------------------------------
USE work.rtlpkg.all;

ARCHITECTURE ARCHdpram OF dpram IS
  TYPE ctrl_states IS (waitcs, r0, r1, r2, rend, w0, w1, w2, wend);       --Internal signal declaration
  SIGNAL rightside, leftside : ctrl_states;
  SIGNAL r_we_ndd, r_we_nd, l_we_ndd, l_we_nd : BIT;
  SIGNAL r_cs_ndd, r_cs_nd, l_cs_ndd, l_cs_nd : BIT;
  SIGNAL r_ready_int, l_ready_int : BIT;
BEGIN

--Double register the input we and cs signals for sync & metastability hardening
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      r_we_ndd <= r_we_nd; r_we_nd <= r_we_n;
      l_we_ndd <= l_we_nd; l_we_nd <= l_we_n;
      r_cs_ndd <= r_cs_nd; r_cs_nd <= r_cs_n;
      l_cs_ndd <= l_cs_nd; l_cs_nd <= l_cs_n;
  END PROCESS;

--RIGHTSIDE STATE MACHINE
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      CASE rightside IS
        WHEN waitcs =>
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
--goto state 0 if : r_cs is active + L_cs  inactive or r_cs active + (l_cs active but at end)
          IF (((r_cs_ndd = '0') AND (l_cs_ndd = '1')) OR
              ((r_cs_ndd = '0') AND (l_cs_ndd = '0') AND
                ((leftside = wend) OR (leftside = rend)))) THEN
--start write state machine if WE active
            IF r_we_ndd = '0' THEN
              rightside <= w0;
              r_add_oe <= '0'; r_dat_oe <= '0'; r_lat_en <= '1';
            ELSE
--start read state machine if WE inactive
              rightside <= r0;
              r_add_oe <= '0'; r_dat_oe <= '1'; r_lat_en <= '1';
            END IF;
          ELSE
            rightside <= waitcs;
          END IF;

--RIGHTSIDE READ STATE MACHINE
        WHEN r0 =>
          rightside <= r1;
          r_add_oe <= '0'; r_dat_oe <= '1'; r_lat_en <= '1';
        WHEN r1 =>
          rightside <= r2;
          r_add_oe <= '0'; r_dat_oe <= '1'; r_lat_en <= '0';
        WHEN r2 =>
          rightside <= rend;
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
```

Appendix B. VHDL Code for Controller (continued)

```
        WHEN rend =>
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
          IF r_cs_ndd = '1' THEN
            rightside <= waitcs;
          ELSE
            rightside <= rend;
          END IF;

--RIGHTSIDE WRITE STATE MACHINE
        WHEN w0 =>
          rightside <= w1;
          r_add_oe <= '0'; r_dat_oe <= '0'; r_lat_en <= '1';
        WHEN w1 =>
          rightside <= w2;
          r_add_oe <= '0'; r_dat_oe <= '0'; r_lat_en <= '1';
        WHEN w2 =>
          rightside <= wend;
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
        WHEN wend =>
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
          IF r_cs_ndd = '1' THEN
            rightside <= waitcs;
          ELSE
            rightside <= wend;
          END IF;
        WHEN others =>
          rightside <= waitcs;
          r_add_oe <= '1'; r_dat_oe <= '1'; r_lat_en <= '1';
      END CASE;
  END PROCESS;

--LEFTSIDE STATE MACHINE
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      CASE leftside IS
        WHEN waitcs =>
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
--goto state 0 if l_cs is active + r_cs is inactive or l_cs active + (r_cs active but at end or in waitcs
 state)
          IF (((l_cs_ndd = '0') AND (r_cs_ndd = '1')) OR
              ((l_cs_ndd = '0') AND (r_cs_ndd = '0') AND
                ((rightside = wend) OR (rightside = rend) OR (rightside = waitcs)))) THEN
--start write state machine if WE active
            IF l_we_ndd = '0' THEN
              leftside <= w0;
              l_add_oe <= '0'; l_dat_oe <= '0'; l_lat_en <= '1';
            ELSE
--start read state machine if WE inactive
              leftside <= r0;
              l_add_oe <= '0'; l_dat_oe <= '1'; l_lat_en <= '1';
            END IF;
          ELSE
            leftside <= waitcs;
          END IF;

--LEFTSIDE READ STATE MACHINE
        WHEN r0 =>
          leftside <= r1;
          l_add_oe <= '0'; l_dat_oe <= '1'; l_lat_en <= '1';
        WHEN r1 =>
          leftside <= r2;
          l_add_oe <= '0'; l_dat_oe <= '1'; l_lat_en <= '0';
        WHEN r2 =>
          leftside <= rend;
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
```

Appendix B. VHDL Code for Controller (continued)

```
        WHEN rend =>
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
          IF l_cs_ndd = '1' THEN
            leftside <= waitcs;
          ELSE
            leftside <= rend;
          END IF;

--LEFTSIDE WRITE STATE MACHINE
        WHEN w0 =>
          leftside <= w1;
          l_add_oe <= '0'; l_dat_oe <= '0'; l_lat_en <= '1';
        WHEN w1 =>
          leftside <= w2;
          l_add_oe <= '0'; l_dat_oe <= '0'; l_lat_en <= '1';
        WHEN w2 =>
          leftside <= wend;
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
        WHEN wend =>
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
          IF l_cs_ndd = '1' THEN
            leftside <= waitcs;
          ELSE
            leftside <= wend;
          END IF;
        WHEN others =>
          leftside <= waitcs;
          l_add_oe <= '1'; l_dat_oe <= '1'; l_lat_en <= '1';
      END CASE;
  END PROCESS;

--RAM_OE and RAM_WE control signal logic
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
          IF (((rightside = waitcs) AND ((((r_cs_ndd = '0') AND (l_cs_ndd = '1') AND (r_we_ndd = '1')) OR
              ((r_cs_ndd = '0') AND (l_cs_ndd = '0') AND (r_we_ndd = '1') AND
                ((leftside = wend) OR (leftside = rend))))))
             OR
             ((leftside = waitcs) AND (((l_cs_ndd = '0') AND (r_cs_ndd = '1') AND (l_we_ndd = '1')) OR
              ((l_cs_ndd = '0') AND (r_cs_ndd = '0') AND (l_we_ndd = '1') AND
                ((rightside = wend) OR (rightside = rend) OR (rightside = waitcs)))))
             OR
             (rightside = r0) OR (rightside = r1)
             OR
             (leftside = r0) OR (leftside = r1))
          THEN
            ram_oe_n <= '0';
          ELSE
            ram_oe_n <= '1';
          END IF;

          IF ((leftside = w0) OR (leftside = w1) OR
              (rightside = w0) OR (rightside = w1))
          THEN
            ram_we_n <= '0';
          ELSE
            ram_we_n <= '1';
          END IF;
  END PROCESS;
```

Appendix B. VHDL Code for Controller (continued)

```
--READY signal logic for leading edge of signal
  PROCESS BEGIN
    WAIT UNTIL clock = '1';
      IF ((rightside = r1) OR (rightside = w1)) THEN
        r_ready_int <= '0';
      END IF;
      IF ((r_cs_nd = '1') OR (reset_n = '0')) THEN
        r_ready_int <= '1';
      END IF;

      IF ((leftside = r1) OR (leftside = w1)) THEN
        l_ready_int <= '0';
      END IF;
      IF ((l_cs_nd = '1') OR (reset_n = '0')) THEN
        l_ready_int <= '1';
      END IF;
  END PROCESS;

--MEALY outputs for READY signal to turn off as soon as CS goes inactive
  l_ready <= '0' WHEN ((l_ready_int = '0') AND (l_cs_nd = '0')) ELSE '1';
  r_ready <= '0' WHEN ((r_ready_int = '0') AND (r_cs_nd = '0')) ELSE '1';

END ARCHdpram;
```

Efficient Arithmetic Designs Targeting FLASH370™ CPLDs

Introduction

The design of fast and efficient arithmetic elements is imperative because of its applications in the many areas of science and engineering. It is important for designers to be aware of the choices available to them in selecting an efficient algorithm for their application. Even the seemingly simple arithmetic operations turn out to be more complex than one expects, when attempting to implement them. There is a lot of literature available in the field, but very little provides the level of detail required to go all the way from a concept to a final implementation.

This application note is intended to help designers create efficient arithmetic designs targeting a FLASH370™ complex programmable logic device (CPLD). The designer has many alternatives in choosing between arithmetic implementations for a given design. The decision on the final choice is typically based on issues like resource availability, speed of operation, and modularity. Creating designs in view of the target device's architecture will definitely yield better results than implementing a generic design on the same device. The discussion in this application note addresses arithmetic algorithms, design methodologies, and implementations tailored to the features and resources offered in the FLASH370 family of CPLDs. These specialized arithmetic designs achieve a balanced tradeoff between speed/area requirements for a given application. In this application note the user is offered a wide variety of algorithms and implementations to choose from. This variety provides the designer with the flexibility to choose the model best suited for the target application. This choice is absolutely necessary, since design requirements and constraints vary from application to application.

The discussion assumes that the designer has a good feel for the features and resources available in the FLASH370 family of CPLDs. The implementation details and design tradeoffs in building adders, subtracters, equality and magnitude comparators are addressed in this application note. This application note includes many VHDL (VHSIC Hardware Design Language) examples to illustrate the working and implementation of the algorithms presented. Block diagrams are also presented wherever necessary to help the designer understand the design better.

All algorithms in this application note are described within the same framework, so that the similarities between different algorithms become evident and consequently, the basic principle behind these algorithms can be easily identified. This application note is also intended to create a solid foundation from which designers can pick up ideas and concepts and create their own algorithms/implementations.

The VHDL code presented in this application note are intentionally presented in a simple style. The intent of this application note is to allow a designer to visualize and implement arithmetic models efficiently and not to explain how to code them. All VHDL keywords are presented in italics. This application note also assumes that the reader has a good grasp of the fundamentals of VHDL. Some of the LPM (library of parameterized modules) elements for CPLDs provided in the *Warp*™ software are built using the concepts and final implementations discussed here. This provides the user with an excellent opportunity to choose the best algorithm

and implementation tailored to the target application.

Adders

The addition of two operands is the most frequent operation in almost any arithmetic unit. The two-operand adder is commonly used in performing additions and subtractions. It is also used when executing complex arithmetic functions like multiplication and division.

ADD : 1-Bit Full Adder

The basic component used in adding two operands is called a *Full Adder*. The full adder element will be henceforth referred to as the 'ADD' component. The block diagram and functionality of ADD is shown in *Figure 1*. A and B are the two operands to be added and CI is the Carry-in to the component. SUM and CO are the Sum and Carry-out from the component.

The VHDL code describing the functionality of the ADD component is shown here. This design takes one pass through the Logic (AND-OR) array to fit into a FLASH370 CPLD. The ADD component instantiated in the VHDL code shown has exactly the same functionality shown in *Figure 1*.

```
-- This VHDL code invokes the im-
plementation of the MATH PKG ele-
ment ADD

USE WORK.CYPRESS.ALL;
USE WORK.MATHPKG.ALL;

ENTITY add IS
 PORT (CI: IN BIT;
   A, B: IN BIT;
   SUM: OUT BIT;
   CO: OUT BIT);
END add;

ARCHITECTURE archadd OF add IS
BEGIN
 i1: add PORT MAP(CI,A,B,SUM,CO);
END archadd;
```

RADD12 : 12-Bit Ripple Carry Adder

An *n*-bit two-operand ripple carry adder can be built using *n* ADD components. All the 2*n* input bits are available to the adder at the same time. However the carries have to propagate from the LSB position to the MSB. In other words, we need to wait until the carries ripple through *n* ADD components to claim that the SUM outputs are correct. Because of this rippling effect, the adder is referred to as the *Ripple Carry* Adder. This is the simplest form of adding any two operands. It uses the least amount of area compared to all other implementations but, on the negative side, is the slowest implementation. This is typically the implementation provided with a synthesis tool when it recognizes the '+' operator in a VHDL code. The block diagram of a 12-bit Ripple Carry Adder (RADD12) is shown in *Figure 2*.

The VHDL code describing the functionality of the RADD12 component is shown here. This design takes 12 passes through the logic array to fit into a FLASH370 CPLD. The outputs of the LSB ADD

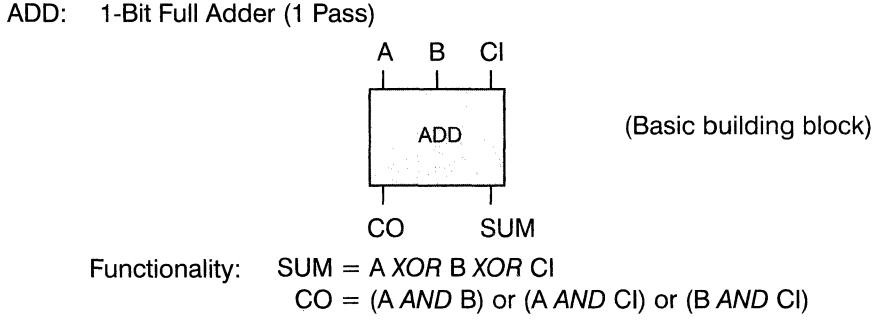


Figure 1. Block Diagram and Functionality of a Full Adder

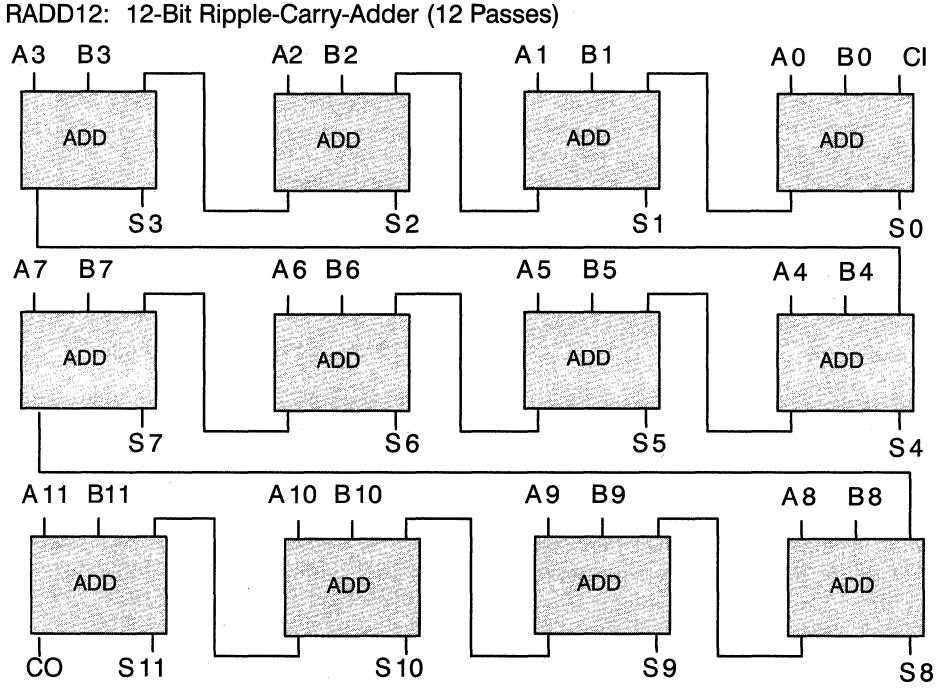


Figure 2. Block Diagram of a 12-Bit Ripple Carry Adder

component are produced in the first pass. The outputs of the succeeding ADD components are produced with every alternate pass through the logic array. Each pass through the logic array has a time penalty associated with it. It is recommended that the reader understand the timing issues associated with the FLASH370 CPLD (refer to the "CY7C37x Timing Parameters" application note).

```
--This VHDL code describes the implementation of a generic
--12 bit ripple carry adder.

USE WORK.CYPRESS.ALL;
USE WORK.MATHPKG.ALL;

ENTITY rippleadd12 IS
  PORT (CI: IN BIT;
    A11, A10, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0 : IN BIT;
    B11, B10, B9, B8, B7, B6, B5, B4, B3, B2, B1, B0 : IN BIT;
    SUM11, SUM10, SUM9, SUM8, SUM7, SUM6, SUM5, SUM4,
    SUM3, SUM2, SUM1, SUM0 : OUT BIT;
    CO: OUT BIT);
END rippleadd12;

ARCHITECTURE archripple12add OF rippleadd12 IS
```

```
SIGNAL C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11 : BIT;

attribute synthesis_off of C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11 :
signal is true;

BEGIN

  i1: add PORT MAP(CI,A0,B0,SUM0,C1);
  i2: add PORT MAP(C1,A1,B1,SUM1,C2);
  i3: add PORT MAP(C2,A2,B2,SUM2,C3);
  i4: add PORT MAP(C3,A3,B3,SUM3,C4);
  i5: add PORT MAP(C4,A4,B4,SUM4,C5);
  i6: add PORT MAP(C5,A5,B5,SUM5,C6);
  i7: add PORT MAP(C6,A6,B6,SUM6,C7);
  i8: add PORT MAP(C7,A7,B7,SUM7,C8);
  i9: add PORT MAP(C8,A8,B8,SUM8,C9);
  i10:add PORT MAP(C9,A9,B9,SUM9,C10);
  i11:add PORT MAP(C10,A10,B10,SUM10,C11);
  i12:add PORT MAP(C11,A11,B11,SUM11,CO);

END archripple12add;
```

The need and use for the 'Synthesis_off' attribute used in the VHDL code will be discussed a little later.

ADD2WC: 2-Bit Adder with Carry-Out

The concept of the ADD component can be extended to create a 2-bit adder which takes in two 2-bit operands with a carry-in and produces a 2-bit SUM and a carry-out as outputs. This component will be referred to as the ADD2WC (2-bit adder with a carry-out). This also takes just one pass through the logic array to yield results. The block diagram of ADD2WC is shown in *Figure 3*. A0, A1 and B0, B1 are the two operands to be added and CI is the Carry-in to the component. S0, S1 and CO are the Sums and Carry-outs from the component.

The VHDL code describing the functionality of the ADD2WC component is shown here. This design takes one pass through the logic array to fit into a FLASH370 CPLD.

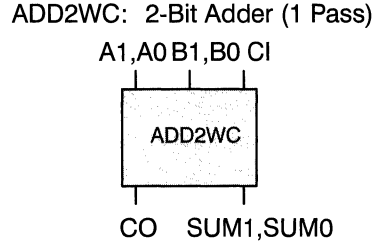


Figure 3. A 2-Bit Full Adder with a Carry-Out

```
--VHDL code describing a 2-bit adder with carry-out.

USE WORK.RTLPKG.ALL;

PACKAGE add2wc_pkg IS
  COMPONENT add2wc PORT(
    CI : IN BIT;
    A1,A0: IN BIT;
    B1,B0: IN BIT;
    SUM1,SUM0 : OUT BIT;
```

```
    CO: OUT BIT);
  END COMPONENT;
END add2wc_pkg;

ENTITY add2wc IS
  PORT (CI : IN BIT;
        A1,A0: IN BIT;
        B1,B0: IN BIT;
        SUM1,SUM0 : OUT BIT;
        CO: OUT BIT);
END add2wc;

ARCHITECTURE archadd2wc OF add2wc IS

BEGIN

SUM0 <= A0 XOR B0 XOR CI;
SUM1 <= A1 XOR B1 XOR ((A0 AND B0) OR (A0 AND CI) OR (B0 AND CI));
CO   <= (A0 AND B0 AND B1)
  OR (A0 AND B0 AND A1)
  OR (CI AND B0 AND B1)
  OR (CI AND B0 AND A1)
  OR (CI AND A0 AND B1)
  OR (CI AND A0 AND A1)
  OR (A1 AND B1);

END archadd2wc;
```

The concept of ADD2WC can be extended to describe the ADD2NC component. The ADD2NC component is a cut-down version of the ADD2WC component, and does *not* have a carry-out. The VHDL code and block diagram for the ADD2NC component is easy to extrapolate and is not shown here.

R2ADD12: 12-Bit Ripple Carry Adder using the ADD2WC as a Basic Block

A 12-bit adder using the ADD2WC component is shown here. This adder takes 6 passes to produce all results, as opposed to the 12 passes needed for the 12-bit adder using the ADD component. The outputs of the LSB ADD2WC component are produced in the first pass. The outputs of the succeeding ADD2WC components are produced with every alternate pass through the logic array. The number of macrocells used by this scheme is less than RADD12, but the product term count is higher. A comparison of different schemes is presented later. The block diagram of R2ADD12 is shown in *Figure 4*. The VHDL code describing the functionality is also attached.

R2ADD12: 12-Bit Adder using ADD2WC (6 Passes)

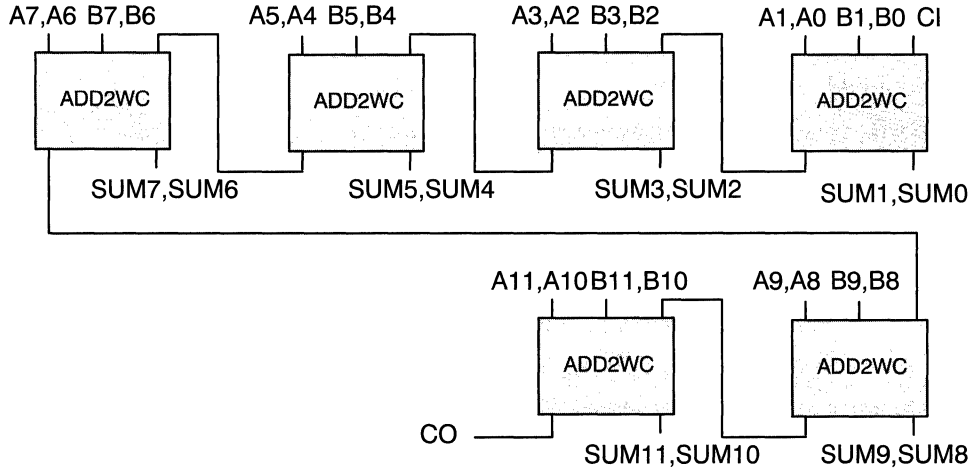


Figure 4. Block Diagram of a 12-Bit Ripple Carry Adder Using 2-Bit Adders

```
--A 12-bit Ripple carry adder built using the ADD2WC element as a basic
--building block

USE WORK.RTLPKG.ALL;
USE WORK.ADD2WC.ALL;

ENTITY add12 IS
  PORT (CI : IN BIT;
        A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
        B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
        SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,
        SUM3,SUM2,SUM1,SUM0 : OUT BIT;
        CO: OUT BIT);
END add12;

ARCHITECTURE archadd12 OF add12 IS

  SIGNAL C2, C4, C6, C8, C10 : BIT;

  attribute synthesis_off of C2, C4, C6, C8, C10 : signal is true;

BEGIN

  i1: add2wc PORT MAP(CI,A1,A0,B1,B0,SUM1,SUM0,C2);
  i2: add2wc PORT MAP(C2,A3,A2,B3,B2,SUM3,SUM2,C4);
  i3: add2wc PORT MAP(C4,A5,A4,B5,B4,SUM5,SUM4,C6);
```

```
  i4: add2wc PORT MAP(C6,A7,A6,B7,B6,SUM7,SUM6,C8);
  i5: add2wc PORT MAP(C8,A9,A8,B9,B8,SUM9,SUM8,C10);
  i6: add2wc PORT MAP(C10,A11,A10,B11,B10,SUM11,SUM10,CO);

END archadd12;
```

ADD3WC: The 3-Bit Ripple Carry Adder

There is yet another way we could implement an *n*-bit ripple carry adder targeting the FLASH370 CPLDs. We can implement the *n*-bit adder using the 3-bit group adder (ADD3WC) as opposed to a 2-bit group adder (ADD2WC). The problem with a 3-bit group adder is the sum-splitting of the functionality of the MSB Sum bit (SUM2). This takes more than 16 product terms (PTs) and takes 2 passes through the logic array to produce the result. All other results, including the carry-out, take less than 16 PTs and take just one pass to produce results. To control sum-splitting the functionality of SUM2, the intermediate carry C2 is created and assigned to a node. C2 is then used to create the functionality of SUM2. Note that the functionality of CO takes less than 16 PTs and is generated at the first pass, so the carry rippling is faster. This makes this component a faster building block. This scheme still takes two passes to create the functionality of SUM2, but without getting sum-split. The resource utilization of a 12-bit adder using the 3-bit group adder is presented later. The block diagram of the ADD3WC component is shown in *Figure 5*.

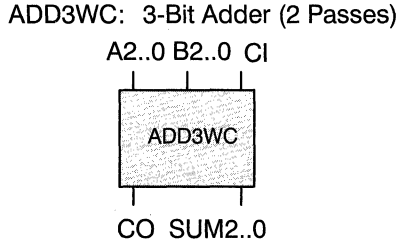


Figure 5. A 3-Bit Full Adder with a Carry-Out

```
-- 3-Bit Adder with Carry-out

PACKAGE add3wc_pkg IS
  COMPONENT add3wC
    PORT (CI : IN BIT;
      A2,A1,A0: IN BIT;
      B2,B1,B0: IN BIT;
      SUM2,SUM1,SUM0 : OUT BIT;
      CO: OUT BIT);
  END COMPONENT;
END add3wc_pkg;

ENTITY add3wc IS
  PORT (CI : IN BIT;
    A2,A1,A0: IN BIT;
    B2,B1,B0: IN BIT;
    SUM2,SUM1,SUM0 : OUT BIT;
    CO: OUT BIT);
END add3wc;

ARCHITECTURE archadd3wc OF add3wc IS

SIGNAL C2: BIT;

  attribute synthesis_off of C2: signal is true;
```

```
BEGIN

  SUM0 <= A0 XOR B0 XOR CI;
  SUM1 <= A1 XOR B1 XOR ((A0 AND B0) or (A0 AND CI) or (B0 AND CI));
  SUM2 <= A2 XOR B2 XOR C2;

  C2   <= (A0 AND B0 AND B1)
    OR (A0 AND B0 AND A1)
    OR (CI AND B0 AND B1)
    OR (CI AND B0 AND A1)
    OR (CI AND A0 AND B1)
    OR (CI AND A0 AND A1)
    OR (A1 AND B1);

  CO <= (A2 AND B2) OR ((A1 AND B1) AND (A2 OR B2))
    OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2))
    OR (CI AND (A0 OR B0) AND (A1 OR B1) AND (A2 OR B2));

END archadd3wc;
```

Function and Use of the *Synthesis_off* Attribute

The Synthesis_off attribute causes a signal to be made into a factoring point for logic equations and keeps the signal from being minimized out during optimization.

The attribute is useful for the following reasons:

1. It gives the user control over which equations or sub-expressions need to be factored into a node.
2. It helps in cutting down on compile time for designs that have a lot of 'signal redirection' (signals getting inverted/reassigned to other signals). This attribute provides the Logic optimizer a better control over the optimization process by reducing the number of signals it needs to deal with.
3. It provides better results for designs where a signal with a large functionality is being used by many other signals. If left alone, the fitter would collapse all the internal signals (which is desirable in many cases) and may drive the design's resource requirements beyond the available limits.

By using the Synthesis_off attribute, the user can assign the commonly-used signal to a node and bring down the resource utilization.

A side effect of using the Synthesis_off attribute is that the design will now take an extra pass through the array to achieve the same functionality. The extra pass may be required anyway, if more than 16 PTs are required.

This attribute is only recommended for use on combinatorial signals. Registered signals are assigned to a node by natural factoring and the Synthesis_off attribute on these signals is redundant.

This attribute can be associated with signals declared both in VHDL and schematics. The 'BUF' component can also be used in schematics and VHDL to achieve the same results as the Synthesis_off attribute. Please refer to the *Warp* Synthesis manual for more details.

Carry-Lookahead Principle

The predominant delay in adders is due to carry propagation. The carry-lookahead principle aims at minimizing this delay. The sum and carry equations for each bit position in an adder is given by:

$$S_i = A_i \text{ xor } B_i \text{ xor } C_i$$
$$C_{i+1} = (A_i \text{ and } B_i) \text{ or } (A_i \text{ and } C_i) \text{ or } (B_i \text{ and } C_i)$$

A carry is *generated* whenever A_i and B_i are both '1' and a carry is *propagated* whenever either A_i or B_i are '1'.

Generate term : $(G_i = A_i \text{ and } B_i)$
Propagate term : $(P_i = A_i \text{ or } B_i)$

Note: P_i can be (A_i xor B_i), but 'OR' is easier to implement than an 'XOR' in CPLDs.

Rewriting the equation for C_{i+1}, we get

$$C_{i+1} = G_i \text{ or } (P_i \text{ and } C_i)$$

Writing the equations for a 4-bit carry-lookahead adder:

$$C_1 = G_0 \text{ or } (P_0 \text{ and } C_0)$$
$$C_2 = G_1 \text{ or } (P_1 \text{ and } C_1)$$
$$C_3 = G_2 \text{ or } (P_2 \text{ and } C_2)$$
$$C_4 = G_3 \text{ or } (P_3 \text{ and } C_3)$$

where $G_i = (A_i \text{ and } B_i)$ and $P_i = (A_i \text{ or } B_i)$. The values of G_i and P_i can be generated in a single pass through the PIM array. The carry-in to any of the bit positions can be computed in a second pass through the array, based upon the values of the various G_is and P_is generated in the first pass.

The generalized carry-lookahead equation to compute the different carry-in signals is shown here:

$$C_{i+1} = G_i \text{ or } (P_i \text{ and } G_{i-1}) \text{ or } (P_i \text{ and } P_{i-1} \text{ and } G_{i-1}) \text{ or } \ldots \text{ or } (P_i \text{ and } P_{i-1} \text{ and } \ldots \text{ and } P_0 \text{ and } C_0)$$

We can further speed up the addition by providing a carry-lookahead over groups in addition to the internal lookahead within the group. We define a *group-generated* carry E and a *group-propagated* carry R, for a group of size 4 as follows: E = '1' if a carry-out (of the group) is generated internally and R = '1' if a carry-in (to the group) is propagated internally to produce a carry-out (of the group). The boolean equations for these carries are:

$$E = G_3 \text{ or } (P_3 \text{ and } G_2) \text{ or } (P_3 \text{ and } P_2 \text{ and } G_1) \text{ or } (P_3 \text{ and } P_2 \text{ and } P_1 \text{ and } G_0)$$
$$R = (P_3 \text{ and } P_2 \text{ and } P_1 \text{ and } P_0)$$

The group-generated and group-propagated carries for several groups can now be used to generate group carry-ins in a manner similar to single-bit carry-ins.

The selection of the group size plays an important role in obtaining the best possible implementation for a carry-lookahead adder in a CPLD. Some of the different possible implementations for a 12-bit carry-lookahead adder are shown in *Figure 6*.

The number of passes each of these implementations take and the number of product terms (PTs) and macrocells (MCs) used vary for each scheme (see *Table 2* in the "Comparison of Resource Utilization for Different Schemes in Building a 12-Bit Adder" section). Each scheme has its own advantage over the other. The user needs to judiciously choose between the different schemes based on the application, bit-size, and the CPLD chosen and its architectural constraints. The number of passes taken through the logic is a direct representation of the total time taken for producing final results. Each extra pass results in a time penalty. The rule to follow is, "The smaller the number of passes through the logic array, the faster your application runs." The implementation of a 12-bit carry-lookahead adder with different group-sizes is presented next.

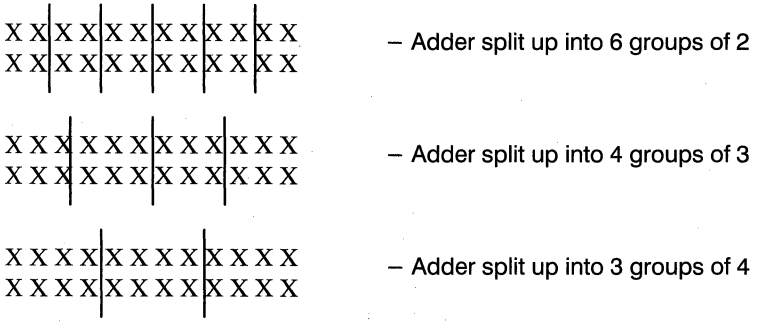


Figure 6. Some Possible Implementations for 12-Bit Carry-Lookahead Adder

FC2ADD12: 12-Bit Full Carry-Lookahead Adder Using a Group-Size of 2 Bits

The FLASH370 CPLD can access up to 16 PTs for each macrocell. The functionality of any signal that has more than 16 PTs is sum-split to fit it into multiple MCs. The number of PTs utilized for signals that sum-split is large and is an undesirable option. With the 2-bit group-size implementation we can accommodate the entire functionality of a 32-bit full carry-lookahead adder without any of the signals getting sum-split. The scheme takes a maximum of three passes through the logic array for all adder sizes up to 32 bits to generate outputs. The various values of Es and Rs, SUM1, SUM0, and C2 are generated in the first pass. All the other intermediate carries are generated in the second pass and the various SUM results are generated in the third pass. A key point to note is that the value of CO is produced in the second pass, even though the various SUM outputs are generated in the third pass only. This makes the component cascadable and modular. Refer to *Table 2* for details on the resource utilization of different 12-bit adder implementations. The FC2ADD12 is built using the ADD2WC and ADD2NC as basic building blocks. The block diagram of a FC2ADD12 is shown in *Figure 7*. The VHDL code for the design is also presented.

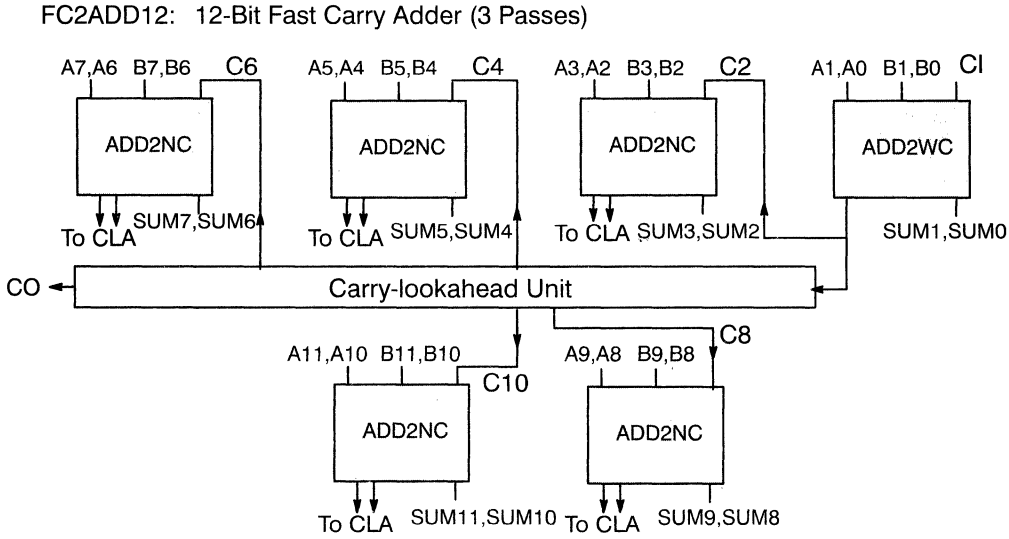


Figure 7. 12-Bit Full Carry-Lookahead Adder Using ADD2WC and ADD2NC

```
--A 12-bit Full carry-lookahead adder built using the ADD2WC and ADD2NC
--elements

USE WORK.RTLPKG.ALL;
USE WORK.ADD2WC.ALL;
USE WORK.ADD2NC.ALL;

ENTITY fc2add12 IS
  PORT(CI : IN BIT;
       A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
       B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
       SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,
```

```
      SUM3,SUM2,SUM1,SUM0 : OUT BIT;
      CO: OUT BIT);
END fc2add12;

ARCHITECTURE archfc2add12 OF fc2add12 IS

  SIGNAL C2, C4, C6, C8, C10 : BIT;
  SIGNAL E1,E2,E3,E4,E5  : BIT;
  SIGNAL R1,R2,R3,R4,R5  : BIT;
  attribute synthesis_off of E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R1,R2,R3,R4,R5 : signal is true;
  attribute synthesis_off of C2, C4, C6, C8, C10 : signal is true;

BEGIN

  i1: add2wc PORT MAP(CI,A1,A0,B1,B0,SUM1,SUM0,C2);
  i2: add2nc PORT MAP(C2,A3,A2,B3,B2,SUM3,SUM2);
  i3: add2nc PORT MAP(C4,A5,A4,B5,B4,SUM5,SUM4);
  i4: add2nc PORT MAP(C6,A7,A6,B7,B6,SUM7,SUM6);
  i5: add2nc PORT MAP(C8,A9,A8,B9,B8,SUM9,SUM8);
  i6: add2nc PORT MAP(C10,A11,A10,B11,B10,SUM11,SUM10);

  E1 <= (A3 AND B3) OR ((A3 OR B3) AND (A2 AND B2));
  R1 <= (A3 OR B3) AND (A2 OR B2);

  C4 <= E1 OR (C2 AND R1);

  E2 <= (A5 AND B5) OR ((A5 OR B5) AND (A4 AND B4));
  R2 <= (A5 OR B5) AND (A4 OR B4);

  C6 <= E2 OR ((E1 OR (C2 AND R1)) AND R2);

  E3 <= (A7 AND B7) OR ((A7 OR B7) AND (A6 AND B6));
  R3 <= (A7 OR B7) AND (A6 OR B6);

  C8 <= E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3);

  E4 <= (A9 AND B9) OR ((A9 OR B9) AND (A8 AND B8));
  R4 <= (A9 OR B9) AND (A8 OR B8);

  C10 <= E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3)) AND
        R4);

  E5 <= (A11 AND B11) OR ((A11 OR B11) AND (A10 AND B10));
  R5 <= (A11 OR B11) AND (A10 OR B10);

  CO <= E5 OR ((E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND
       R3)) AND R4)) AND R5);

END archfc2add12;
```

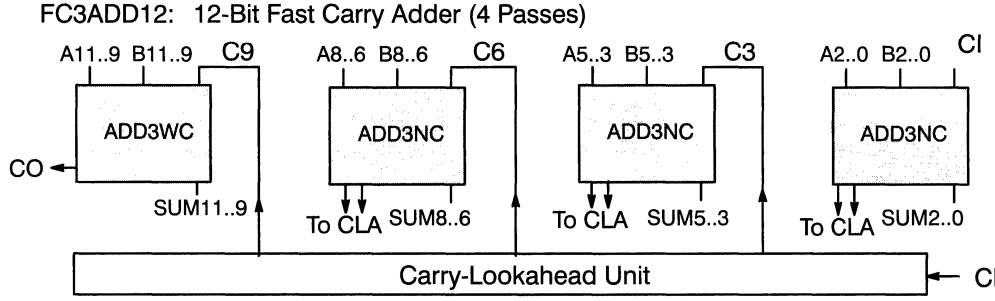


Figure 8. 12-Bit Full Carry-Lookahead Adder using ADD3WC and ADD3NC

FC3ADD12: 12-Bit Full Carry-Lookahead Adder using a Group-Size of 3 Bits

This is very similar to the FC2ADD12, differing in the group-size of the adder used as the basic building block. The basic building blocks in this scheme are the ADD3WC and the ADD3NC components. The VHDL code attached and the block diagram in *Figure 8* illustrate the design. This scheme takes four passes through the logic array to yield all the results. The Es and the Rs are generated in the first pass. The intermediate carries C3, C6, and C9 are generated in the second pass. The carries internal to the group are generated in the third pass and the final SUM outputs in the fourth pass. As a different approach, the CO is generated by the MSB ADD3WC as opposed to the Carry-lookahead unit. This results in CO being generated in the third pass as opposed to the second pass. The VHDL code clearly indicates the manner in which the model is built.

For some bit-sizes, given that the 3-bit group-size is odd-numbered, the designer will have to choose a non-modular structure in building the adder. For example, a 32-bit adder cannot be built using just ADD3NCs and can be built using 10 ADD3NCs and one ADD2NC. The designer needs to choose the final implementation based on the constraints of the application.

```
--12-Bit Fast carry-Lookahead adder with 3-bit groups

USE WORK.ADD3WC.ALL;
USE WORK.ADD3NC.ALL;

ENTITY fc3add12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0 : IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0 : IN BIT;
    CI : IN BIT;
    CO : OUT BIT;
    SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,SUM3,
    SUM2,SUM1,SUM0 : OUT BIT);
END fc3add12;

ARCHITECTURE fc3add12arch OF fc3add12 IS

SIGNAL  E1,E2,E3 : BIT;
SIGNAL  R1,R2,R3  : BIT;
SIGNAL  C3,C6,C9 : BIT;
```

```
attribute synthesis_off of C3,C6,C9 : signal is true;
attribute synthesis_off of E1,E2,E3 : signal is true;
attribute synthesis_off of R1,R2,R3 : signal is true;

BEGIN

i1: add3nc PORT MAP(CI,A2,A1,A0,B2,B1,B0,SUM2,SUM1,SUM0);
i2: add3nc PORT MAP(C3,A5,A4,A3,B5,B4,B3,SUM5,SUM4,SUM3);

i3: add3nc PORT MAP(C6,A8,A7,A6,B8,B7,B6,SUM8,SUM7,SUM6);
i4: add3wc PORT MAP(C9,A11,A10,A9,B11,B10,B9,SUM11,SUM10,SUM9,CO);

E1 <=  (A2 AND B2)
  OR ((A1 AND B1) AND (A2 OR B2))
  OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2));

R1 <=  (A2 OR B2) AND (A1 OR B1) AND (A0 AND B0);

C3 <= E1 OR (R1 AND CI);

E2 <=  (A5 AND B5)
  OR ((A4 AND B4) AND (A5 OR B5))
  OR ((A3 AND B3) AND (A4 OR B4) AND (A5 OR B5));

R2 <=  (A5 OR B5) AND (A4 OR B4) AND (A3 AND B3);

C6 <= E2 OR (E1 AND R2) OR (R2 AND R1 AND CI);

E3 <=  (A8 AND B8)
  OR ((A7 AND B7) AND (A8 OR B8))
  OR ((A6 AND B6) AND (A7 OR B7) AND (A8 OR B8));

R3 <=  (A8 OR B8) AND (A7 OR B7) AND (A6 AND B6);

C9 <= E3 OR (E2 AND R3) OR (E1 AND R3 AND R2) OR (R3 AND R2 AND R1 AND CI);

END fc3add12arch;
```

FC4ADD12: 12-Bit Full Carry-Lookahead Adder using a Group-Size of 4 Bits

This is very similar to the FC2ADD12 and, again, differs in the group-size of the adder used as the basic building block. The basic building block in this scheme is the ADD4NC component. The ADD4NC component is built using a combination of ADD2WC and ADD2NC in the same order. This component is replicated to create the adder of the desired size. In the very last stage, two ADD2WCs are used instead of an ADD2WC and an ADD2NC.

The VHDL code attached and the block diagram in *Figure 9* illustrate the design's functionality. This scheme takes four passes through the logic array to yield results. The various Es and Rs are generated in the first pass, the values of C4 and C8 in the second pass, the outputs from all the ADD2WCs in the third pass, and the outputs from ADD2NC in the fourth pass. Note that the value of CO is generated in the second pass. This scheme uses fewer MCs and more PTs than the previously mentioned schemes. The resource utilization of this model is shown in *Table 2*.

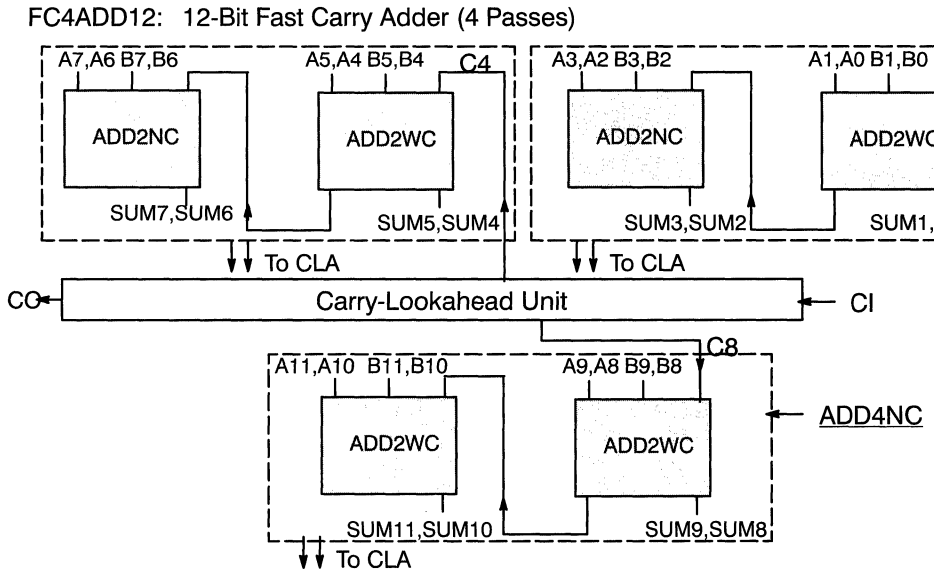


Figure 9. 12-Bit Full Carry-Lookahead Adder using ADD4NC

```
--A 12-bit Full carry-lookahead adder built using the ADD2WC and ADD2NC
--elements. The  ADD2WC and ADD2NC elements are part of the ADD4NC in the
--same order

USE WORK.RTLPKG.ALL;
USE WORK.ADD2WC.ALL;
USE WORK.ADD2NC.ALL;

ENTITY fc4add12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0 : IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0 : IN BIT;
    CI : IN BIT;
    CO : OUT BIT;
    SUM11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,SUM3,
    SUM2,SUM1,SUM0 : OUT BIT);
END fc4add12;

ARCHITECTURE fc4add12arch OF fc4add12 IS

 SIGNAL E1,E2 : BIT;
 SIGNAL R1,R2  : BIT;
 SIGNAL C2,C4,C6,C8,C10 : BIT;

attribute synthesis_off of C2,C4,C6,C8,C10 : signal is true;
attribute synthesis_off of E1,E2 : signal is true;
attribute synthesis_off of R1,R2 : signal is true;
```

```
BEGIN

i1: add2wc PORT MAP(CI,A1,A0,B1,B0,SUM1,SUM0,C2);
i2: add2nc PORT MAP(C2,A3,A2,B3,B2,SUM3,SUM2);

i3: add2wc PORT MAP(C4,A5,A4,B5,B4,SUM5,SUM4,C6);
i4: add2nc PORT MAP(C6,A7,A6,B7,B6,SUM7,SUM6);

i5: add2wc PORT MAP(C8,A9,A8,B9,B8,SUM9,SUM8,C10);
i6: add2wc PORT MAP(C10,A11,A10,B11,B10,SUM11,SUM10,CO);

E1 <=  (A3 AND B3)
   OR ((A2 AND B2) AND (A3 OR B3))
   OR ((A1 AND B1) AND (A2 OR B2) AND (A3 OR B3))
   OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2) AND (A3 OR B3));

R1 <= (A3 OR B3) AND (A2 OR B2) AND (A1 OR B1) AND (A0 AND B0);

C4 <= E1 OR (R1 AND CI);

E2 <=  (A7 AND B7)
   OR ((A6 AND B6) AND (A7 OR B7))
   OR ((A5 AND B5) AND (A6 OR B6) AND (A7 OR B7))
   OR ((A4 AND B4) AND (A5 OR B5) AND (A6 OR B6) AND (A7 OR B7));

R2 <= (A7 OR B7) AND (A6 OR B6) AND (A5 OR B5) AND (A4 AND B4);

C8 <= E2 OR (E1 AND R2) OR (R2 AND R1 AND CI);

END fc4add12arch;
```

Subtracters

Subtracters are just a modified form of adders. The discussion presented for the adders can be easily extended to the subtracters. For any given sized adder or subtracter, the resource utilization is exactly the same for both in all respects.

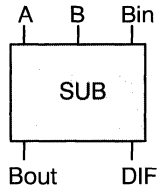
SUB: 1-Bit Full Subtracter

The basic component used in subtracting two operands is called a *Full subtracter*. The full subtracter element will be referred to as the 'SUB' component.

The block diagram and functionality of SUB is shown in *Figure 10*. A (minuend) and B (subtrahend) are the two operands to be subtracted and Bin is the Borrow-in to the component. DIF and Bout are the Difference and Borrow-out from the component.

The VHDL code describing the functionality of the SUB component is shown here. This design takes one pass through the Product Term Matrix logic array to fit into a FLASH370 CPLD. The SUB component instantiated in the VHDL code has the exact same functionality shown in *Figure 10*.

SUB: 1-Bit Full Subtracter (1 Pass)



(Basic building block)

Functionality: DIF = *NOT* (*NOT* (A *XOR* B) *XOR* Bin)
Bout = (*NOT* A *AND* B) OR (*NOT* A *AND* CI) or (B *AND* CI)

Figure 10. Block Diagram and Functionality of a Full Subtracter

```
-- This VHDL code invokes the
MATHPKG element SUB

USE WORK.CYPRESS.ALL;
USE WORK.MATHPKG.ALL;

ENTITY sub IS
 PORT (Bin: IN BIT;
   A, B: IN BIT;
   DIF: OUT BIT;
    Bout: OUT BIT);
END sub;

ARCHITECTURE archsub OF sub IS

BEGIN

 i1: sub PORT
MAP(Bin,A,B,DIF,Bout);

END archsub;
```

SUB2WB: A 2-Bit Subtracter with a Borrow-Out

The structure of a 2-bit group subtracter (SUB2WB) is very similar to that of the ADD2WC and is shown here. This component can be used as a building block to build larger sized subtracters, exactly like ADD2WC was used to build larger sized adders. The block diagram of the SUB2WB is shown in *Figure 11*. The corresponding VHDL code used to describe the functionality of the SUB2WB is also attached. As in the case of ADD2WC, the functionality for SUB2WB is realized in one pass through the logic array.

SUB2: 2-Bit Adder (1 Pass)

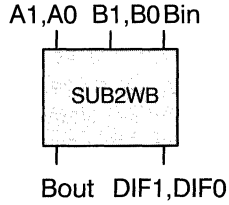


Figure 11. Block Diagram of a 2-Bit Subtracter with a Borrow-Out

```
USE WORK.RTLPKG.ALL;

PACKAGE sub2wb_pkg IS
  COMPONENT sub2wb PORT(
    Bin : IN BIT;
      A1,A0: IN BIT;
      B1,B0: IN BIT;
      DIF1,DIF0 : OUT BIT;
      Bout: OUT BIT);
  END COMPONENT;
END sub2wb_pkg;
```

```
ENTITY sub2wb IS
  PORT(Bin : IN BIT;
    A1,A0: IN BIT;
    B1,B0: IN BIT;
    DIF1,DIF0 : OUT BIT;
    Bout: OUT BIT);
END sub2wb;

ARCHITECTURE archsub2wb OF sub2wb IS

BEGIN

DIF0 <= NOT (NOT (A0 XOR B0) XOR Bin);
DIF1 <= NOT (NOT (A1 XOR B1) XOR ((NOT A0 AND B0) OR (NOT A0 AND Bin) OR
            (B0 AND Bin)));

Bout   <= (NOT A0 AND B0 AND B1)
  OR (NOT A0 AND B0 AND NOT A1)
  OR (BI AND B0 AND B1)
  OR (BI AND B0 AND NOT A1)
  OR (BI AND NOT A0 AND B1)
  OR (BI AND NOT A0 AND NOT A1)
  OR (NOT A1 AND B1);

END archsub2wb;
```

FB2SUB12 : 12-Bit Full Borrow-Lookahead Subtracter using 2-Bit Subtracters

It was mentioned before that we can build equivalent subtracter models for all the adder models discussed earlier. The functionality and the implementation of an FB2SUB12 (subtracter equivalent of an FC2ADD12) is shown here as an example. The implementation of all the possible subtracter elements is not discussed in this application note, since the concept involved in building them is identical to that of the adders.

The block diagram of the FB2SUB12 is very similar to that of the adder element FC2ADD12 and is shown in *Figure 12*. The FB2SUB12 is built using the basic elements SUB2WB and SUB2NC (2-bit subtracter with no borrow-out). This takes three passes through the logic array. The values of the various Es and Rs are generated in the first pass, the intermediate carries (borrows) in the second pass, and the various DIFs in the third pass. Note that the value of BO is generated in the second pass. The VHDL code for FB2SUB12 is also attached.

FBSUB12: 12-Bit Fast Borrow Subtracter (3 Passes)

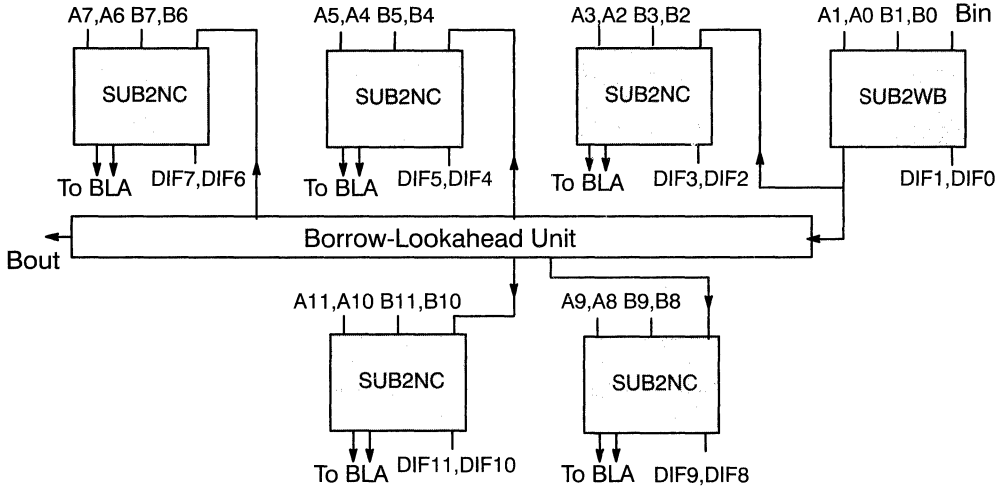


Figure 12. 12-Bit Fast Borrow Subtracter Built using SUB2WB and SUB2NC

```
--A 12-bit Full borrow-lookahead subtracter built using the SUB2WC and
--SUB2NC elements

USE WORK.RTLPKG.ALL;
USE WORK.SUB2WB.ALL;
USE WORK.SUB2NC.ALL;

ENTITY fb2sub12 IS
  PORT (Bin : IN BIT;
       A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
       B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
       DIF11,DIF10,DIF9,DIF8,DIF7,DIF6,DIF5,DIF4,
       DIF3,DIF2,DIF1,DIF0 : OUT BIT;
    Bout: OUT BIT);
END fb2sub12;

ARCHITECTURE archfb2sb12 OF fb2sub12 IS

  SIGNAL C2, C4, C6, C8, C10 : BIT;
  SIGNAL E1,E2,E3,E4,E5  : BIT;
  SIGNAL R1,R2,R3,R4,R5  : BIT;

--The internal carries are referred to as C's to distinguish between
--borrow-out's and the operands
  attribute synthesis_off of E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R1,R2,R3,R4,R5 : signal is true;
```

```
  attribute synthesis_off of C2, C4, C6, C8, C10 : signal is true;

BEGIN

  i1: sub2wb PORT MAP(Bin,A1,A0,B1,B0,DIF1,DIF0,C2);
  i2: sub2nc PORT MAP(C2,A3,A2,B3,B2,DIF3,DIF2);
  i3: sub2nc PORT MAP(C4,A5,A4,B5,B4,DIF5,DIF4);
  i4: sub2nc PORT MAP(C6,A7,A6,B7,B6,DIF7,DIF6);
  i5: sub2nc PORT MAP(C8,A9,A8,B9,B8,DIF9,DIF8);
  i6: sub2nc PORT MAP(C10,A11,A10,B11,B10,DIF11,DIF10);

    E1 <= (NOT A3 AND B3) OR ((NOT A3 OR B3) AND (NOT A2 AND B2));
    R1 <= (NOT A3 OR B3) AND (NOT A2 OR B2);

    C4 <= E1 OR (C2 AND R1);

    E2 <= (NOT A5 AND B5) OR ((NOT A5 OR B5) AND (NOT A4 AND B4));
    R2 <= (NOT A5 OR B5) AND (NOT A4 OR B4);

    C6 <= E2 OR ((E1 OR (C2 AND R1)) AND R2);

    E3 <= (NOT A7 AND B7) OR ((NOT A7 OR B7) AND (NOT A6 AND B6));
    R3 <= (NOT A7 OR B7) AND (NOT A6 OR B6);

    C8 <= E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3);

    E4 <= (NOT A9 AND B9) OR ((NOT A9 OR B9) AND (NOT A8 AND B8));
    R4 <= (NOT A9 OR B9) AND (NOT A8 OR B8);

    C10 <= E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2)) AND R3))
               AND R4);

    E5 <= (NOT A11 AND B11) OR ((NOT A11 OR B11) AND (NOT A10 AND B10));
    R5 <= (NOT A11 OR B11) AND (NOT A10 OR B10);

    Bouy <= E5 OR ((E4 OR ((E3 OR ((E2 OR ((E1 OR (C2 AND R1)) AND R2))
            AND R3)) AND R4)) AND R5);

END archfb2sub12;
```

Comparison of Resource Utilization for Different Schemes in Building a 12-Bit Adder

A comparison chart showing the resource utilization for the different models that can be used in building a 12-bit adder is shown in *Table 2*. This table summarizes some of the key issues that have been presented in the discussion so far. Some comparisons and comments from the charts and are listed here:

Ripple Carry Adders

1. For a given group-size, the number of passes taken to yield results is *dependent* on the size of the adder being built.

Table 2. Comparison of Different 12-Bit Adder Schemes

Resource	R1ADD12	R2ADD12	R3ADD12	FC2ADD12	FC3ADD12	FC4ADD12
PTs used	84	138	165	148	153	169
MCs used	24	18	16	28	26	22
# of passes	12	6	5	3	4	4

2. As the group-size increases, the number of passes taken through the logic array is *(n/k) −1 + # of passes for final stage*, where *n* is the size of the adder and *k* is the group size. For example, a R2ADD12 takes *(12/2) −1 + 1* = 6 passes to yield the desired result.

3. In the R3ADD12 (ripple carry adder built using 3-bit groups) scheme, the value of the MSB sum bit within a 3-bit group is produced only in the second pass through the array. This, however, does not affect the 12-bit adder yielding results in 5 passes *(12/3) −1 + 2 = 5)* as expected. This is possible because the carry-out from the 3-bit group is produced in the first pass. The implementation of the ADD3WC was discussed in detail earlier. This solution is a very desirable solution for most applications that use small sized adders.

4. The R1ADD12 uses fewer PTs and more MCs among the different versions of ripple-carry adders. The opposite is the case for the R3ADD12. The R2ADD12 provides an intermediate solution between the two extremes.

5. The macrocell count in R1ADD12 can be brought down from 24 to 18, if the attribute 'synthesis_off' is used on the even-numbered carries only. The number of passes is also brought down from 12 to 6. This, however, pushes the product term count from 84 to 138. In either case, none of the equations sum-split. This is, in fact, R2ADD12. The designer can choose the implementation that best chooses the application.

6. The R4ADD12 (ripple carry adder built using 4-bit groups) is not a viable solution, since the carry-out from one of the 4-bit groups would take two passes to be generated. This results in a implementation that takes six passes to yield results as opposed to the expected three passes. This solution is inefficient and is not considered.

Carry-Lookahead Adders

1. For a given group-size, the number of passes taken to yield results is *largely independent* of the size of the adder being built. This is the biggest advantage with carry-lookahead adders.

2. All the *group generates* (*Es*) and *group propagates* (*Rs*) are generated in the first pass and the carry-ins to all groups in the second pass through the logic array. The Sum outputs are generated in the third or the fourth pass, depending on the group-size being used.

3. The FC2ADD12 takes three passes to complete, and four passes for the FC3ADD12 and FC4ADD12. The number of passes *remains the same* up to 32-bit versions of the adder.

4. Similar to the ripple carry adders, the FC2ADD12 uses fewer PTs and more MCs among the different versions of carry-lookahead adders. The opposite is the case for the FC4ADD12. The FC3ADD12 provides an intermediate solution between the two extremes.

5. The FC5ADD12 (carry-lookahead adder built using 5-bit groups) is not a viable solution, since the extra number of PTs and number of passes (5) taken through the logic array do not justify its usage. The design is also not modular and difficult to deal with. A designer can, however, extend the discussion presented to build his own FC5ADD12 model if the application demands it. This, however, would be an extreme case and is not presented.

Summary

Comparing ripple carry and carry-lookahead adders, it is evident that ripple carry adders are area efficient but have poor speed performance. The carry-lookahead adders on the other hand are faster but utilize more resources. Given the different choices, the user needs to make a careful selection of the scheme best suited for his application.

Large-Sized Adders/Subtracters

As PLDs have grown in size and speed over the past few years, a lot of designers have been pushing towards larger sized adders and subtracters. A lot of focus and time has been dedicated towards building efficient smaller sized elements and it was left to the designer's discretion to build and implement the algorithm. Cypress believes in providing designers with the best possible implementation for their adders/subtracters and relieve them of the problems they would normally face. This provides the customer with the opportunity to get the best implementation for their application without spending a lot of time.

Table 1 talks about the resource utilization for 24-bit and 32-bit adders using 2-bit, 3-bit, and 4-bit group-sizes with carry/borrow-lookahead principle. In the previous sections, different implementation strategies and the VHDL codes for a 12-bit full-carry-lookahead adder were shown as an example. The VHDL codes for most variations of the 24- and 32-bit implementations are not presented here due to space constraints. The codes are provided, however, as a part of the tutorial section in the *Warp* VHDL compiler. *Figure 12* illustrates three schemes used in implementing a 24-bit adder. The VHDL code for a 24-bit carry-lookahead adder with a 4-bit group size is shown here as an example. The code for other models is very similar and can be easily extrapolated.

Table 1. Comparison of Different 24-Bit and 32-Bit Adder Schemes

Resource	FC2ADD24	FC3ADD24	FC4ADD24	FC2ADD32	FC3ADD32	FC4ADD32
PTs used	272	314	359	393	427	488
MCs used	58	54	46	78	73	62
# of passes	3	4	4	3	4	4

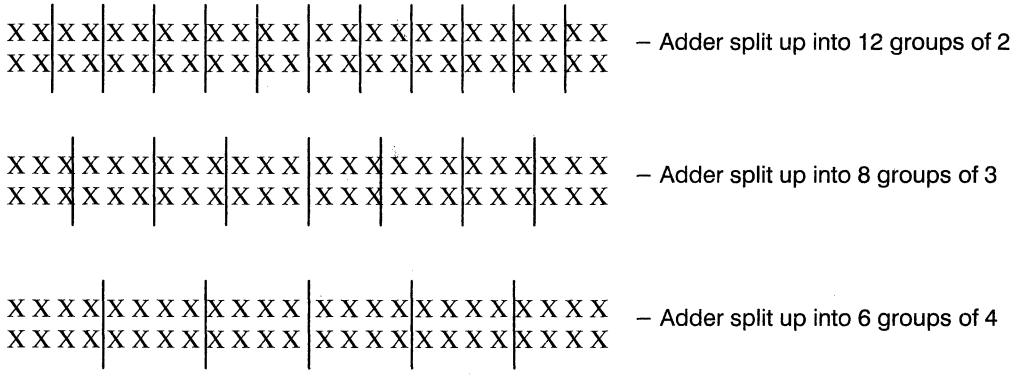


Figure 13. Three Different Carry-Lookahead Schemes to Implement a 24-Bit Adder

```
--24-bit Fast Carry lookahead adder with 4-bit groups

USE work.add2wc_pkg.all;
USE work.add2nc_pkg.all;

ENTITY fc4add24 IS
PORT (

A23,A22,A21,A20,A19,A18,A17,A16,A15,A14,A13,A12,
A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0 : IN BIT;
B23,B22,B21,B20,B19,B18,B17,B16,B15,B14,B13,B12,
B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0 : IN BIT;
CI : IN BIT;
CO : OUT BIT;
SUM23,SUM22,SUM21,SUM20,SUM19,SUM18,SUM17,SUM16,SUM15,SUM14,SUM13,SUM12,SUM
11,SUM10,SUM9,SUM8,SUM7,SUM6,SUM5,SUM4,SUM3,SUM2,SUM1,SUM0: OUT BIT);

END fc4add24;

ARCHITECTURE fc4add24arch OF fc4add24 IS

SIGNAL  E1,E2,E3,E4,E5 : BIT;
SIGNAL  R1,R2,R3,R4,R5  : BIT;
SIGNAL  C2,C4,C6,C8,C10,C12,C14,C16,C18,C20,C22 : BIT;

attribute synthesis_off of C2,C4,C6,C8,C10,C12,C14,C16,C18,C20,C22 : signal
is true;
attribute synthesis_off of E1,E2,E3,E4,E5 : signal is true;
attribute synthesis_off of R1,R2,R3,R4,R5 : signal is true;

BEGIN

i1: add2wc PORT MAP (CI,A1,A0,B1,B0,SUM1,SUM0,C2);
i2: add2nc PORT MAP (C2,A3,A2,B3,B2,SUM3,SUM2);

i3: add2wc PORT MAP (C4,A5,A4,B5,B4,SUM5,SUM4,C6);
i4: add2nc PORT MAP (C6,A7,A6,B7,B6,SUM7,SUM6);

i5: add2wc PORT MAP (C8,A9,A8,B9,B8,SUM9,SUM8,C10);
i6: add2nc PORT MAP (C10,A11,A10,B11,B10,SUM11,SUM10);

i7: add2wc PORT MAP (C12,A13,A12,B13,B12,SUM13,SUM12,C14);
i8: add2nc PORT MAP (C14,A15,A14,B15,B14,SUM15,SUM14);

i9: add2wc PORT MAP (C16,A17,A16,B17,B16,SUM17,SUM16,C18);
i10: add2nc PORT MAP (C18,A19,A18,B19,B18,SUM19,SUM18);

i11: add2wc PORT MAP (C20,A21,A20,B21,B20,SUM21,SUM20,C22);
i12: add2wc PORT MAP (C22,A23,A22,B23,B22,SUM23,SUM22,Co);
```

```
E1 <=  (A3 AND B3)
  OR ((A2 AND B2) AND (A3 OR B3))
  OR ((A1 AND B1) AND (A2 OR B2) AND (A3 OR B3))
  OR ((A0 AND B0) AND (A1 OR B1) AND (A2 OR B2) AND (A3 OR B3));

R1 <= (A3 OR B3) AND (A2 OR B2) AND (A1 OR B1) AND (A0 AND B0);

C4 <= E1 OR (R1 AND CI);

E2 <=  (A7 AND B7)
  OR ((A6 AND B6) AND (A7 OR B7))
  OR ((A5 AND B5) AND (A6 OR B6) AND (A7 OR B7))
  OR ((A4 AND B4) AND (A5 OR B5) AND (A6 OR B6) AND (A7 OR B7));

R2 <= (A7 OR B7) AND (A6 OR B6) AND (A5 OR B5) AND (A4 AND B4);

C8 <= E2 OR (E1 AND R2) OR (R2 AND R1 AND CI);

E3 <=  (A11 AND B11)
  OR ((A10 AND B10) AND (A11 OR B11))
  OR ((A9 AND B9) AND (A10 OR B10) AND (A11 OR B11))
  OR ((A8 AND B8) AND (A9 OR B9) AND (A10 OR B10) AND (A11 OR B11));

R3 <= (A11 OR B11) AND (A10 OR B10) AND (A9 OR B9) AND (A8 AND B8);

C12 <= E3 OR (E2 AND R3) OR (E1 AND R3 AND R2)  OR (R3 AND R2 AND R1 AND
CI);

E4 <=  (A15 AND B15)
  OR ((A14 AND B14) AND (A15 OR B15))
  OR ((A13 AND B13) AND (A14 OR B14) AND (A15 OR B15))
  OR ((A12 AND B12) AND (A13 OR B13) AND (A14 OR B14) AND (A15 OR B15));

R4 <= (A15 OR B15) AND (A14 OR B14) AND (A13 OR B13) AND (A12 AND B12);

C16 <= E4 OR (E3 AND R4) OR (E2 AND R4 AND R3) OR (E1 AND R4 AND R3 AND R2)
  OR (R3 AND R2 AND R1 AND CI);

E5 <=  (A19 AND B19)
  OR ((A18 AND B18) AND (A19 OR B19))
  OR ((A17 AND B17) AND (A18 OR B18) AND (A19 OR B19))
  OR ((A16 AND B16) AND (A17 OR B17) AND (A18 OR B18) AND (A19 OR B19));
R5 <= (A19 OR B19) AND (A18 OR B18) AND (A17 OR B17) AND (A16 AND B16);

C20 <= E5 OR (E4 AND R5) OR (E3 AND R5 AND R4) OR (E2  AND R5  AND R4 AND
  R3) OR (E1 AND R5 AND R4 AND R3 AND R2) OR (R5 AND R4 AND R3 AND R2 AND
  R1 AND CI);

END fc4add24arch;
```

Equality Comparators

Equality comparators are used often to compare the value of two operands. Equality comparators are built using the Exclusive-OR gate as the building block. A bit-wise comparison of the two data streams is done using XOR gates and each of the individual results are OR-ed together to obtain the final result.

EQCOMP4: 4-Bit Equality Comparator

The EQCOMP4 is a 4-bit equality compare element. The model can be described as:

```
EQ = NOT ((A3 XOR B3)
   OR (A2 XOR B2)
   OR (A1 XOR B1)
   OR (A0 XOR B0))
```

This implementation takes 8 PTs. *Figure 14* shows the block diagram for EQCOMP4. NEQCOMP4 is the 4-bit non-equality comparator. The EQCOMP4 is implemented as an inverted version of the NEQCOMP4. The NEQCOMP4 element takes 8 PTs and the EQCOMP4 takes 16 PTs. The FLASH370 CPLD has a polarity control in the macrocell and can create the EQCOMP4 element using the NEQCOMP4 element, resulting in a implementation with a reduced product term count.

The equality comparator for all bit sizes greater than 8 takes more than 16 PTs to produce the result and takes two passes, since the FLASH370 CPLD architecture takes in a maximum of 16 PTs into one macrocell.

EQCOMP24: 24-Bit Equality Comparator

The EQCOMP24 uses three EQCOMP8s in parallel and combines the results of the three components to produce the result. This takes two passes through the logic array, 4 MCs, and 49 PTs. The block diagram of this model is shown in *Figure 15*.

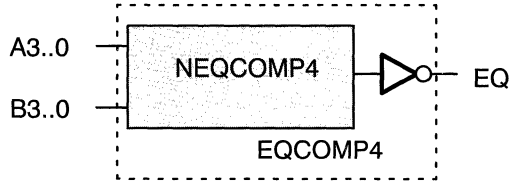


Figure 14. Block Diagram of a 4-Bit Equality Compare

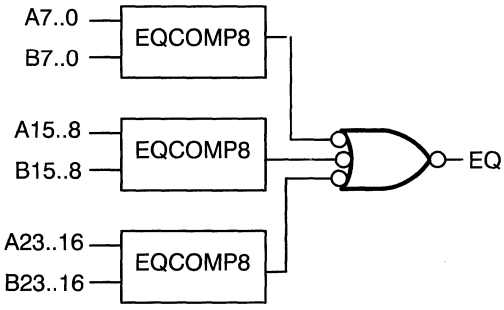


Figure 15. Block Diagram of a 24-Bit Equality Compare

Magnitude Comparators

Magnitude comparators are also widely used in the industry in comparing values of two operands. The magnitude comparators provide information if a signal is greater than (>) , or less than (<) another signal of the same length.

MAGCOMP8: 8-Bit Magnitude Comparator

This is the generic implementation of a magnitude comparator and does a bit-wise comparison, similar to that of the equality comparison. However, in the case of a magnitude comparator the results of a bit-wise comparison are to be retained and passed onto the succeeding set of bits. This passage of information continues and tends to increase the resource utilization of the design exponentially.

The VHDL implementation of an 8-bit magnitude comparator is shown here. The design takes 255 PTs and fits in two passes through the logic array. The block diagram of MAGCOMP8 is shown in *Figure 16.*

Figure 16. Block Diagram of an 8-Bit Magnitude Compare

```
-- Flattened version of the Magni-
tude comparator

USE work.int_math.all;

ENTITY magcomp IS
PORT (
A,B : IN BIT_VECTOR(7 DOWNTO 0);
MAG : OUT BIT);

END magcomp;

ARCHITECTURE magarch OF magcomp IS

BEGIN

MAG <= '1' WHEN (A < B) ELSE '0';

END magarch;
```

A fully flattened implementation of a magnitude comparator would take $(2^n - 1)$ PTs to implement. It is, however, not recommended to use the fully-flattened version of the magnitude comparator for any bit-size greater than 4 bits. This is to ensure that there is no sum-splitting involved in the equations. There are other means to achieve better results and the best scheme is presented next.

FB2MGCMP8: 8-Bit Borrow-Lookahead Magnitude Comparator

The block diagram of a 8-bit magnitude compare is shown in *Figure 17.*

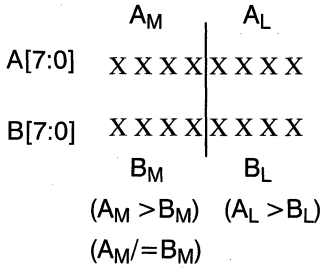
Figure 17. Block Diagram of an 8-Bit Magnitude Compare

This scheme uses a different approach to compare the magnitudes of two binary bit vectors. As an example, the scheme is illustrated for a 8-bit magnitude comparator. The 4 MSB bits of the bit vectors A[7:0] and B[7:0] are called A_M and B_M, respectively. Similarly, the 4 LSB bits are referred to as A_L and B_L respectively. The bit vector A is greater than B if $(A_M > B_M)$ or if $(A_M = B_M)$ and $(A_l > B_L)$.

It is evident from the set of equations in *Figure 18* that the magnitude comparison of two binary bit vectors can be done by evaluating the values of G_M, G_L and P_M. G_M and G_L are the *generate* functions for the MSHalf (most significant half) and the LSHalf (least significant half) for the two bit vectors and P_M is the *propagate* function for the MSHalf. This scheme is a stripped down version of the borrow-lookahead scheme used to build fast subtractors. In this implementation we need to determine the values of the *generate* and *propagate* functions for the bit vectors and need not produce any of the difference results. The borrow-out signal determines the output of the magnitude comparison. If the borrow-out is a '1' then $(A < B)$, else $(A \geq B)$.

This scheme allows for a fast and efficient means to do magnitude comparisons. Magnitude Comparators up to 32 bits can be built to produce the result in just 2 passes. The number of PTs used is also substantially less than the 'flattened' implementation of the magnitude comparators.

The discussion presented earlier on group-sizes can also be extended here. The group-size over which the *propagate* and *generate* functions are generated can be varied to be 2, 3 or 4. In *all* cases the design takes 2 passes to produce the desired result. The various values of Es and Rs are generated in the first



$$(A > B) = (A_M > B_M) + \overline{((A_M /= B_M)} * (A_L > B_L))$$

$$G_M \qquad P_M \qquad G_L$$

$$(A > B) = G_M + P_M * G_L$$

Figure 18. Bit Vector Magnitude Comparison Equations

pass and the value of the borrow-out in the second pass. However, there is a trade-off between the number of PTs and MCs used among the different group-sizes chosen. A comparison between these different implementations is discussed later.

The number of PTs used to implement the P_M (propagate) function can be halved if 'OR' gates are used instead of 'XOR' gates. This was mentioned earlier in the discussion on carry-lookahead. This extension makes the implementation of the borrow-lookahead magnitude comparator fast and efficient.

Comparison of Two Implementations of a 12-Bit Magnitude Compare

Two different implementations of a 12-bit magnitude comparator are shown here. The first implementation is an extension of MAGCOMP4. The second implementation uses the borrow-lookahead scheme and is built using borrow-lookahead over a group-size of 2 bits. This comparison illustrates the advantage of using FB2MGCMP12 over the simple MAGCOMP12.

The block diagram of MAGCOMP12 is shown in *Figure 19*. The flattened version of MAGCOMP12 takes $(2^{12} - 1)$ PTs. This is a large amount of logic and will not fit into any of the FLASH370 CPLDs.

Figure 19. Block Diagram of a 12-Bit Magnitude Compare

The MAGCOMP12 with the synthesis_off attribute on the intermediate signals uses 44 unique PTs, but is very slow and takes 11 passes through the array.

The block diagram of FB2MGCMP12 is shown in *Figure 20*. The VHDL code for this design is also shown here. This design takes just two passes through the array and uses 36 unique PTs. The various values of Es and Rs are generated in the first pass and the value of the borrow-out in the second pass. Each of the Es uses 3 PTs and Rs 2 PTs and the output MAG takes 6 PTs. This is clearly a much better implementation than the MAGCOMP12.

Figure 20. Block Diagram of a 12-Bit Magnitude Compare with Borrow-Lookahead

```
--The borrow-lookahead principle using 2-bit groups was used to build this
--element

USE WORK.RTLPKG.ALL;

ENTITY fb2mgcmp12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
      MAG: OUT BIT);
END fb2mgcmp12;

ARCHITECTURE archfb2mgcmp12 OF fb2mgcmp12 IS

  SIGNAL E0,E1,E2,E3,E4,E5  : BIT;
  SIGNAL R0,R1,R2,R3,R4,R5  : BIT;
  SIGNAL BO : BIT;

  attribute synthesis_off of E0,E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R0,R1,R2,R3,R4,R5 : signal is true;
```

```
BEGIN

    E0 <= (NOT A1 AND B1) OR ((NOT A1 OR B1) AND (NOT A0 AND B0));
    R0 <= (NOT A1 OR B1) AND (NOT A0 OR B0);

    E1 <= (NOT A3 AND B3) OR ((NOT A3 OR B3) AND (NOT A2 AND B2));
    R1 <= (NOT A3 OR B3) AND (NOT A2 OR B2);

    E2 <= (NOT A5 AND B5) OR ((NOT A5 OR B5) and (NOT A4 AND B4));
    R2 <= (NOT A5 OR B5) AND (NOT A4 OR B4);

    E3 <= (NOT A7 AND B7) OR ((NOT A7 OR B7) AND (NOT A6 AND B6));
    R3 <= (NOT A7 OR B7) AND (NOT A6 OR B6);

    E4 <= (NOT A9 AND B9) OR ((NOT A9 OR B9) AND (NOT A8 AND B8));
    R4 <= (NOT A9 OR B9) AND (NOT A8 OR B8);

    E5 <= (NOT A11 AND B11) OR ((NOT A11 OR B11) AND (NOT A10 AND B10));
      R5 <= (NOT A11 OR B11) AND (NOT A10 OR B10);

    BO <= E5 OR
          (R5 AND E4) OR
          (R5 AND R4 AND E3) OR
          (R5 AND R4 AND R3 AND E2) OR
          (R5 AND R4 AND R3 AND R2 AND E1) OR
          (R5 AND R4 AND R3 AND R2 AND R1 AND E0);

MAG <= '1' WHEN (BO = '1') ELSE '0';

--MAG is a '1' if B > A

END archfb2mgcmp12;
```

A comparison between 2-, 3-, and 4-bit group sized implementation of a 12-bit magnitude comparator based on the borrow-lookahead scheme is shown in *Table 3*. As mentioned before, the number of passes through the logic array is the same for all group-bit-sizes. The number of PTs and MCs used vary as shown in the table. The user has a wide choice and needs to choose the right group-size depending on the application.

Table 2. Comparison of a 12-Bit Magnitude Compare between Different Group-Sizes

Group-Bit-Size	2	3	4
# of PTs	34	44	60
# of MCs	13	9	7
# of passes	2	2	2

Three-Output Comparators

The discussion on magnitude comparators has so far been restricted to the values of less than (<) and greater than or equal to (≥) only. The discussion in this section talks about producing all three outputs, namely '<', '>' and '='.

FB2EQMCMP12: 12-Bit Borrow-Lookahead Three-Output Magnitude Comparator Using 2-Bit Groups

This model combines all the concepts discussed in the magnitude comparator section into one design. This uses borrow-lookahead, 2-bit groups, and also produces three outputs. The block diagram of this model is shown in *Figure 21*.

There are two ways in which the Borrow-lookahead principle can be used to achieve the functionality of a three-output comparator.

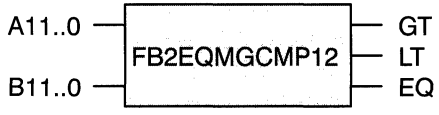


Figure 21. Block Diagram of a 12-Bit Borrow-Lookahead Three-Output Magnitude Compare

1. Use two passes for 'A < B' and 'A = B' each, then use a third pass for A > B using the results from A < B and A = B. This uses 62 PTs. The EQCOMP12 required for this model is built using three EQCOMP4s similar to the block diagram shown in *Figure 15*. The EQCOMP12 can also be built using four EQCOMPs, or two EQCOMP6s, or an EQCOMP8 and an EQCOMP4 or any other combination. As long as the EQCOMP model chosen does not sum-split, the value of EQCOMP12 can be realized in two passes using 25 PTs.

2. Use two passes to generate all three outputs. In this implementation a set of Es and Rs is required to create a value of LT (A – B). A second set of Es and Rs is required to obtain the value of GT (B – A). The value of EQ is also produced in 2 passes along with GT and LT. This scheme uses 97 PTs.

The first scheme is area efficient, but takes three passes though the logic array to generate the final results. The VHDL implementation for the first scheme is presented here. It is very easy to extrapolate the code for the second scheme.

```
--This VHDL code describes the implementation of a 3-output magnitude
--comparator. The borrow-lookahead principle using 2-bit groups was used
--to build this element

USE WORK.RTLPKG.ALL;

ENTITY fb2eqmgcmp12 IS
  PORT (
    A11,A10,A9,A8,A7,A6,A5,A4,A3,A2,A1,A0: IN BIT;
    B11,B10,B9,B8,B7,B6,B5,B4,B3,B2,B1,B0: IN BIT;
    EQ,LT,GT: OUT BIT);
END fb2eqmgcmp12;

ARCHITECTURE archfb2eqmgcmp12 OF fb2mgeqcmp12 IS

  SIGNAL E0,E1,E2,E3,E4,E5   : BIT;
  SIGNAL R0,R1,R2,R3,R4,R5   : BIT;
  SIGNAL X11,X10,X9,X8,X7,X6,X5,X4,X3,X2,X1,X0 : BIT;
  SIGNAL INT1, INT2, INT3: BIT;
  SIGNAL BO : BIT;
```

```
  attribute synthesis_off of E0,E1,E2,E3,E4,E5 : signal is true;
  attribute synthesis_off of R0,R1,R2,R3,R4,R5 : signal is true;
  attribute synthesis_off of INT1, INT2, INT3 : signal is true;

BEGIN

    E0 <= (NOT A1 AND B1) OR ((NOT A1 OR B1) AND (NOT A0 AND B0));
      R0 <= (NOT A1 OR B1) AND (NOT A0 OR B0);

    E1 <= (NOT A3 AND B3) OR ((NOT A3 OR B3) AND (NOT A2 AND B2));
    R1 <= (NOT A3 OR B3) AND (NOT A2 OR B2);
    E2 <= (NOT A5 AND B5) OR ((NOT A5 OR B5) and (NOT A4 AND B4));
    R2 <= (NOT A5 OR B5) AND (NOT A4 OR B4);

    E3 <= (NOT A7 AND B7) OR ((NOT A7 OR B7) AND (NOT A6 AND B6));
    R3 <= (NOT A7 OR B7) AND (NOT A6 OR B6);

    E4 <= (NOT A9 AND B9) OR ((NOT A9 OR B9) AND (NOT A8 AND B8));
    R4 <= (NOT A9 OR B9) AND (NOT A8 OR B8);

    E5 <= (NOT A11 AND B11) OR ((NOT A11 OR B11) AND (NOT A10 AND B10));
    R5 <= (NOT A11 OR B11) AND (NOT A10 OR B10);

    BO <= E5 OR
          (E4 AND R5) OR
          (E3 AND R5 AND R4) OR
          (E2 AND R5 AND R4 AND R3) OR
          (E1 AND R5 AND R4 AND R3 AND R2) OR
          (E0 AND R5 AND R4 AND R3 AND R2 AND R1);

LT <= '1' WHEN (BO = '1') ELSE '0';

-- LT is a '1' if A < B

GT <= '1' WHEN (LT = '0' AND EQ = '0' ) ELSE '0';

-- GT is a '1' if A > B

    X11 <= A11 XOR B11;
    X10 <= A10 XOR B10;
    X9 <= A9 XOR B9;
    X8 <= A8 XOR B8;
    X7 <= A7 XOR B7;
    X6 <= A6 XOR B6;
    X5 <= A5 XOR B5;
    X4 <= A4 XOR B4;
    X3 <= A3 XOR B3;
    X2 <= A2 XOR B2;
    X1 <= A1 XOR B1;
    X0 <= A0 XOR B0;
```

```
    INT1 <= (X11 OR X10 OR X9 OR X8);
    INT2 <= (X7 OR X6 OR X5 OR X4);
    INT3 <= (X3 OR X2 OR X1 OR X0);

    EQ <= NOT (INT1 OR INT2 OR INT3);

END archfb2eqmgcmp12;
```

Summary

A number of arithmetic elements frequently used in various applications were presented in this application note. The underlying concepts and the final implementations for all these models were also presented. Designs created with an understanding of the target architecture always perform better than generic designs. The LPM elements available in *Warp* are all geared towards obtaining the best performance, both in speed and area, for CPLDs. The concepts and implementations presented in this application note are used to build the various LPM elements. Understanding this application note will enable the user to understand the LPM elements better and exploit their availability in the best possible manner.

CPLDs are getting to be very popular with the programmable logic industry, and are widely used in DSP applications, PCs, Motherboards, Data Communication equipment, Multimedia, Instrumentation. etc. They have many advantages over other programmable logic devices. A few key advantages are listed here:

- Ease of use—Simple extension of AND-OR structure of small PLDs like 22V10
- Predictable timing model
- No fanout penalty
- Provide high speed of operation
- Off the shelf availability
- Cost effective solution

These advantages make CPLDs an ideal platform to implement high-performance arithmetic circuits in a cost-effective manner.

FPGAs inherently have more useable gates than CPLDs and also provide a very fine grain architecture. The major constraints to deal with FPGAs are I/O utilization, logic utilization, and timing. A particular design can be literally placed in many different places in an FPGA because of its fine grain architecture. In CPLDs the structure is very coarse grained and this pushes the number of constraints higher. The typical constraints to deal with arithmetic designs in CPLDs are product term count, macrocell count, number of inputs into a logic block, product term and macrocell placement, number of passes through logic array, and sum-splits. All of these facts make designing arithmetic operations with CPLDs a tougher task. Understanding the structure and capabilities of CPLDs is absolutely essential in creating efficient designs.

With the background provided in this application note, a designer should be able to create any algorithm or implementation for an arithmetic application. The user is strongly encouraged to read the VHDL textbook written by the PLD applications group to get a good grasp of VHDL and using it to implement efficient designs in CPLDs and FPGAs.

Design Considerations for On-Board Programming of the CY7C374 and CY7C375

If on-board reprogrammability is a must for your design, certain considerations must be met before the design is completed and before the board is laid out. The first step in setting up a board for in-circuit programming is to know which pins have to be controlled in programming and erasing the device. One must know whether these pins are inputs, outputs, or bidirectional. If the pins require any special voltage levels, care must be taken in protecting the other parts on the same net. On the 7C374 and 7C375, only one pin is required to handle a voltage above normal TTL 'safe' levels. After the board is set up with the above conditions in mind, on-board programming of the 7C374 and 7C375 is quite simple.

The easiest way to program a part is to place it into a programming station. The next easiest way is to place the board into a programming mode and hook the programming station up to the board. If the environment of the board looks the same to the programmer as if the part were in its socket, a part can be easily programmed. This eliminates the many problems, including supplying a 'super' voltage, toggling signals HIGH and LOW, reading signals, writing signals, bringing in the programming file, and many others. These problems will be incurred if the desire is to be able to program the CPLD without outside help. Since an applications note on how to program with a programming station would prove to be duller than reading the phone book, this application note will show how to simply program these CPLDs by hooking your board to a programming station.

There are four types of signals which can feed the CY7C374/5. Three types are used in normal operation, INPUT, OUTPUT, and I/O. Programming mode supplies the fourth type, VPP or supervoltage.

All inputs and I/O signals to the device that are on the nets listed in *Table 1*, must be in High-Z while programming. This will eliminate contention from the programmer and the board's circuitry. There are several ways to accomplish this. Many parts have the ability to isolate themselves from other nets with built in three-state controls. Output enable or chip selects are found on most SRAMs, PLDs, FIFOs, and logic. If a device is driving a net that is used for programing and does not have the ability to be three-stated, a simple near-zero delay buffer can be added. An example of this device is the CYBUS3384. Once the output is enabled on a CYBUS3384, the delay time through the part is only 250 picoseconds. These parts are bidirectional without any direction-control hardware necessary (see *Figure 1*). The CYBUS3384 provides ten buffers in one space-saving QSOP package.

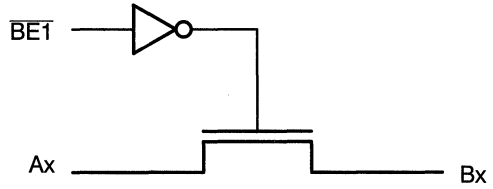


Figure 1. 'Zero' Delay Buffer

A 26-pin header may be installed on the card to allow on-board programming access. Nothing needs to be done with the OUTPUTs from the CY7C374/5 because no contention exists. The last signal type to contend with is the VPP signal. This signal has 12 volts applied to it during programming. There are two simple ways to isolate this high voltage from the system. The first is to reserve this pin for programming only. Because most designs only use one or two clocks, dedicating one of four for programming is usually not an issue. For those designs that require all clocks or all inputs, a jumper can simply be removed during programming to isolate the high voltage (*Figure 2*).

A signal needs to be generated to let the designed system know that it is in a programming state. One simple way to produce this signal is shown in *Figure 3*. By simply installing a jumper, the signal $\overline{\text{PROGRAM\_ENABLE}}$ is driven active. This signal should then be incorporated into the logic that controls the output enables and three-states of all signals that drive programming pins.

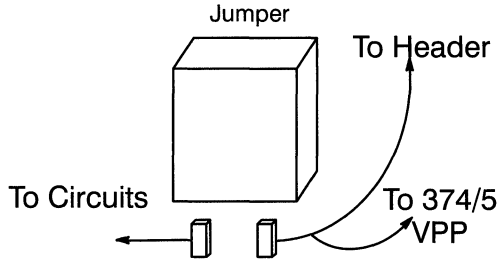


Figure 2. Isolation Jumper

By having a jumper installed for programming mode and a jumper removed for voltage isolation, a jumper will always be available on the board for use. Simply swap the jumper from one to the other. The list of the signals on the CY7C374 and CY7C375, the programming function, type, and the pin number for its location on the header are given in *Table 1*.

After using the information in *Table 1* to connect the appropriate signals to the twenty-six pin header, the rest is easy. A simple ribbon cable is used to connect the programming station (Quickpro II) to your board. Install the jumper to enable programming (*Figure 3*), isolate the super voltage by removing that jumper (*Figure 2*), if needed (remember, that pin can be dedicated to programming), and power up your board. The programming station takes care of the rest. Use it to read in the part's programming file and program the device. Now power down the board, and swap the jumper from $\overline{\text{PROGRAM\_EN}}$ generation to reconnecting the net connected to CLK1/I1. Power your system back on and you're ready to go.

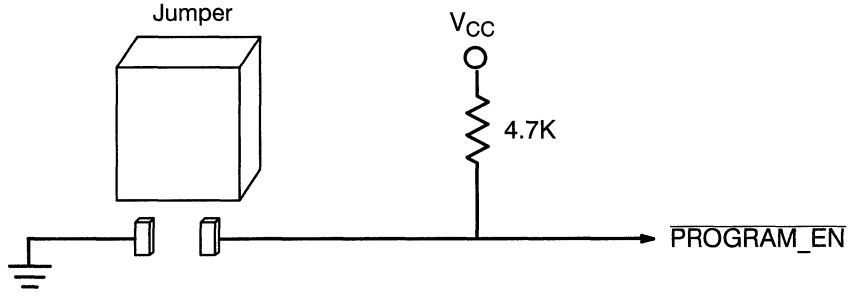


Figure 3. Install Jumper for Programming

Table 1. Pin Function and Position.

Function	Type	7C374 Signal	7C375 Signal	HDR Pin Number	QPII
rdenableb	input	CLK2/I3	CLK2/I3	9	A11
pgenableb	input	I/O15	I/O30	12	A15
data(7)	I/O	I/O38	I/O76	21	A26
data(6)	I/O	I/O36	I/O72	20	A25
data(5)	I/O	I/O34	I/O68	19	A24
data(4)	I/O	I/O32	I/O64	18	A23
data(3)	I/O	I/O30	I/O60	17	A22
data(2)	I/O	I/O28	I/O56	16	A21
data(1)	I/O	I/O26	I/O52	15	A20
data(0)	I/O	I/O24	I/O48	14	A19
mode(3)	input	I/O12	I/O24	25	A31
mode(2)	input	I/O14	I/O28	24	A30
mode(1)	input	I/O16	I/O32	23	A29
mode(0)	input	I/O18	I/O36	22	A28
verify	input	CLK0/I0	CLK0/I0	13	A17
VPP	VPP	CLK1/I1	CLK1/I1	8	A10
lsel	input	I/O8	I/O16	10	A13
leb	input	I/O10	I/O20	11	A14
it6/it9	input	I/O4	I/O8	7	A8
it5/it8	input	I/O2	I/O4	6	A7
it4/it7	input	I/O0	I/O0	5	A6
it3/pt3	input	I/O62	I/O124	4	A4
it2/pt2	input	I/O60	I/O120	3	A3
it1/pt1	input	I/O58	I/O116	2	A2
it0/pt0	input	I/O56	I/O112	1	A1
GROUND	GROUND			26	B32

Simulation of Cypress CPLDs with Mentor's QuickSim II

Simulation of Cypress CPLDs and smaller programmable logic devices in the Mentor Graphics environment is possible without the need for purchasing third party simulation models. Designs ranging the entire density span of Cypress programmable logic devices can quickly be placed into a form that can be imported into the mentor QuickSim II environment. It will be assumed that the person attempting to perform this task has some familiarity with the Cypress *Warp*™ software and Mentor QuickSim II.

After a design has been successfully compiled in the *Warp* environment, four easy steps are needed to get the design in the final form that QuickSim II can understand. The first step is to create a Viewlogic VHDL simulation model from the *Warp* design environment. Please refer to your *Warp* documentation for detailed instructions on how to do so. The second step is to do some slight editing to the VHDL files associated with the part family chosen for the design and the VHDL file exported from *Warp*. Thirdly, a 'wrapper' file must be constructed around the output file to convert Viewlogic I/O to Quicksim I/O. Finally, all the files edited and produced above are placed onto a disk for transfer into the Mentor environment.

The Four Steps for Simulating Cypress PLDs & CPLDs in the QuickSim II Environment:

1. Export Viewlogic VHDL file from *Warp*.
2. Small editing to the VHDL file.
3. Create the wrapper file.
4. Transfer files to Mentor environment and compile.

Let's take a detailed look at the four above steps.

Step 1: Export Viewlogic VHDL File from *Warp*

The first step in the process is to generate a Viewlogic VHDL simulation file from the *Warp* design tool. Please refer to the *Warp* documentation for instructions on how to generate this file. Once the *Warp* design tool is run and your VHDL has been created, you will find it in the /vhd subdirectory of your current project. The filename will be the same as your source code top-level filename. Once you have located this file, you are ready for step 2.

Step 2: Small Editing to the VHDL File

The file that was just written out is in a format that Viewsim understands. To put the file in a format for QuickSim, first we modify the beginning of the file as shown in *Figure 1*. Notice that one line is commented out and two are added. The second line added will vary depending upon which part was chosen when the design was compiled. The last changes that need to be made in this file are to add the lines as shown in *Figure 2*. The lines shown in *Figure 2* will change depending on your target device.

The proper 'use work.c{devicename}p.all;' clause and 'FOR ALL:...' statement for each target device is listed in Appendix A.

Each of the files listed in the 'FOR ALL:...' statements (c37xclk.vhd, c37xinp.vhd, ..., and

```
--  CYPRESS NOVA XVL Structural Architecture
--  JED2VHD Reverse Assembler - Ver 0.09 Oct 26, 1993
--    Viewlogic HDL File: FORDT.vhd
--      Date: Tue Oct 18 22:15:45 1994
--  Disassembly from Jedec file for: c371
--  Device Ordercode is: CY7C371-143JC
-- library primitive;  **** Commented out this line ****
use work.pack1076.all; **** Added these two lines ****
use work.c37xp.all;    **** work.c37xp.all is used for any Flash370 device ***
```

Figure 1. First Modifications to Example File

```
ARCHITECTURE DSMB of design_FORDT is

-- stuff that needs to be added for MENTOR system 1076

FOR ALL: c37xclk use entity work.c37xclk(sim); -- These statements will
FOR ALL: c37xinp use entity work.c37xinp(sim); -- change with different target
FOR ALL: c37xm use entity work.c37xm(sim);     -- devices and/or families.
FOR ALL: c37xmux use entity work.c37xmux(sim);
FOR ALL: c37xoreg use entity work.c37xoreg(sim);
FOR ALL: c37xprod use entity work.c37xprod(sim);
--
```

Figure 2. Additions to the Architecture

c37xprod.vhd) also have small changes that must be made (see *Figure 3*). For ease of use, the Cypress BBS contains all of these files pre-modified and they can be downloaded at your convenience. The files are in a self-extracting archive file called: VHDL_SIM.EXE.

After completing all of these modifications, step 2 is complete.

```
--------------------------------------------------------------------------
--     Entity / Architecture pairs
--     For c37xclk
--------------------------------------------------------------------------
--
--   Copyright Cypress Semiconductor Corporation, 1994
--     as an unpublished work.
--
--       $Id: c37xclk.vhd,v 1.8 1994/09/22 20:08:23 hemmert Exp $
--
use work.pack1076.all;      -- This one line must be added to the top of
                            -- every device library (FOR ALL:...) file
```

Figure 3. Modification to FOR ALL:... Files, If Premodified Files Are Not Used

Step 3: Create the Wrapper File

Creating the wrapper file is accomplished by simply performing multiple cut-and-pastes and search-and-replaces. The wrapper is used to translate vlbits (Viewlogic bits) to qsim_states (Mentor simulation states). To do this, a pair of functions is used. One function translates from qsim_state to vlbit, and the other translates vlbit to qsim_state. The first step is to copy the **entity** from the *Warp*-produced VHDL file into the file that contains our two functions. Now with your text editor, search for **vlbit** and replace it with **qsim_state** *(Figure 4)*. This completes the entity of the wrapper.

```
--  CYPRESS NOVA XVL Structural Architecture
--  JED2VHD Reverse Assembler - Ver 0.09 Oct 26, 1993
--    Viewlogic HDL File: FORDT.vhd
--      Date: Tue Dec 27 16:23:47 1994
--  Disassembly from Jedec file for: c22v10
--  Device Ordercode is: PAL22V10C-10JC

use work.pack1076.all;  --  This line is part of the standard template.
use work.c22v10p.all;   --  For a Flash370 device, use work.c37xp.all;

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY FORDT IS
     PORT(
  clock      : in  qsim_state ;
  right      : in  qsim_state ;
  left       : in  qsim_state ;
  flash      : in  qsim_state ;
  brake      : in  qsim_state ;
  node6      : in qsim_state ;
  node7      : in qsim_state ;
  node8      : in qsim_state ;
  node9      : in qsim_state ;
  node10     : in qsim_state ;
  node11     : in qsim_state ;
  node12     : in qsim_state ;
  node13     : in qsim_state ;
  r_outer    : inout qsim_state ;
  r_inner    : inout qsim_state ;
  l_middle   : inout qsim_state ;
  vl1i139_H2 : inout qsim_state ;
  vl1i137_H2 : inout qsim_state ;
  vl1i136_H2 : inout qsim_state ;
  vl1i138_H2 : inout qsim_state ;
  l_inner    : inout qsim_state ;
  l_outer    : inout qsim_state ;
  r_middle   : inout qsim_state ;
  node24     : in qsim_state
     );
END FORDT;
```

Figure 4. The Wrapper Entity

The next step is to create the architecture of the wrapper. To start this step, first type in the function template that converts vlbits to/from qsim_states, as mentioned above (*Figure 5*).

Next, the design that we are wrapping around is called in as a component. The port mapping for the component is created by simply copying the original entity used above. This time, no search-and-replace is needed (*Figure 6*).

The final step in creating the wrapper is instantiating the design as a component and hooking up the I/O to the wrapper through the functions listed in *Figure 5*. For the port map, start by copying the entity from the top of the file once again. For all signals of type in, use the qsim_state2vlbit function on the right side of the port map. For all inout, vlbit2qsim_state is used on the left and qsim_state2vlbit is used once again on the right (*Figure 7*).

This ends the creation of the wrapper. The wrapper is shown in its entirety in Appendix B. The more I/O pins a device has, the larger the wrapper file will be. However, because the creation is simply several copy-and-pastes and search-and-replaces, the size of the design will not seriously increase the amount of time needed to put the wrapper together.

Step 4: Transfer Files to Mentor Environment and Compile

We are now ready to transfer the design into the Mentor environment. In addition to the VHDL file we modified (that was generated by *Warp*), copy the files listed in Appendix B to your transfer media (tape, floppy, punch cards?). Place these files in a directory in the Mentor environment. Compile the files in the following order:

```
ARCHITECTURE structural OF fordt_wrapper IS

 --  Mapping functions for viewlogic states to/from  qsim_state

function qsim_state2vlbit (i : in qsim_state) return vlbit is
   begin
      case i is
         when '0' =>       return '0';
         when '1' =>       return '1';
         when 'X' =>       return 'X';
         when 'Z' =>       return 'Z';
      end case;
   end;

-------------------------------------------------------------------
function vlbit2qsim_state (i : in vlbit) return qsim_state is
   begin
      case i is
         when '0' =>       return '0';
         when '1' =>       return '1';
         when 'X' =>       return 'X';
         when 'Z' =>       return 'Z';
      end case;
   end;
-------------------------------------------------------------------
```

Figure 5. Functions for vlbit to/from qsim_state

```
component design_FORDT
  PORT(
    clock       : in  vlbit ;
    right       : in  vlbit ;
    left        : in  vlbit ;
    flash       : in  vlbit ;
    brake       : in  vlbit ;
    node6       : in vlbit ;
    node7       : in vlbit ;
    node8       : in vlbit ;
    node9       : in vlbit ;
    node10      : in vlbit ;
    node11      : in vlbit ;
    node12      : in vlbit ;
    node13      : in vlbit ;
    r_outer     : inout vlbit ;
    r_inner     : inout vlbit ;
    l_middle    : inout vlbit ;
    vl1i139_H2 : inout vlbit ;
    vl1i137_H2 : inout vlbit ;
    vl1i136_H2 : inout vlbit ;
    vl1i138_H2 : inout vlbit ;
    l_inner     : inout vlbit ;
    l_outer     : inout vlbit ;
    r_middle    : inout vlbit ;
    node24      : in vlbit
  );
end component;

FOR ALL: design_fordt USE ENTITY work.design_fordt;
```

Figure 6. Calling in the Original Design as a Component

1. pack1076.vhd
2. c{devicename}p.vhd
   Example: c37xp.vhd or c22v10p.vhd
3. The rest of the device library files listed for your target device in Appendix B.
4. The wrapper file.

After successful compilation, the design is ready to be connected to a symbol for board and system-level simulation.

```
BEGIN
-------------------------
-- instantiate the design
-------------------------
u1: design_fordt
  port map
    (
      clock    => qsim_state2vlbit(clock),
      right    => qsim_state2vlbit(right),
      left     => qsim_state2vlbit(left),
      flash    => qsim_state2vlbit(flash),
      brake    => qsim_state2vlbit(brake),
      node6    => qsim_state2vlbit(node6),
      node7    => qsim_state2vlbit(node7),
      node8    => qsim_state2vlbit(node8),
      node9    => qsim_state2vlbit(node9),
      node10   => qsim_state2vlbit(node10),
      node11   => qsim_state2vlbit(node11),
      node12   => qsim_state2vlbit(node12),
      node13   => qsim_state2vlbit(node13),
      vlbit2qsim_state(r_outer)    => qsim_state2vlbit(r_outer),
      vlbit2qsim_state(r_inner)    => qsim_state2vlbit(r_inner),
      vlbit2qsim_state(l_middle)   => qsim_state2vlbit(l_middle),
      vlbit2qsim_state(l_middle)   => qsim_state2vlbit(l_middle),
      vlbit2qsim_state(vlli137_H2) => qsim_state2vlbit(vlli137_H2),
      vlbit2qsim_state(vlli136_H2) => qsim_state2vlbit(vlli136_H2),
      vlbit2qsim_state(vlli138_H2) => qsim_state2vlbit(vlli138_H2),
      vlbit2qsim_state(l_inner)    => qsim_state2vlbit(l_inner),
      vlbit2qsim_state(l_outer)    => qsim_state2vlbit(l_outer),
      vlbit2qsim_state(l_middle)   => qsim_state2vlbit(l_middle),
      node24   => qsim_state2vlbit(node24)
    );
end structural;
```

Figure 7. Instantiating and Mapping the Design

Appendix A. List of Files Needed for Mentor QuickSim II by Part Type

Part Type	Files Needed	Line Added Before the Entity	Lines Added in the Architecture
16L8	C16L8P.VHD PACK1076.VHD	use work.c16l8p.all; use work.pack1076.all;	
16R4	C16R4P.VHD PACK1076.VHD	use work.c16r4p.all; use work.pack1076.all;	
16R6	C16R6P.VHD PACK1076.VHD	use work.c16r6p.all; use work.pack1076.all;	
16R8	C16R8P.VHD PACK1076.VHD	use work.c16r8p.all; use work.pack1076.all;	
16V8	C16V8M.VHD C16V8P.VHD PACK1076.VHD	use work.c16v8p.all; use work.pack1076.all;	FOR ALL: c16v8m use entity work.c16v8m(sim);
20G10	C20G10CM.VHD C20G10CP.VHD C20G10M.VHD C20G10P.VHD PACK1076.VHD	use work.c20g10p.all; use work.pack1076.all;	FOR ALL: c20g10cm use entity work.c20g10cm(sim); FOR ALL: c20g10cp use entity work.c20g10cp(sim); FOR ALL: c20g10m use entity work.c20g10m(sim);
20RA10	C20RA10M.VHD C20RA10P.VHD PACK1076.VHD	use work.c20ra10p.all; use work.pack1076.all;	FOR ALL: c20ra10m use entity work.c20ra10m(sim);
22V10	C22V10M.VHD C22V10P.VHD PACK1076.VHD	use work.c22v10p.all; use work.pack1076.all;	FOR ALL: c22v10m use entity work.c22v10m(sim);
22VP10	C22VP10M.VHD C22VP10P.VHD PACK1076.VHD	use work.c22vp10p.all; use work.pack1076.all;	FOR ALL: c22vp10m use entity work.c22vp10m(sim);
7C331	C331CKMX.VHD C331M.VHD C331P.VHD PACK1076.VHD	use work.c331p.all; use work.pack1076.all;	FOR ALL: c331ckmx use entity work.c331ckmk(sim); FOR ALL: c331m use entity work.c331m(sim);
7C335	C335CKMX.VHD C335H.VHD C335IREG.VHD C335M.VHD C335P.VHD PACK1076.VHD	use work.c335p.all; use wor.pack1076.all;	FOR ALL: c335ckmx use entity work.c335ckmx(sim); FOR ALL: c335h use entity work.c335h(sim); FOR ALL: c335ireg use entity work.c335ireg(sim); FOR ALL: c335m use entity work.c335m(sim);

Appendix A. List of Files Needed for Mentor QuickSim II by Part Type (continued)

Part Type	Files Needed	Line Added Before the Entity	Lines Added in the Architecture
7C34X	C34XCKMX.VHD C34XEXIN.VHD C34XEXP.VHD C34XH.VHD C34XIN.VHD C34XM.VHD C34XPIA.VHD C34XP.VHD PACK1076.VHD	use work.c34xp.all; use work.pack1076.all;	FOR ALL: c34xckmx use entity work.c34xckmx(sim); FOR ALL: c34xexin use entity work.c34xexin(sim); FOR ALL: c34xexp use entity work.c34xexp(sim); FOR ALL: c34xh use entity work.c34xh(sim); FOR ALL: c34xin use entity work.c34xin(sim); FOR ALL: c34xm use entity work.c34xm(sim); FOR ALL: c34xpia use entity work.c34xpia(sim);
7C37X	C37XCLK.VHD C37XINP.VHD C37XM.VHD C37XMUX.VHD C37XOREG.VHD C37XPROD.VHD C37XP.VHD PACK1076.VHD	use work.c37xp.all; use work.pack1076.all;	FOR ALL: c37xclk use entity work.c37xclk(sim); FOR ALL: c37xinp use entity work.c37xinp(sim); FOR ALL: c37xm use entity work.c37xm(sim); FOR ALL: c37xmux use entity work.c37xmux(sim); FOR ALL: c37xoreg use entity work.c37xoreg(sim); FOR ALL: c37xprod use entity work.c37xprod(sim);

Appendix B. The Wrapper

```
--  CYPRESS NOVA XVL Structural Architecture
--  JED2VHD Reverse Assembler - Ver 0.09 Oct 26, 1993
--    Viewlogic HDL File: FORDT.vhd
--      Date: Tue Dec 27 16:23:47 1994
--  Disassembly from Jedec file for: c22v10
--  Device Ordercode is: PAL22V10C-10JC

use work.pack1076.all;  --  This line is part of the standard template.
use work.c22v10p.all;   --  For a 37x part, use work.c37xp.all;

LIBRARY mgc_portable;            -- These lines are added for Mentor's
USE mgc_portable.qsim_logic.all; -- System 1076 VHDL compiler

ENTITY FORDT IS
     PORT(
  clock      : in  qsim_state ;
  right      : in  qsim_state ;
  left       : in  qsim_state ;
  flash      : in  qsim_state ;
  brake      : in  qsim_state ;
  node6      : in qsim_state ; --Notice that unused pins are assigned a
  node7      : in qsim_state ; --node number equivalent to their pin number.
  node8      : in qsim_state ;
  node9      : in qsim_state ;
  node10     : in qsim_state ;
  node11     : in qsim_state ;
  node12     : in qsim_state ;
  node13     : in qsim_state ;
  r_outer    : inout qsim_state ;
  r_inner    : inout qsim_state ;
  l_middle   : inout qsim_state ;
  vl1i139_H2 : inout qsim_state ;
  vl1i137_H2 : inout qsim_state ;
  vl1i136_H2 : inout qsim_state ;
  vl1i138_H2 : inout qsim_state ;
  l_inner    : inout qsim_state ;
  l_outer    : inout qsim_state ;
  r_middle   : inout qsim_state ;
  node24   : in qsim_state
     );
END FORDT;

ARCHITECTURE structural OF fordt_wrapper IS

 --  Mapping functions for viewlogic states to/from  qsim_state
```

Appendix B. The Wrapper (continued)

```
function qsim_state2vlbit (i : in qsim_state) return vlbit is
   begin
      case i is
         when '0' =>       return '0';
         when '1' =>       return '1';
         when 'X' =>       return 'X';
         when 'Z' =>       return 'Z';
      end case;
   end;

--------------------------------------------------------------------------
function vlbit2qsim_state (i : in vlbit) return qsim_state is
   begin
      case i is
         when '0' =>       return '0';
         when '1' =>       return '1';
         when 'X' =>       return 'X';
         when 'Z' =>       return 'Z';
      end case;
   end;
--------------------------------------------------------------------------
component design_FORDT
     PORT(
  clock       : in  vlbit ;
  right       : in  vlbit ;
  left        : in  vlbit ;
  flash       : in  vlbit ;
  brake       : in  vlbit ;
  node6       : in vlbit ;
  node7       : in vlbit ;
  node8       : in vlbit ;
  node9       : in vlbit ;
  node10      : in vlbit ;
  node11      : in vlbit ;
  node12      : in vlbit ;
  node13      : in vlbit ;
  r_outer     : inout vlbit ;
  r_inner     : inout vlbit ;
  l_middle    : inout vlbit ;
  vl1i139_H2  : inout vlbit ;
  vl1i137_H2  : inout vlbit ;
  vl1i136_H2  : inout vlbit ;
  vl1i138_H2  : inout vlbit ;
  l_inner     : inout vlbit ;
  l_outer     : inout vlbit ;
  r_middle    : inout vlbit ;
  node24    : in vlbit
     );
end component;
```

Appendix B. The Wrapper (continued)

```
FOR ALL: design_fordt USE ENTITY work.design_fordt;
BEGIN

-------------------------
-- instantiate the design
-------------------------

u1: design_fordt
   port map
      (
  clock    => qsim_state2vlbit(clock),
  right    => qsim_state2vlbit(right),
  left     => qsim_state2vlbit(left),
  flash    => qsim_state2vlbit(flash),
  brake    => qsim_state2vlbit(brake),
  node6    => qsim_state2vlbit(node6),
  node7    => qsim_state2vlbit(node7),
  node8    => qsim_state2vlbit(node8),
  node9    => qsim_state2vlbit(node9),
  node10   => qsim_state2vlbit(node10),
  node11   => qsim_state2vlbit(node11),
  node12   => qsim_state2vlbit(node12),
  node13   => qsim_state2vlbit(node13),
  vlbit2qsim_state(r_outer) => qsim_state2vlbit(r_outer),
  vlbit2qsim_state(r_inner) => qsim_state2vlbit(r_inner),
  vlbit2qsim_state(l_middle) => qsim_state2vlbit(l_middle),
  vlbit2qsim_state(l_middle) => qsim_state2vlbit(l_middle),
  vlbit2qsim_state(vlli137_H2) => qsim_state2vlbit(vlli137_H2),
  vlbit2qsim_state(vlli136_H2) => qsim_state2vlbit(vlli136_H2),
  vlbit2qsim_state(vlli138_H2) => qsim_state2vlbit(vlli138_H2),
  l_inner  vlbit2qsim_state(l_inner) => qsim_state2vlbit(l_inner),
  vlbit2qsim_state(l_outer) => qsim_state2vlbit(l_outer),
  vlbit2qsim_state(l_middle) => qsim_state2vlbit(l_middle),
  node24 => qsim_state2vlbit(node24)
         );
end structural;
```

Warp is a trademark of Cypress Semiconductor Corporation.

Architectures and Technologies for FPGAs

Introduction

The FPGA (Field Programmable Gate Array) is the newest concept in programmable logic. Previously the most complex programmable logic device was the Complex Programmable Logic Device, the CPLD. The CPLD concept is a simple extension of the basic PLD. Taking a small PLD device design and repeating it multiple times on the same die provides large resources in a single device. It is then necessary is to provide interconnect resources to allow each repeated cell to share resources and to communicate with one another and the I/O cells. The individual repeated cells are called macrocells which are, of course, relatively large and functionally complex.

Before the FPGA, the next level up from the PLD in solution alternatives was the sea-of-gates gate array. This is a fixed die, consisting of transistors, that is customized by the user by specifying the interconnect of the transistors. The user, in actuality, specifies the interconnection between a set of functional primitives such as NAND gates and flip-flops, which the gate array vendor has predefined and placed in a library. The gate array is not user programmable and must be customized by the vendor in the manufacturing process. Delivery of first articles is many weeks, and non-recurring costs are usually above ten thousand dollars.

Between the CPLD and the gate array is the FPGA which borrows from the solutions above and below. The FPGA logic cells are small and have less functionality than those of the CPLD. Thus they are a move toward the sea-of-gates concept in the Gate Array ASIC. Since the FPGA logic cells are smaller than those of the CPLD, there are many more of them in the same die. The FPGA logic cells are arranged in a rectangular array as in the gate array sea-of-gates concept. Between each logic cell is a routing channel so that multiple interconnect wires can run vertically and horizontally across the chip. Programmable connection points are provided where the logic cell I/O enters the routing channel and at the cross points where vertical routing channels meet horizontal routing channels. By appropriate programming of the cell I/O connections and the cross channel connections, signals can be routed throughout the chip.

Although the FPGA concept is relatively simple, realization of the FPGA is complex. There are many interrelated technology and architecture issues which must be addressed to produce a successful device. Success in this context means a device which can make maximum use of the available resources to accommodate large designs, achieve the highest possible performance that the semiconductor process technology has to offer, and give the designer flexibility (in, for example, pin assignments). This application note is intended to explain key factors in technology and architecture issues and how they relate. From this understanding, benefits to the designer will emerge. Different FPGA approaches have very different characteristics that can make the difference between a design achieving the required performance or being able to fit into a specific device. The material in this note is intended to help the design engineer make choices that will help achieve design goals.

Detailed Architecture

The global form of an FPGA is shown in *Figure 1*. The layout is a matrix of logic cells with a grid of routing channels running between the cells. I/O cells surround the array and allow access to the ex-

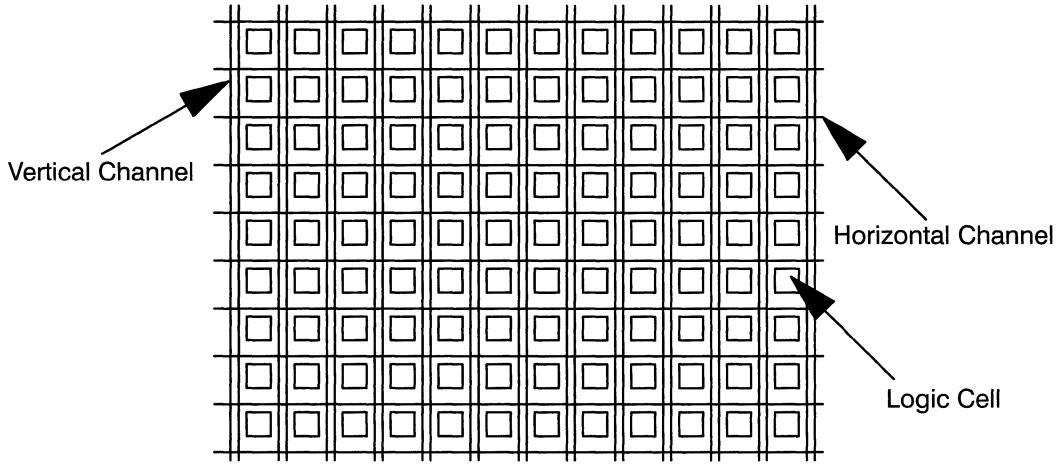


Figure 1. Global FPGA Architecture

ternal pins of the device. The programmable connections are located where the vertical and horizontal routing channels cross; where the I/O of the logic cells meet the routing channels; and where the I/O cells around the periphery meet the routing channels. In contrast to the CPLD, the FPGA usually has no programmability within the logic cells themselves (Some FPGA architectures do include programmable elements within the logic cell. Beyond this fundamental architecture, FPGAs can differ widely in the details. Key considerations are: (1) the number of wires in the routing channels, (2) the flexibility in the interconnect programmability where channels meet channels and where logic cells meet channels, and (3) the functionality contained within the logic cell. The details of the architecture choices in these key considerations are not obvious and are closely tied to the semiconductor process technology that is used to realize the device. The remainder of this section focuses on some of the more important aspects of these details: to show how they influence architecture choices and what impact the choices have on the performance, cost, and utility of the final product.

FPGA Logic Cells

There are three approaches to the form of a logic cell in FPGA implementations.

One approach is to make the logic cell as complex as in the CPLD. Such an implementation is termed a Coarse-Grain Logic Cell. An example is shown in *Figure 2*. This cell contains multiple flip-flops, several multiplexers, a combinatorial function block, and a variety of different inputs. Each of the flip-flops may be bypassed in order to implement combinatorial functions. From a first level analysis, it is clear that this logic cell can implement complex logic as well as register intensive functions. Three important characteristics are significant to note. First, the cell is complex and, as will be explained later, unless there is programmability in the cell, this can lead to inefficient use of the available logic. Second, there are a small number of cell outputs (two in this case). Third, many of the inputs are dedicated to flip-flop control and are not usable for other purposes if the flip-flop is not required. Timing analysis of this cell can be complicated. Since the cell possesses a large amount of functionality, it reduces the burden on the cell-to-cell interconnect.

The second approach in the cell architecture is to make the logic cell very simple. This approach is termed the Fine-Grain Logic Cell. An example is shown in *Figure 3*. This cell contains no flip-flops and very minimal logic. Only one output is available. The cell can realize simple AND-OR logic implementations and, because of its simplicity, it can

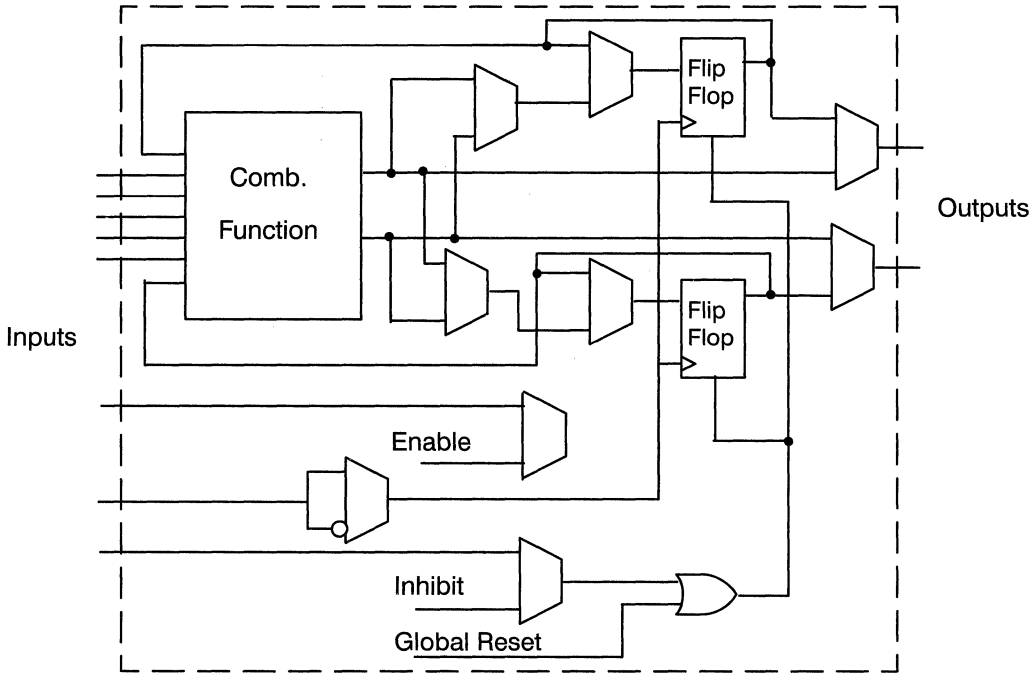


Figure 2. Coarse-Grain Logic Cell

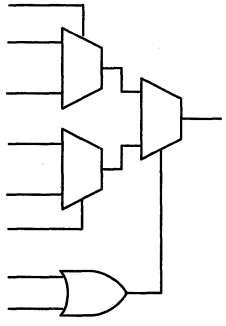


Figure 3. Fine-Grain Logic Cell

have very low propagation delays through the cell and there tends to be a high cell utilization. This cell will rely heavily on the cell-to-cell routing resources. The greater use of routing may add to the overall delay negating the low delay of the cell.

The third approach is a hybrid between the coarse-grain and fine-grain extremes but with some variations that enhance the other trade offs in the global architecture design. To understand the hybrid role, the relationship of the fine-grain and coarse-grain logic cell approaches to the global architecture design must be understood.

The global architecture relation to the logic cell type can be illustrated by starting from the basic FPGA concept and identifying and working through the implementation trade offs. Consider a sea of logic cells surrounded by the routing channels. Assume that the routing channels are very large and the interconnect completely flexible. Then, as in the custom sea-of-gates ASIC, the logic cells can be small, consisting of elementary logic primitives. The first trade off is that the FPGA is not a gate array where gates are sacrificed for routing. In the FPGA, the routing resources are limited and unused gates offer no increase in interconnect capability. This suggests

that the logic cell possess more than minimal functionality and should include at least one flip-flop and wide AND-OR combinations to realize complex logic. It also suggests that the logic cell have multiple outputs so that if some logic functions are very simple, more than one of these functions could be implemented in the same cell. This allows maximum utilization of the logic cell resources. Continuing this pattern would further increase functionality in the logic cell. Bear in mind, however, that the logic cell lacks programmability in the examples shown here and that directing signals within the logic cell is done with multiplexing and judiciously selecting input combinations. Further expansion of this would waste resources in the signal directing logic and increase logic cell propagation delay. This is detrimental to the objective of implementing the desired logic function. Added circuitry and controls in the logic cell to maximize its flexibility do not contribute to the realization of the desired function. That flexibility is the task of the interconnect. There is, therefore, an optimum logic cell complexity which lies somewhere between the coarse- and fine-grained extremes. With finite routing resources available, but without fuse related constraints, the optimal cell would look something like the cell in *Figure 4*. This cell design is simple and symmetric yielding small propagation delays regardless of the signal path through the cell. Since the interconnect is not a factor, a large number of inputs is provided so that logic functions of many variables can be implemented. The cell also provides for the realization of any logic function up to a given number of variables. Multiple outputs are also provided. The flip-flop may be used or the "D" input of the flip-flop is available as an output for combinatorial only functions. The large number of outputs allows the cell to be split, implementing two or more simple logic functions in the same cell or sharing logic across function in the same cell.

Programmable Connections

Now examine the interconnect surrounding the sea of logic cells. There are two key issues in the implementation of the programmable connections. First is size. The programmable interconnect circuit may

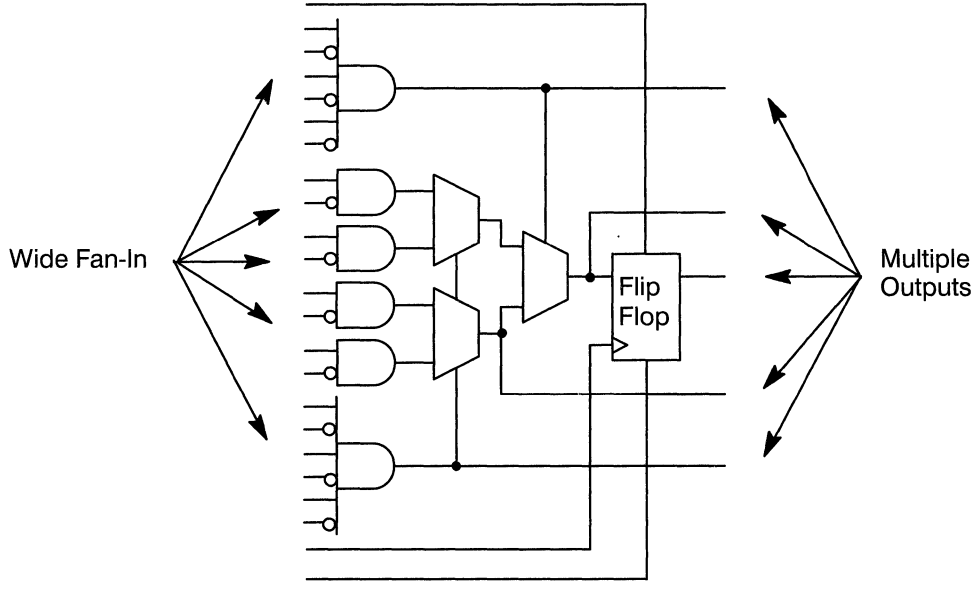


Figure 4. An Optimal Logic Cell

use an area which limits the number of interconnects which can be put into a given area. Second is the electrical characteristics of the interconnect. The interconnect may not look like a wire to the signal that it is carrying. Resistance and capacitance of the interconnect can affect the propagation delay of the signal.

Infinite routability is not realistic. The routing channels can contain only a finite number of wires and the interconnect possibilities where wires cross or meet is not endless. There are three classes of ways to connect two wires: RAM-based connections, large fuse technology, and via fuse technology. Consider the rectangular area where horizontal and vertical wiring channels cross. At this intersection there is potentially a connection possibility at each wire crossing (intersection). An ideal case of the user programmable interconnect possibilities are shown in *Figure 5*. Each circle at the intersection point represents a potential user-programmable connection. This scheme offers a great deal of interconnect capability. There is a lot of redundancy in the interconnect possibilities. Once a wire is connected to a signal, it is dedicated to that signal throughout the extent of that wire. Wires may be segmented so that they can support local interconnect. The implementation of the connection mechanism (fuse, RAM cell, etc.) may be larger than the dimension of the wire size and inter-wire spacing as shown in *Figure 6*. Because of the size of the connection cell for connection "O," programmable connection at points "X" are not possible. Further programmable connection cells cannot fit into the area unless the interconnect wires are spread apart. This latter approach is not an effective alternative since it upsets the regularity and fit of the wiring channels and the logic cells. The only alternative is to limit the number of connection possibilities.

An example of this situation is RAM-based programmable interconnect. RAM-based connectivity uses a memory cell to control the connection of one wire to another. The memory cell is far larger than the wire intersect area. What is done is to limit the connection possibilities to a small fraction of the possible signal paths and limit the number of wires in the routing channel. The result is that the routability of the FPGA becomes what is known as "interconnect constrained." That is, the number of wire connection possibilities is so small that the interconnect limits the realization of functions with a given logic cell architecture. To compensate for this, coarse-grained logic cells are usually chosen and programming internal to the logic cell may be added. The ability to perform more function in a logic cell tends to make up for the inability to implement the equivalent function in a set of interconnected logic cells. It is difficult to quantify the results of such choices. Can a large complex logic cell adequately compensate for interconnect constrained situations? There is no clear answer and the results are dependent on what is to be implemented in the FPGA. However, it can be determined by realizing various types of functions in various architecture forms that the interconnect-constrained architec-

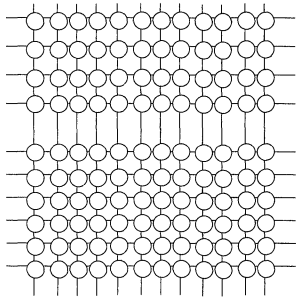


Figure 5. Connection Points at Routing Channel Intersection

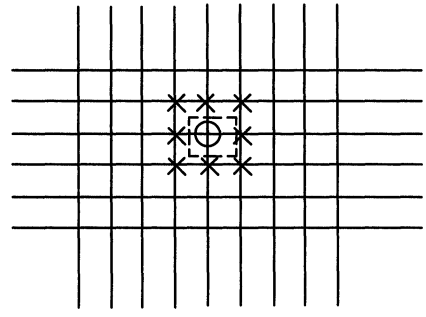


Figure 6. Connection Limitations due to Programmable Connect Cell Size

tures tend to have routability limitations which are exacerbated when any constraints are placed on the device pin/signal association.

Architectures implemented with large-fuse technology tend to have the same characteristics as RAM-based connectivity except they are not as severe. Both approaches limit the number of outputs in the logic cell to only one or two. This is because increased outputs add to the number of potential connections, which places a further burden upon an already stressed interconnect mechanism. Because the number of outputs is small, simple functions will tend to waste logic resources in the interconnect-constrained, complex logic cell architectures. Therefore, these architectures may tend to be more efficient in implementing complex state machines than the fine-grained logic cell architectures with the same interconnect capability.

Of the programmable interconnect technologies available for FPGAs, (RAM-based, large-fuse, and via-fuse) the optimum interconnect is achieved by via-fuse technology. This technology realizes an antifuse in the same physical area as that used by a normal semiconductor process via which connects two layers of metal. The via fuse will be described in detail in a later section. This technology approaches the interconnect characteristics found in sea-of-gates gate arrays and almost completely eliminates programmable interconnect as a factor in FPGA wiring channel and logic cell architectures.

The electrical characteristics of the interconnect technology play a major role in the performance of the FPGA. A first level summary of the technology impact on the interconnect (not including wire delay) is given in *Table 1*. When the connection is made, it exhibits some ON resistance in series with the logic signal path. When a connection is OFF it presents a shunt capacitance from the logic signal wire to ground. A single ON fuse followed by a single OFF fuse represents an RC combination in the signal path. The product of the ON resistance and the OFF capacitance of this combination is given in the Time Constant column of the table.

Timing Model

The timing model is a representation of signal delays in an actual FPGA that allow the designer to determine the performance of a design when it is realized in a particular device. The nature of the timing model is of concern to the designer since it affects the level of difficulty in determining performance. All FPGAs, by virtue of their architectures, inherently have variable timing models. The pin-to-pin propagation delay depends upon the number of logic cells cascaded together to achieve a given logic function. There are two types of variable timing models: simple and fine structured. In the simple variable timing model, the pin-to-pin propagation delay is chiefly dependent upon the number of cascaded logic cells, signal fan out, and the wire delay that would normally be encountered in a sea-of-gates gate array. The number of interconnect points in the signal path and the logic function implemented in the logic cell tend to have secondary and lesser effects on the timing. Architectures suitable for the simple timing model will have actual device delay characteristics which are independent of where the logic cell is placed in the array. In the fine structured variable timing model, pin-to-pin propagation delay is strongly dependent upon not only the simple model factors but also the number of programmable interconnects in the signal path and the function implemented in the logic cell. In these cases, the logic cell itself has a variable timing model due to its complex structure and antisymmetry. Programmable interconnect points with unfavorable electrical characteristics raise the effect of the number of interconnect points in a signal path to being a first order effect.

Table 1. Electrical Characteristics of Fuse Technologies

Technology	ON Resistance	OFF Cap.	T	Tgate
Via Fuse	50 ohms	1 fF	0.05 ps	50 ps
Large Fuse	400 ohms	5 fF	2 ps	400 ps
RAM Cell	800 ohms	10 fF	8 ps	800 ps

The actual performance of devices does not differ by these orders of magnitude. This is because the OFF capacitance of a large number of no-connect fuses is small compared to the metal and gate input capacitances. Therefore the fuse series resistance is the dominant component in limiting performance due to programmable interconnect. To put this factor into perspective, *Table 1* includes a column which is the time constant for one series fuse connected to a gate with a capacitance of 1 pF. Note that with as few as five programmable interconnects in the path from the signal source to one gate, the time constant, for some technologies, can be as large as the gate delay itself.

Interconnect and Logic Cell Trade Off Summary: Advantages and Weaknesses

The technology has a profound effect on the total FPGA architecture. SRAM and large fuse based technologies cause interconnect-constrained architectures which force non-optimal logic cell architectures. In general, interconnect-constrained architected logic cells tend to be large and complex to make up for the interconnect limitations. Moreover, these complex logic cells tend to be wasteful of resources in certain applications. Such architectures are characterized by routing and capacity limitations and an inability to fit a design when there are pinout constraints (user fixes signals to particular pins). Router and fitter software may take many iterations to fit a design.

It is clear that the small-size fuse technology, combined with well chosen routing channel wire complement and an optimum complexity logic cell, will yield a high performance, small die size device. *Figure 7* shows system performance of a fixed benchmark fitted into devices of the three technologies described above. The system performance is plotted versus the semiconductor process technology line width in order to perform an apples to apples

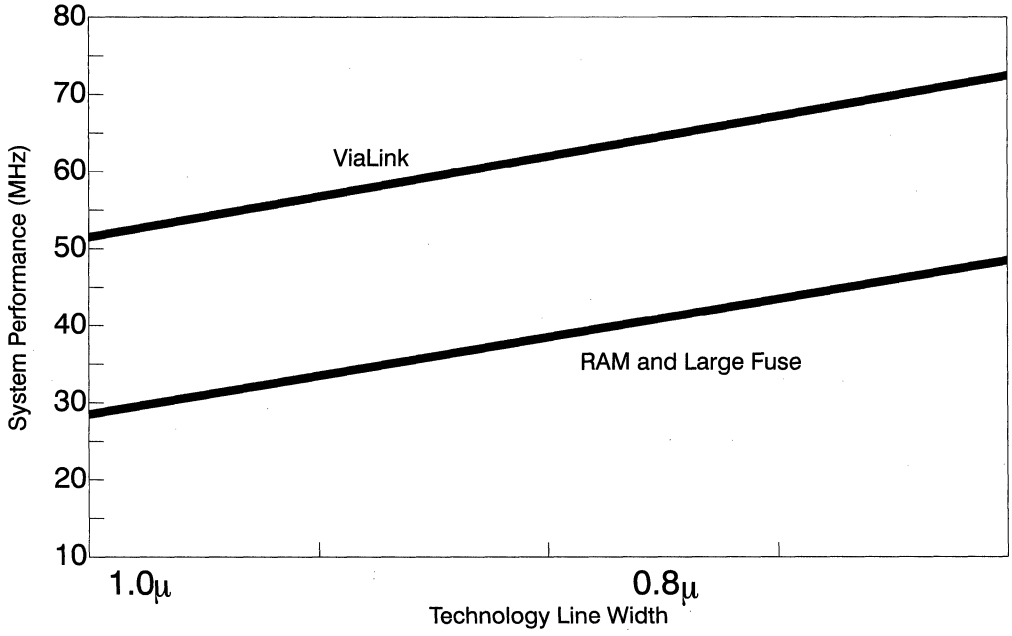


Figure 7. Performance Relative to Connect Technology

comparison of the key factors in the FPGA implementation. As expected, the architectures optimized for ViaLink™ exhibit a significant performance advantage over the large-fuse and RAM-based interconnect approaches.

Comparison to CPLDs

The architecture of the FPGA manifests itself in the device characteristics in much the same way that the sea-of-gates gate array architecture does. The two major influences on the device characteristics are the small logic cell size and the channel routing. First, the cell sizes are small and considerably less complex than those of the CPLD. Therefore, the propagation delay through the FPGA logic cell is much smaller than that through the CPLD macrocell. Functions such as multiplexing, which need only one cell per signal path, will typically achieve much higher performance in the FPGA. In contrast, functions which require cascading of many logic cells to implement may be at a performance disadvantage in the FPGA. Complex state machines with a lot of decoding are in this category. This does not mean that use of the FPGA is to be avoided. In the example to follow, a complex state machine is implemented successfully. Secondly, an abundance of routing resources can permit complicated interconnects as well as convenient handling of buses.

Cypress pASIC380™ Family FPGA Architectures

The previous architecture discussions have pointed out the strong relationship between the technology, the architecture of the FPGA, and the device characteristics. The 380 family possesses a unique technology which impacts all of the remaining architecture trade offs positively. The discussion of the 380 family begins, therefore, with a presentation of the interconnect technology.

pASIC380 Family Fuse Technology

In usual integrated circuits two crossing metal lines that are on different layers may be connected by a via. A via is a small hole in the insulating glass that lies between the two layers of metal. This small hole, which is about the size of the metal lines themselves, is filled with metal from above making the connection to the underlying metal line. The programmable via is a modified via used in standard CMOS semiconductor processing. The modification consists of depositing a thin layer of amorphous silicon in the via hole so that the silicon separates the two layers of metal. As manufactured, this special via has a resistance in excess of 1 gigaohm and an insignificantly small capacitance (about 1 fF). Its size is no larger than the standard via normally used to connect two layers of metal. A cross section of the programmable via is shown in *Figure 8*. A programming pulse applied across the programmable via causes a change in the characteristics of the silicon layer forming a bidirectional conductive link between the top and bottom metal. This programmed link has a series resistance of about 52 ohms and in practice is no more than 65 ohms. The parasitic capacitance is no larger than a normal metal to metal via. The technology is appropriately termed "ViaLink."

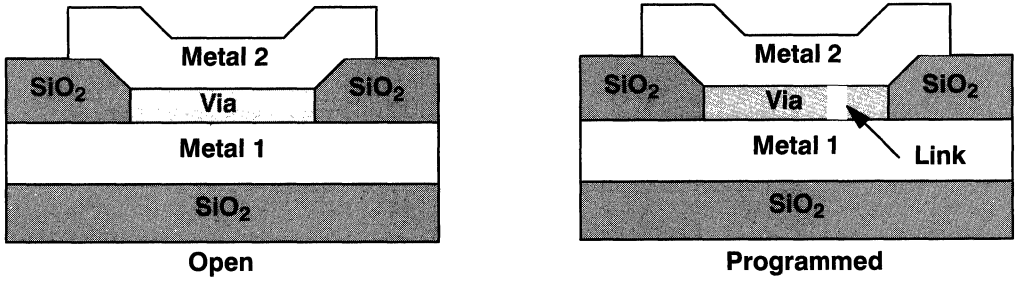


Figure 8. The ViaLink

Routing

ViaLink technology has significant impact on FPGA architecture. Since the programmable site is no larger than the associated metal interconnect wires, there is no real restriction on the number of interconnect points (fuses) and no fuse related restrictions on the number of wires in the interconnect channels. The 380 family routing scheme is architected with this added freedom.

Four types of signal wires are employed in the routing channels:

- segmented wires
- quad segmented wires
- express wires
- clock wires

Segmented wires are wires that extend only from one routing channel to the next, both vertically and horizontally. At the channel junction, a horizontal segmented wire may be programmed to interconnect to a vertical segmented wire at points called cross links. In *Figure 9*, programmable cross links are denoted by the open circle at intersections of vertical and horizontal wires. Also at the channel juncture, the segmented wire may be continued in the original horizontal or vertical direction by connection to another segmented wire running in the same channel. This connection is provided by a pass link. These links are denoted by an "x" in the figure. Segmented wires are most applicable for local wiring around or between adjacent logic cells.

Quad segmented wires are similar to the segmented wires described above except that the wire extends across four logic cells before it is segmented. Like segmented wires, the quad segmented wires may be continued to the next quad segmented wire by a pass link. The quad segmented wires are applicable to signal distribution over a larger but still local group of logic cells.

Express wires are similar to segmented wires except they do not include pass links. An express wire will therefore run the entire length of the device. These wires are most suitable for global signals within the device. Routing software with specific knowledge of the device architecture will automatically route signals over the appropriate wire type.

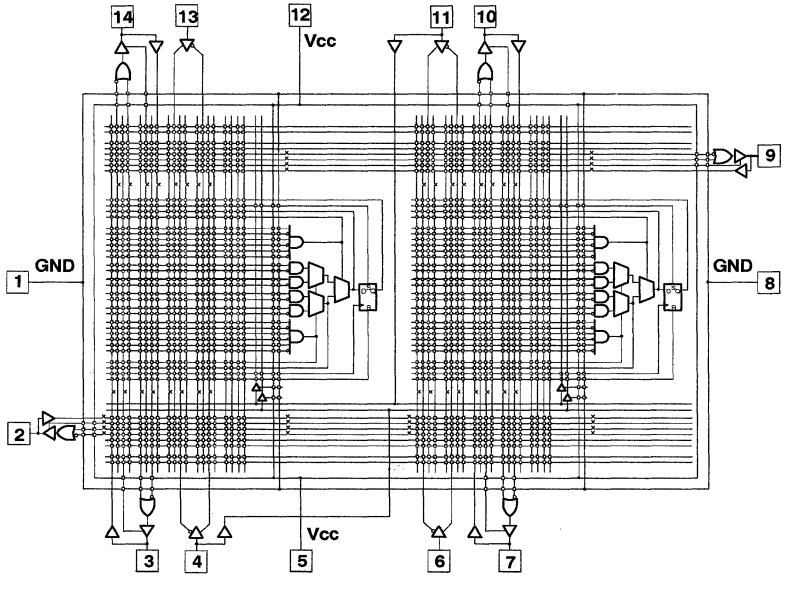


Figure 9. Simplified pASIC380 Family Model

Clock wires are special signal lines that include an array of buffers for minimal skew. Clock wires are similar to express wires except that the cross links are limited. This is to insure that the clock wires are lightly loaded by programmable interconnects and can be used maximally in routing high-speed clocks or reset signals globally throughout the device with minimal skew. The source of the signal on the clock wires is specific device pins with the designation "I/CLK." After passing through the special input buffers, the signal is routed horizontally across the center of the die, as shown in *Figure 10*. There are four high drive buffers. One pair drive clock 1 and clock 2 to the upper half of the column of logic cells, and the other pair drive the two clocks to the lower half column of logic cells. There is a cluster of these buffers for each column of logic cells in the array. The buffers can be enabled to drive the clock lines or disabled if a clock is not required in a given column.

Vertical channels include all three wire types plus V_{CC} and ground wires. The V_{CC} and ground connections allow unused inputs of any logic cell to be tied to an appropriate logic level. The vertical channels run to the left of each logic cell column and extend the full height of the device. The I/O wires, which run from each of the logic cells to the right of the vertical channel, intersect the wires of the vertical channel with cross links at all segmented wires and at judicious points for express wires. At the extreme ends of the vertical channels are I/O cells that connect to the device pins. The number of wires in the vertical channel is chosen to be commensurate with the number of inputs and outputs of a logic cell, the added wires for V_{CC}, ground, and the I/O cells at the device periphery. There are 24 of these wires.

Horizontal channels provide connection by way of cross links from vertical channel to vertical channel

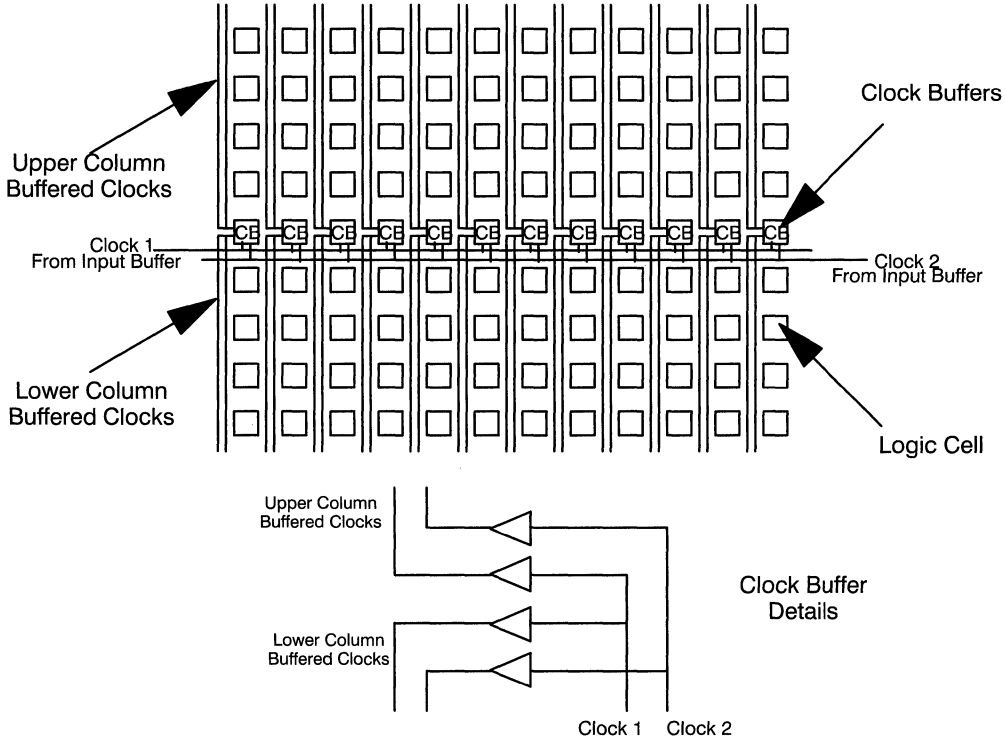


Figure 10. pASIC380 Family Clock Distribution

and from the vertical channels to I/O cells on the left and right periphery of the device. All wire types are included in the horizontal channels (which contain 12 wires each) except for the clock wires. (These are the dedicated wires that carry the clocks to the buffers.)

I/O Cells

There are three types of interface buffers that connect the internal array to the device pins. The dedicated input buffer provides high drive internally and generates both true and complementary versions of the input signal. This high drive capability allows signals coming from these input only buffers to fan out to a larger number of cells than the normal I/O cell. The clock input buffer is similar to the dedicated input buffer except that it provides a third output that is routed to the internal clock distribution buffers described previously. The I/O cell provides a bidirectional connection to the devices pins. The cell can be used as input only, output only, or a bidirectional pin connection. Internally the cell has an output enable, an input data connection, and two output data connections which are ORed together to produce the output. This cell is shown schematically in *Figure 11*. The output driver provides 8 mA drive level (I_{OH} and I_{OL}).

Logic Cells in the 380 Family

Since the routing resources of the 380 family are abundant and without expectation of being interconnect constrained, there is freedom in the logic cell architecture to choose the optimum complexity. The 380 family logic cell is shown in *Figure 12*. This cell has been optimized to maintain the speed advantage of the ViaLink technology while insuring maximum logic flexibility.

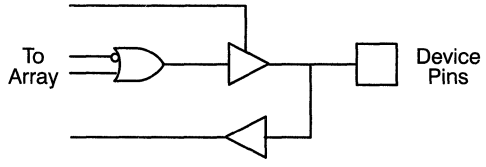


Figure 11. Bidirectional I/O Buffer

The logic cell consists of two 6-input AND gates, four 2-input AND gates, three 2-to-1 multiplexers and a D flip-flop. This cell represents approximately 30 gate equivalents of logic capability. The cell has 23 logic and control inputs and 5 outputs. The arrangement of the gates permits 14-bit-wide gating functions and can realize all possible Boolean transfer functions of up to three variables. The D flip-flop possesses asynchronous set and reset inputs to independently control the output state. The multiplexer and logic feeding the D input allow the flip-flop to be configured as D, T, JK, or SR.

The outputs of the logic cell include the Q output of the flip-flop (QZ) plus four other outputs tapped at selected points within the logic cell. The OZ output is the same as the D input to the flip-flop. The OZ output facilitates combinatorial functions. The three other combinatorial outputs tap the logic cell at selected places. If simple logic functions are to be implemented, the multiple outputs permit more than one of these functions to be realized in a single logic cell. Maximum use of the available logic can be made. Note the ability to provide this multifunction utilization without any significant impact on routing. The additional utilization factor is ob-

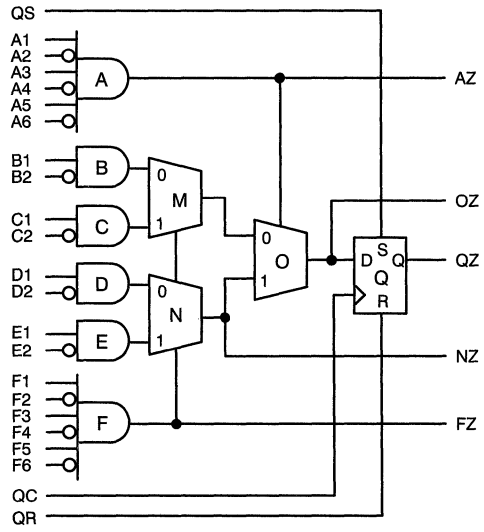


Figure 12. pASIC380 Internal Logic Cell

tained for free. When implementing multiple functions, the flip-flop may still be employed in many cases.

The logic cell is not so complex as to adversely impact propagation delay. The internal multiplexers are positioned to participate in implementing logic functions. Since the multiplexers are all in the path to the D input of the flip-flop, they contribute significantly to combinatorial logic function realization and are not expended on signal steering. The logic cell is also noticeably symmetric and regular. Combinatorial delays are thus also symmetric. That is, input to output delays tend to be roughly the same, although the AZ and FZ output will be faster than the others. Whereas some architectures bypass large sections of cell logic by the multiplexing, thereby making the cell delay dynamically changeable, the 380 logic cell delay is not subject to this condition.

Performance and Timing Model

An inherent characteristic of any FPGA is that the timing model is a variable model: logic implementation is accomplished by cascading a number of logic cells that is dependent upon the function to be implemented. For the non-ideal FPGA model, the device input to output propagation delay is a function of the number of logic cells in the signal path; the dynamics of the signal path in the logic cells; the number of programmable interconnects through which the signal traverses; the normal integrated circuit routing delay; and the I/O cell delay. This relationship can cause the variable timing model to be quite complex and depend upon the routing, placement, and cell dynamics. Since the 380 family has a fixed timing model for each logic cell, the cell configuration dynamics are not a factor in the 380 family timing model. Only the cell delays need to be summed for their contribution to the overall delay.

The 380 family programmable interconnect affects the propagation delay in much the same way as normal integrated circuit interconnects. That is, the fuses contribute to the delay as if they were slightly longer wires. This characteristic greatly reduces the complexity of the timing model and the variability in the timing results when a design is fitted to a device.

Conclusions and Summary

This application note is a first introduction to FPGAs. Specifics of the Cypress 380 family of devices were presented. It was shown that the fuse or connection technology has a very strong influence on the architecture of the FPGA logic cells and the interconnect scheme. Specifically the physical size of the interconnect (fuse or RAM cell) and its ON resistance are major influences on the FPGA logic cell complexity and interconnect architecture. The low ON resistance, physically small fuse permits

- an interconnect scheme virtually unconstrained by the number and location of fuses
- an optimized logic cell architecture
- device performance with minimal limitations from the fuse electrical characteristics

When an FPGA is architected with the freedoms listed above, the results are significant benefits to the user:

- FPGAs where designs are easy to fit, i.e., large capacity (non interconnect constrained)
- Flexibility in pin assignments
  (user can define/keep fixed after modifications)
- High performance/low propagation delays
- Easy to use timing models

When these are primary concerns, the small fuse technology offer the greatest opportunity for extracting these benefits.

Designing with FPGAs
An Introduction to Cypress's pASIC380 Family of FPGAs and the *Warp3* ™ Design Tool

Introduction

Field Programmable Gate Arrays (FPGA) borrow the sea of gates concept from the gate array semicustom integrated circuit and add field programmability. The similarity of the FPGA to the semicustom approaches opens many possibilities for the design engineer. With a large number of gates available, complex designs can be implemented into a single device. In the semicustom approach it may be many weeks from sign off to prototypes. Moreover, simulation is usually exhaustive (due to the cost of a design change), taking many weeks of the design cycle. With field programmability, a design may be realized in a device in a week or two of design time, in contrast to the many months with a semicustom approach. The FPGA brings gate-array-like possibilities to many design projects. The lack of a non-recurring engineering charge for FPGAs makes this technology financially available to a large number of developments.

This application note is intended to be an introduction to using FPGAs by taking the reader through a complete design. The first part of this note presents the design tools for FPGA designs. Here a design flow is followed from design entry in its multiple forms. The design flow is top down. That is, the process starts from a description of high-level abstract entry of the design and progresses to adding more hardware-specific details as required for realization in a device. To illustrate the back-end part of the design process, a DRAM memory controller is presented. In this part, details of speed optimization, simulation, and device specifics required in the design description are discussed.

Design Development and Cypress's *Warp3* ™ Design Tool

FPGA devices are resource-rich entities capable of implementing designs that use from 1K to 20K or more gates. These designs will not be simple, consequently the development of the design, its debugging, and final performance analysis will be a complicated task. Fortunately, gate array design and analysis tools can be used to make these tasks comparatively easy. The design proceeds beginning with the entry of the design description. This may be in schematic form or in a high level language form, such as VHDL. After entry, the VHDL code (or VHDL equivalent of the schematic) can be simulated directly to verify the functionality of the design description. The design is then synthesized and committed to a particular device. At this point the performance of the implementation can be analyzed and optimized if necessary to achieve a design target. Lastly, the actual device is programmed.

Warp3 is a modern, self-contained CPLD and FPGA design tool that supports all phases of the design process. At the front end, it includes VHDL and schematic entry design capture tools for efficient and convenient user entry. The synthesis tools include native compilation of VHDL (for accurate VHDL interpretation), and Cypress device-specific compilers for maximum utilization of device architectural features. The schematic capture, simulation, and the framework are ViewLogic™ tools

adapted specifically for PLD, CPLD, and FPGA designs.

VHDL Design

VHDL is a rich and powerful language for the description of logic circuits. The language offers the capability to use different styles of design entry. There are three styles of logic description that can be used in any combination: behavioral, dataflow, and structural. Behavioral descriptions are C- or Pascal-like constructs that specify the action of the logic in high level abstract terms. Language constructs such as the IF/ELSE statement or the CASE statement specify the behavior. Dataflow descriptions include Boolean equations that can be used to describe logic circuits. Tabular descriptions are also possible. These can be considered a subset of the behavioral descriptions but where the action of the logic is specified by a truth table. The structural description is much less abstract and can be considered a verbal description of a schematic. In one version of this form of VHDL, gates, flip-flops, and other primitives are instantiated, and their interconnection described through signals that tie the output of one primitive to the input of others. Conversely, several entities can be instantiated in a structural description of their interconnect, but the description of any of the individual entities can be behavioral.

VHDL is a hierarchical language. Just as most programming languages support subroutines, functions, and procedures, VHDL supports components, packages, and the ability to combine a set of entities into a higher-level entity. A complex VHDL design can be built by successively combining building blocks in related layers. This allows two, very powerful design approaches. First, a complex design can be done from a top-down approach. In this method, the whole of the design can be described in very abstract, high level terms. Then the design is decomposed—breaking the design apart into specific functional blocks that are described in more specific terms. This moves the design from concept to implementation, from abstract description to near hardware level realization and optimization. At the top level, the designer need not be concerned with the exact details of the design. The concern at this top level is the conformance of the design to the given functional specification. Once this is achieved, the design can be decomposed for realization purposes while being assured that the overall functional requirements are still met. Even at the top level, the design can be built up of manageable entities which can be debugged separately. Second, the design can be done from a bottom up approach. After a block diagram is sketched out, the individual blocks can be designed according to their function and the interfaces to the other blocks. The individual block designs can be done at the most detailed level. This can consist of schematics using components instantiated from a library or the design can be a structural VHDL description. With each block fully designed, they are then connected together to build the complete design.

As an example of VHDL, consider a 12-bit wide 4-to-1 multiplexer. A version of this multiplexer is used in the design example section of this application note. The first input is a 12-bit bus that is the column address for the DRAM. The second input is a select signal which controls whether the row (row_ad) or column (col_ad) is selected. The third input is a 12-bit bus that is the row address for the DRAM. The forth input is a 12-bit bus that is the refresh address for the DRAM, and the last input is the state of the controller finite state machine. The output of the multiplexer is a 12-bit address to be sent to the DRAM memory devices. When the finite state machine is in states *refad*, *wr1*, or *wr2*, the refresh address is placed on the multiplexer output. This selection is independent of the state of *col_sel*. When the finite state machine is not in states *refad*, *wr1*, or *wr2*, COL_SEL controls the multiplexing. When COL_SEL is 0, row_ad is placed on the output of the multiplexer, and when COL_SEL is 1, col_ad is placed on the output of the multiplexer. The VHDL code to implement this function is given in *Figure 1*. The code is compact and simple. One of the twelve synthesized logic equations is given in *Figure 2*. These logic equations are available in the report file generated during compilation. The equivalence of the behavioral code and the logic equation should be self evident: in the logic equation, the state machine state vector bits (for the states in the if statement) are ANDed with the re-

```
mux: process(col_ad,col_sel,row_ad,re_ad,state)
  begin
    if(state = refad or state or state = wr1 or state = wr2) then
      rc_ad <= re_ad;
    elsif(col_sel = '1') then
      rc_ad <= col_ad;
    else
      rc_ad <= row_ad;
    end if;
  end process;
```

Figure 1. Behavioral Description of Multiplexer

```
rc_ad_11_ = /controller_state_12_.Q * /controller_state_11_.Q *
            /controller_state_10_.Q * /controller_state_11_.Q *
            stored_ad_11_DFF.Q * col_sel.Q
          +
            /controller_state_12_.Q * /controller_state_11_.Q *
            /controller_state_10_.Q * /controller_state_11_.Q *
            stored_ad_23_DFF.Q * /col_sel.Q
          +
            controller_state_12_.Q * ref_ad_11_DFF.Q *
          +
            controller_state_11_.Q * ref_ad_11_DFF.Q *
          +
            controller_state_10_.Q * ref_ad_11_DFF.Q *
          +
            controller_state_9_.Q * ref_ad_11_DFF.Q *
```

Figure 2. Logic Equations for the Multiplexer

fresh address bit (REF_AD(11)) and the compliment of these bits are a factor in the remaining product terms. One of the other two product terms ANDs COL_SEL with col_ad(11); the other product term ANDs COL_SEL with row_ad(11). The logic equations refer to stored_ad_11 instead of col_ad(11) and stored_ad_23 instead of row_ad(11). This is because col_ad and row_ad are aliases for these signals. Refer to the appendix for the alias definition.

Schematic Entry

In some cases VHDL may not be the preferred method of capturing the design. A discrete implementation of the design may already exist with the objective of reducing size and improving performance by putting the circuit into an FPGA. Many designers may feel more comfortable with schematic design capture than with a high-level language description.

Mixed Mode

Some functions are difficult to describe directly in schematic form. A state machine, for example, is far easier to describe in terms of a transition table (possible in VHDL) or VHDL conditional constructs (for example, a CASE statement). Not only is the description easier but design changes and debug-

ging are also far easier in nonschematic form. It is therefore important for a tool to be capable of mixed mode design description. In its most probable form, it is desirable to place and connect a component into a schematic where the function of the component is described in VHDL.

Whether the design is done in VHDL, schematic, or mixed mode, *Warp3* transforms the user's captured form of the design into VHDL. From this VHDL description, the design synthesis and compilation takes place. This is important since a schematically captured design is collapsed in the synthesis process. Several layers of elementary gates are combined where possible into a single AND/OR plane, thus removing redundant gates. The final implementation may therefore look quite different than the original schematic.

Source Level Design Verification

It is very convenient to verify the functionality of the design at the VHDL source code level. Clearly, debugging at this stage saves considerable time and effort in that the design does not have to be synthesized and fitted to a device before simulation can take place. *Warp3* features a VHDL source level debugger that will simulate VHDL code and produce functional results. The results can be as graphical waveform displays, active line indication in the source code, and tabular displays of variable values. Various other debug facilities are included. The VHDL code can be conveniently debugged at this level leaving the post compilation simulation to speed optimization.

Synthesis, Optimization, and Place and Route

After the design is captured, the software produces a hardware realization of the design description. This process involves three steps: synthesis, optimization, and place and route, all of which are relatively transparent to the user. The user may interface with these processes to apply synthesis directives (constraints), or timing driven constraints for place and route.

If the design was captured with schematics, then the schematics are translated to a structural VHDL netlist. The netlist is flattened (i.e., hierarchy and intermediate nodes are removed). If the design was entered in behavioral VHDL, then it is converted into a flattened register-transfer-level netlist, which describes the interconnection of components. Behavioral constructs are translated to gates. Operator inferencing is used to instantiate arithmetic components.

Up to this point the internal design description is still device independent. Optimization is based on the target device. Different algorithms are used for different device families to produce an optimized netlist for use with the place and route software. The place and route software may perform some additional optimization, if necessary, to pack the gates into logic cells. The software then places logic cells in locations that will minimize total routing delays. After placement, routing software chooses the best path among many comparable solutions to route signals between I/O and logic cells, logic cells and logic cells, and logic cells and I/O.

Directive Driven Synthesis and Place and Route

In some cases, the designer will want to supply additional information to the synthesis and place and route processes to effect specific performance or resource utilization results. Synthesis directives can be used to provide buffering of high fanout signals, or to specify an area-optimized or performance-optimized implementation of a module (be it a counter, adder, or other arithmetic circuit). These optimization directives can help to eliminate any unnecessary delays due to either routing or the levels of logic required to implement a function in the critical path. Synthesis directives may also be used to dictate pad assignment so that high fanout signals will utilize high drive pads or clock pads as necessary. State encoding can also be affected by using a synthesis directive.

Another optimization technique often used in high-speed ASIC designs is pipelining. Pipelining allows complex functions to be performed over multiple clock cycles while operating at high speeds. Pipelining is not an option that is automatically performed by synthesis software, but is an option to the designer when capturing a design.

Place and route constraints can be used to affect place and route results. A path analysis tool within the place and route tool enables the designer to examine set-up and clock-to-output timing as well as maximum operating frequency. Constraints can be placed on specific paths in order to effect a more optimal placement of a given signal (for example, to improve the clock-to-output delay of a given signal).

Refer to the *Warp*™ documentation for a complete description of the available synthesis and place and route options.

Automatic Test Vector Generation

Some programmers are equipped to exercise a programmed device with a user supplied functional test program. Such programmers have enough hardware to permit driving and sensing all pins of a device. *Warp3* can generate test vector files for these programmers.

A Design Example

A design example is presented in order to illustrate the significant features of the pASIC380 family of FPGAs and the *Warp3* design tools. The design is a DRAM memory controller that interfaces to a system address and control bus on one side and a DRAM memory array on the other. The controller includes a slave bus interface, a DRAM address generator with burst transfer capability, and a state machine to effect the DRAM control signal timing, refresh, and bus handshake. This example was chosen because of its wide variety of implementations: a state machine, counters, registers, signal multiplexing, and decoding. This example is not meant to be a full featured DRAM controller. Certain obvious functionality is left out so as not to obscure the illustration objectives of the example. Specifically,

- There are variable splits in the row/column address multiplexing depending upon the DRAM type populating a given address range.
- The RAS/CAS that is asserted depends upon the address range occupied by a given DRAM type that happens to be present
- The column address does not use bus addresses [3:0] for a 64-bit data bus because these are byte addresses
- The actual refresh counter is left out of the synthesized design

The design is done as a high-level behavioral design with no hierarchy. The design is divided into four sections: the address and control register, the state machine, the refresh counters, and the address multiplexer. This is a natural division producing design tasks that are easy to grasp. Each section is coded in VHDL and simulated separately to verify the functionality of each section. Simulation at these simpler levels is easier to accomplish and can be more thorough than simulation of the entire design. Simulations at this level attempt to verify all details of the operation of the particular section. After each section is functionally verified, they are combined into the top-level controller description, which is then functionally simulated again except with a different focus. After the design verification, it is then compiled and performance analyzed. The following paragraphs will describe the design process, present representative samples of the outputs at each step, and highlight significant results of tool and device architecture capabilities.

Detailed Design

The entire controller is described as a single entity with a behavioral level architecture description. Each section of the design is coded as a separate process. These sections are described below. The entity declaration includes hardware-specific attribute statements. These specify the target device and particular pin assignments to take advantage of device-specific features for clock distribution and high fanout signal distribution. Other non-behavioral declarations are made later in the design for optimization purposes. In all other respects, the design is purely abstract.

The address register interfaces to the main system bus and captures the address and control information for the memory transaction. The register is synchronous. The system bus clock is applied to the register, which captures bus data on the rising edge of the system bus clock while the address strobe is

asserted. The output of the register supplies the address and control information to the rest of the circuit. The VHDL description of the register is given in *Figure 3*. The register circuitry includes handshake flags to communicate with other parts of the controller. When an address strobe is detected, a flag is set to inform the state machine that a bus transaction has started. The handshake is closed by the state machine clearing the flag, indicating that it has been recognized. The flag is necessary since, as will be seen in the later parts of the design, the state machine may not always be able to respond to (recognize) this one clock cycle event. The address register circuitry also includes comparator circuitry to see if the address is valid for the memory array. This comparator produces the match result combinatorially from the register output. There are other architectural options: making the address register a transparent latch and/or placing the address comparator inputs on the input side of the latch. For simplicity, these options are not pursued.

A refresh counter is required to provide a refresh address to the DRAM memory as well as to make request of the controlling state machine to periodically execute a refresh operation. The refresh counter does not add any instructional value to the description here. It is therefore not included in the example details. The refresh circuitry provides a refresh request to the state machine and a refresh address to the address multiplexer. These are set to zero in the example VHDL source code.

```
adreg: process
    begin
      wait until clk = '1';
      if(reset = '1') then
        as_flag <= '0';
      elsif(as = '0') then
        as_flag <= '1';
      elsif(clr_as = '0') then
        as_flag <= '0';
      endif;

      if(as = '0') then
        stored_ad <= address;
        burst_stored <= burst;
      end if;
    end process;
```

Figure 3. VHDL Behavioral Description of Address Register

The column address counter produces addresses for a burst transaction. In these transactions the system provides the first address of the burst and the remaining addresses are expected to be generated by the slave. The column address counter is loaded, under control of the state machine, with the low-order portion of the incoming address and is advanced also under control of the state machine. The counter also serves to hold the column address in those cases where a new address is received and the state machine is not yet finished with the present address. The counter is a fixed width for a given transaction burst length. The counter provides an address which wraps around the maximum length of the counter. This is consistent with burst transactions that do not start the transaction at the lowest address of the items in the data burst. The counter is configured to count in Intel order (refer to the *i486 Hardware Reference Manual*).

The address multiplexer is combinatorial logic that produces twelve address outputs for the DRAM array. Under control of the state machine, the multiplexer can output the refresh counter address, the column address, or the row address. The column address is the lowest-order bits of the address contained in the address register. The row address is a selected set of the upper-order bits of the address contained in the address register. The particular upper-order bits depend upon the DRAM type used in the array.

The state machine controls all actions of the DRAM controller including the timing for the DRAM access. The state diagram is given in *Figure 4* and the VHDL code is given in *Figure 5*. The state machine is designed so that the outputs come directly from a dedicated flip-flop, principally so that the bus acknowledge back to the system has a favorable set-up time for high-speed buses. Upon recognizing the assertion of the address flag from the address register, the state machine advances to the next state where the validity of the address is checked. The additional clock cycle from the receipt of the address flag to the test of the address validity allows

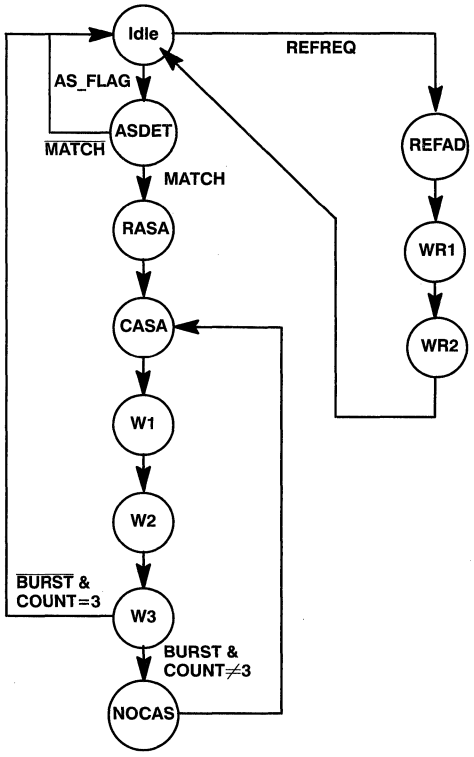


Figure 4. Controller State Machine State Diagram

time to test the address after it has entered the address register. If the upper address bits are 0, the access is taken to be valid for the DRAM memory and the state machine advances to start the memory cycle. If there is no address match, the state machine asserts the address flag clear signal to prepare for a subsequent transaction.

From the state machine *IDLE* state, the address multiplexer is set to select the upper portion of the stored address (the ROW address). With an address strobe detected, the state machine advances to the *ASDET* state where the address match is examined. If there is no match, the state machine returns to the *IDLE* state and if there is a match, the state machine advances to the *RASA* state. Here the state machine asserts RAS and simultaneously causes the address multiplexer to select the lower portion of the stored address (the COLUMN address). In the next cycle, the state machine advances to the *CASA* state where the DRAM CAS signal is asserted. The next cycles, W1, W2, and w3, are wait states keeping CAS and RAS asserted for the required time given in the DRAM data sheets. Depending upon the clock speed, W1 and/or W2 may be omitted. In state W3, a decision is made to return to the idle state if no burst is to be performed, or to advance to the state *NOCAS* if the transaction is a burst. In the *NOCAS* state, CAS is deasserted and then the state machine then loops back to the *casa* state to continue with the burst. The burst is ended with an examination of the burst counter in state w3. If the burst is completed, the state machine returns to the *IDLE* state. Refreshes are performed by advancing from the *IDLE* state to the *REFAD* state. Here, the address multiplexer is directed to output the address from the refresh address counter. In the next states, RAS is asserted for three cycles: *REF_RAS*, *WR1*, and *WR2*. Refresh is completed, and the state machine returns to the idle state. In the IDLE state the code gives priority to refresh requests over bus related requests (address strobe detection).

Although this is a high-level behavioral design, some aspects of the hardware have been taken into account. The statement

```
ATTRIBUTE state_encoding OF state:type IS
    ONE_HOT_ONE;
```

directs the synthesizer to use one flip-flop per state for the state machine. A portion of the synthesized logic is shown in *Figure 6*. Note the simplicity of the resulting logic equations. Outputs and transition terms include only the one flip-flop for the state in question. This results in a great simplification of the logic since complex state decoding is not required. Although this approach uses more flip-flops than encoded states, the decoding is far simpler. Most states and the related decoding should be realizable in one logic cell.

The state machine, by virtue of the VHDL coding, forces the outputs to be derived directly from the output of a flip-flop. This gives signals such as BACK (bus acknowledge) a short clock-to-output

```
-- State Machine process

     control:process
     begin
       wait until clk_in = '1';
       if(rs_in = '1') then
         state <= idle;
         cas_en <= '1';
         ras_en <= '1';
         back <= '1';
         col_sel <= '0';
         ref_sel <= '0';
       else
       case state is
         when idle =>
           cas_en <= '1';
           ras_en <= '1';
           back <= '1';
           if (ref_req = '1') then
             state <= refad;
             ref_sel <= '1';
           elsif (as_flag = '1') then
             state <= asdet;
             clr_as <= '1';
           end if;
         when asdet =>
           clr_as <= '0';
           if (match = '1') then
             state <= rasa;
             ras_en <= '0';
             col_sel <= '1';
             burst_flag <= burst_stored;
           else
             state <= idle;
           end if;
         when rasa =>
             state <= casa;
             cas_en <= '0';
             --back <= '0';

         when casa =>
             state <= w1;
             --back <= '1';

         when w1 =>
             state <= w2;
```

Figure 5. State Machine Behavioral Description

```
            when w2 =>
                 state <= w3;
                 back <= '0';

            when w3 =>
              cas_en <= '1';
              back <= '1';
              if(burst_flag = '1' and bst_cnt /= "11") then
                state <= nocas;
                col_sel <= '1';
              else
                state <= idle;
                ras_en <= '1';
                col_sel <= '0';
              end if;

            when nocas =>
                 state <= casa;
                 cas_en <= '0';
    -- Refresh
            when refad =>
                 state <= wr1;
                 ras_en <= '0';

            when wr1 =>
                 state <= wr2;
            when wr2 =>
                 state <= idle;
                 ref_sel <= '0';
                 ras_en <= '1';

          end case;
          end if;
        end process;
```

Figure 5. State Machine Behavioral Description (continued)

propagation delay, allowing it to meet the requirements of a high speed bus.

Design Analysis

The final step in the design process is to analyze the device performance and make adjustments to the

```
controller_0_state_bv_4_.D =
   controller_0_state_bv_3_.Q
 + controller_0_state_bv_8_.Q * burst_flag.Q * /bst_cnt_0__BEH_i42_0_DFF.Q
 + controller_0_state_bv_8_.Q * burst_flag.Q * /bst_cnt_1__BEH_i42_0_DFF.Q
```

Figure 6. A Portion of the Synthesized State Machine Logic Equations (State CASA)

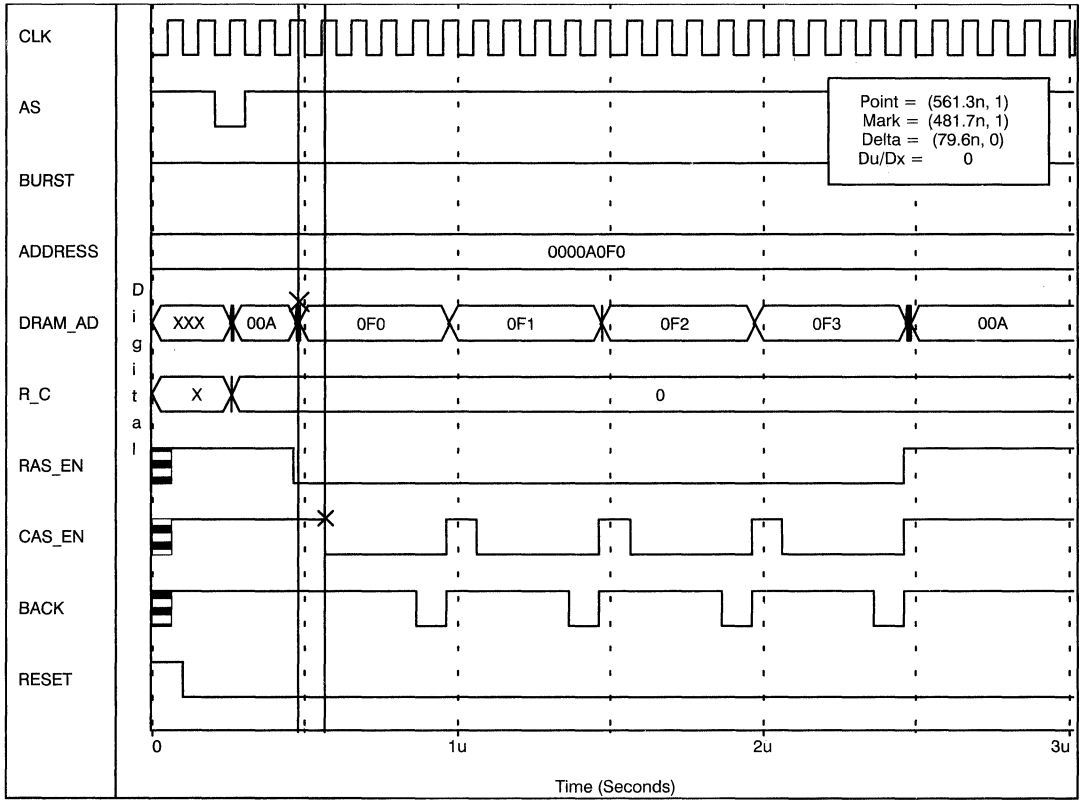


Figure 7. Graphic Output of the Simulator

design to optimize certain speed paths. At this point, the functionality of the device should be correct per the design specification as verified by the VHDL source level debugger. The performance optimization is accomplished using the timing analyzer. Timing measurements can be made in the simulation as well. There are several concerns in this design: the set-up time for the address and control information to the input register, the clock-to-output delay for the BACK signal, and the delay in the column address output. This latter concern is to determine if the multiplexing from the row to the column address (in state *rasa*) will cause the column address to be too close to the assertion of CAS (state *casa*) potentially violating the DRAM column address set up time. *Figure 7* illustrates a timing measurement being made in the simulation environment. The figure shows one burst transaction. A Mark and Point are placed at the falling edge of CAS_EN (the CAS enable output) and the place where the DRAM address (DRAM_AD) has switched to the column address and stabilized. The small window displays the Mark and Point times and DELTA is the time difference between these points.

As an example of an optimization the clock to output of the BACK signal is examined. *Table 2* shows the output of the Path Analyzer. The table shows a timing analysis result selecting the flip-flop to PAD options in the analyzer. The flip-flop to BACK path is then selected in the table of signal delays. The tabular results show a delay of 6.3 ns from the BACK

flip-flop output to the pad. The schematic-like representation (the physical view) of the device shows the generalized placement and signal routing in the device. The Physical View is shown in *Figure 8*. The selection of the flip-flop output to PAD path in the analyzer results table has caused that path to be highlighted in the schematic.

An improvement in this delay is sought by placing a timing constraint in the analyzer results table and rerunning the place and route. *Table 2* shows the new result after the place and route is rerun. The new place and route Physical View results are shown in *Figure 9*. Comparing *Figures 8* and *9* it can be observed that the flip-flop storing BACK has been moved closer to the pad resulting in a near 1 ns improvement in the clock to output delay for this signal. The analyzer shows that the variation in the clock pad to flip-flop delay is less than 100 ps, therefore it is not fruitful to attempt to improve the clock pad to BACK flip-flop delay. The other signal paths are similarly analyzed and optimized if necessary.

After the optimization is completed, the final design is simulated. There are two ways to define the stimulation for the simulation: graphical and textual. For designs with wide bus inputs or a large number of inputs, the most convenient method is textual input. The textual input of commands to drive the simulation is shown in *Figure 10*. Note in particular the command lines beginning with 'wfm'. These lines describe the waveform of the input signals. Of particular interest is the line beginning with wfm address, which describes the input address signal. The entire vector, address, can be assigned values in hex—a task which would be very cumbersome in a graphic input only environment. This has been done for the simulation result shown in *Figure 7*. The figure shows the graphical results of the simulation output along with the input stimuli.

Table 2. SpDE Path Analyzer

Path #	Delay	Delay Path	Constraint
−1−	4.0	RAS_CAS_3_−I2 −− RAS_CAS_3_	
−2−	4.0	RAS_CAS_0_−I2 −− RAS_CAS_0	
−3−	4.0	RAS_CAS_1_−I2 −− RAS_CAS_1	
−4−	4.3	RAS_CAS_2_−I2 −− RAS_CAS_2	
−5−	5.7	RAS_EN−I2 −− RAS_EN	
−6−	6.0	CAS_EN−I2 −− CAS_EN	
−7−	6.3	BACK−I2 −− BACK	
−8−	6.8	STORED_AD_19_ −− RC_AD_7_	
−9−	7.0	STORED_AD_16_ −− RC_AD_4_	
−10−	7.0	STORED_AD_23_ −− RC_AD_11_	
−11−	7.0	STORED_AD_21_ −− RC_AD_9_	
−12−	7.1	STORED_AD_15_ −− RC_AD_3_	
−13−	7.2	STORED_AD_13_ −− RC_AD_1_	
−14−	7.4	STORED_AD_14_ −− RC_AD_2_	
−15−	7.4	STORED_AD_18_ −− RC_AD_6_	
−16−	7.5	STORED_AD_20_ −− RC_AD_8_	
−17−	7.8	STORED_AD_12_ −− RC_AD_0_	
−18−	7.8	STORED_AD_22_ −− RC_AD_10_	
−19−	8.0	STORED_AD_17_ −− RC_AD_5_	
−20−	8.7	STORED_AD_19_ −− RC_AD_7_	
−21−	8.8	STORED_AD_23_ −− RC_AD_11_	
−22−	8.8	STORED_AD_16_ −− RC_AD_4_	
−23−	8.8	STORED_AD_21_ −− RC_AD_9_	
−24−	8.9	COL_AD_9_ −− RC_AD_9_	

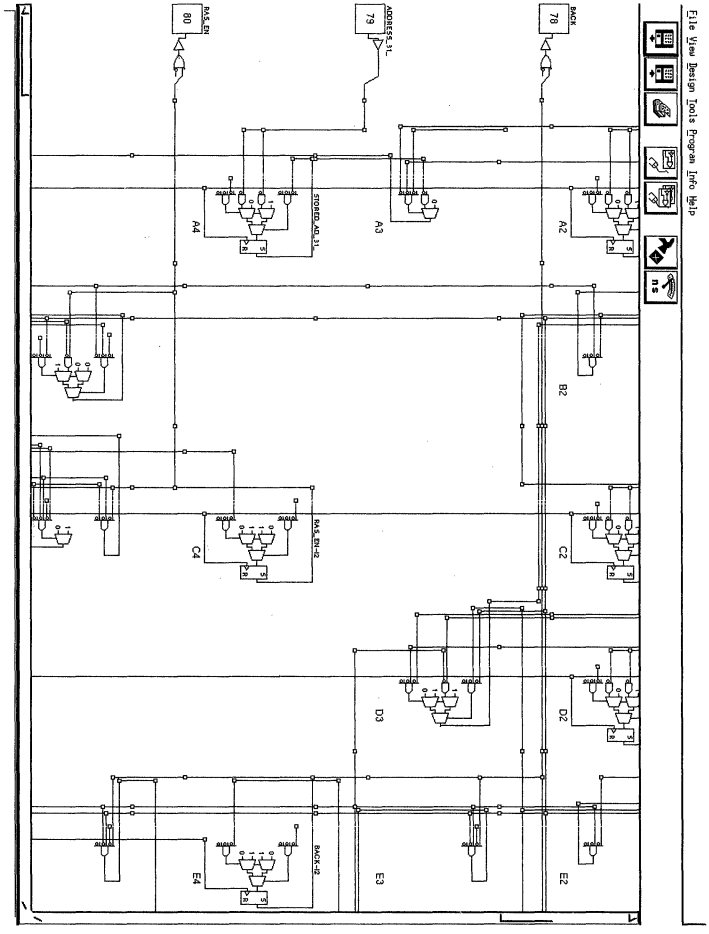


Figure 8. Physical View of Initial Design

Table 3. SpDE Path Analyzer with Applied Constraint

Path #	Delay	Delay Path	Constraint
−1−	4.0	RAS_CAS_3_−I2 −− RAS_CAS_3_	
−2−	4.0	RAS_CAS_0_−I2 −− RAS_CAS_0	
−3−	4.0	RAS_CAS_1_−I2 −− RAS_CAS_1	
−4−	4.3	RAS_CAS_2_−I2 −− RAS_CAS_2	
−5−	5.5	BACK−I2 −− BACK	4.0
−6−	5.7	RAS_EN−I2 −− RAS_EN	
−7−	6.0	CAS_EN−I2 −− CAS_EN	
−8−	6.8	STORED_AD_19_ −− RC_AD_7_	
−9−	7.0	STORED_AD_16_ −− RC_AD_4_	
−10−	7.0	STORED_AD_23_ −− RC_AD_11_	
−11−	7.0	STORED_AD_21_ −− RC_AD_9_	
−12−	7.1	STORED_AD_15_ −− RC_AD_3_	
−13−	7.2	STORED_AD_13_ −− RC_AD_1_	
−14−	7.4	STORED_AD_14_ −− RC_AD_2_	
−15−	7.4	STORED_AD_18_ −− RC_AD_6_	
−16−	7.5	STORED_AD_20_ −− RC_AD_8_	
−17−	7.8	STORED_AD_12_ −− RC_AD_0_	
−18−	7.8	STORED_AD_22_ −− RC_AD_10_	
−19−	8.0	STORED_AD_17_ −− RC_AD_5_	
−20−	8.7	STORED_AD_19_ −− RC_AD_7_	
−21−	8.8	STORED_AD_23_ −− RC_AD_11_	
−22−	8.8	STORED_AD_16_ −− RC_AD_4_	
−23−	8.8	STORED_AD_21_ −− RC_AD_9_	
−24−	8.9	COL_AD_9_ −− RC_AD_9_	

Summary

This application note is an introduction to FPGAs and high-level design tools. Specifics of the Cypress pASIC380 FPGA family of devices were presented along with the *Warp3* design tool set. A DRAM memory controller was presented to illustrate the global flow of a complete development. This was not meant to be a design tutorial for the design tools, schematic entry, VHDL encoding or design optimization, but rather it is intended as an overall map as to how a designer would proceed and the options available. The pASIC380 family FPGAs and the *Warp3* tool set is a powerful combination of device architecture and development tool that can help a designer achieve design success in a very short period of time.

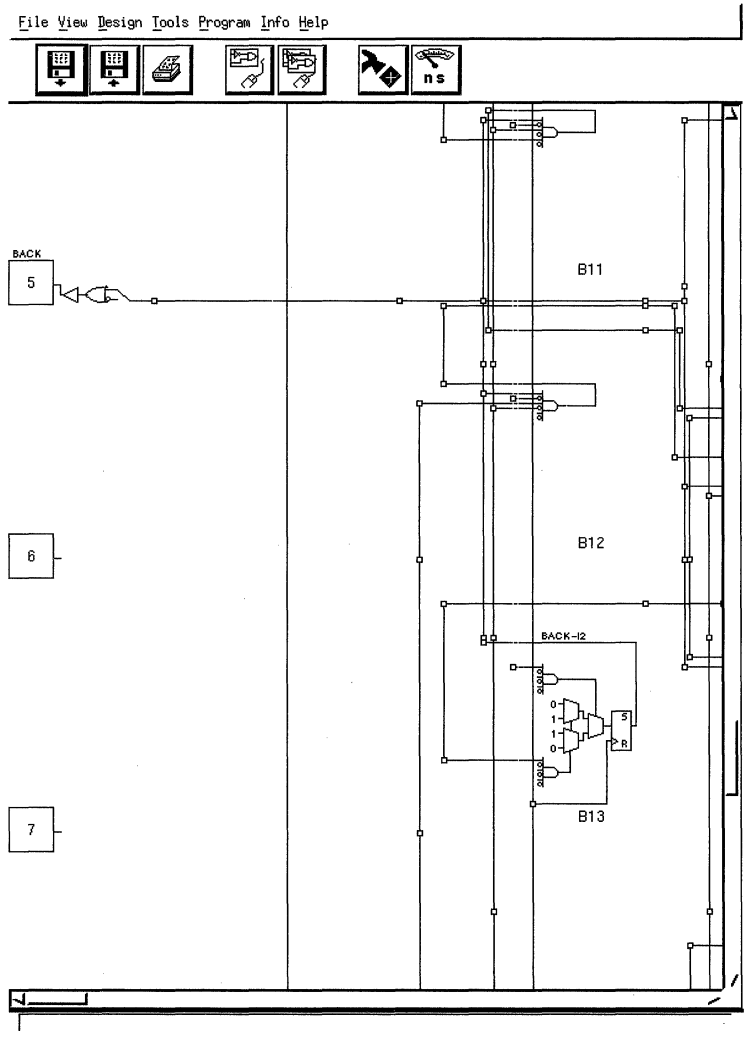


Figure 9. Physical View of Design Re−Placed and Routed with Constraint

```
| controller command file
logfile controller.log
stepsize 50ns
defaults -bignet -cmdfile -time
wave
vector address address[31:0]
radix hex address

vector dram_ad RC_AD[11:0]
radix hex dram_ad

vector r_c RAS_CAS[3:0]
radix hex r_c

wave controller.wfm clk as burst address dram_ad r_c ras_en cas_en back
       reset

clock clk 0 1

wfm reset @0=1 100ns=0
wfm as @0=1 200ns=0 100ns=1
wfm burst @0=1
wfm address @0=a0f0\h
cycle 200
log
```

Figure 10. Simulation Command File

Appendix A. Complete Design Behavioral Description

```
entity controller is
  port(clk,burst,as,reset: in bit;
    address: in bit_vector(31 downto 0);
    back,ras_en,cas_en: out bit;
    ras_cas: out bit_vector(3 downto 0);
    rc_ad: out bit_vector(11 downto 0));
attribute part_name of controller:entity is "C384";
end controller;

use work.bv_math.all;
use work.rtlpkg.all;

architecture behavior of controller is

  type name is (idle,asdet,rasa,casa,w1,w2,w3,nocas,refad,wr1,wr2);
  ATTRIBUTE state_encoding OF name: type IS ONE_HOT_ONE

  signal state: name;
  signal bst_cnt: bit_vector(1 downto 0);
  signal col_sel, ref_sel, as_flag, burst_flag, burst_stored, match,
      clr_as, ref_req, clk_in, ck_x, ck_y, as_x, as_in, rs_x, rs_in: bit;
  signal stored_ad: bit_vector(31 downto 0);
  signal re_ad, col_ad: bit_vector(11 downto 0);

  alias top_ad: bit_vector(3 downto 0) is stored_ad(31 downto 28);
  alias row_ad: bit_vector(11 downto 0) is stored_ad(23 downto 12);
  alias bank: bit_vector(3 downto 0) is stored_ad(27 downto 24);

begin

-- Special IO ports for device

  ck1:PAckcell
    PORT MAP(clk, ck_x, ck_y, clk_in);

  hd1:PAincell
    PORT MAP(as, as_x, as_in);

  hd2:PAincell
    PORT MAP(reset, rs_x, rs_in);

-- Address register process

  adreg:process
  begin
    wait until clk_in = '1';
```

Appendix A. Complete Design Behavioral Description (continued)

```
    if(rs_in = '1') then
      as_flag <= '0';
    elsif(as_in = '0') then
      as_flag <= '1';
    elsif(clr_as = '1') then
      as_flag <= '0';
    end if;

    if(as_in = '0') then
      stored_ad <= address;
      burst_stored <= burst;
    end if;

  end process;

-- Match Comparator

  match <= '1' when top_ad = "0000" else '0';

-- DRAM address multiplexer
  re_ad <= "000000000000";
  ref_req <= '0';

  mux:process(col_ad,col_sel,row_ad,re_ad,state)
  begin
    if(state = refad or state = wr1 or state = wr2) then
      rc_ad <= re_ad;
    elsif(col_sel = '1') then
      rc_ad <= col_ad;
    else
      rc_ad <= row_ad;
    end if;
  end process;

-- Encoded RAS / CAS select

  ras_cas <= bank;

-- Column Address, Intel Order

  col_ad(11 downto 2) <= stored_ad(11 downto 2);
  col_ad(1) <= stored_ad(1) xor bst_cnt(1);
  col_ad(0) <= stored_ad(0) xor bst_cnt(0);

-- State Machine process

  control:process
  begin
    wait until clk_in = '1';
```

Appendix A. Complete Design Behavioral Description (continued)

```
if(rs_in = '1') then
  state <= idle;
  cas_en <= '1';
  ras_en <= '1';
  back <= '1';
  col_sel <= '0';
  ref_sel <= '0';
else
case state is
  when idle =>
    cas_en <= '1';
    ras_en <= '1';
    back <= '1';
    if (ref_req = '1') then
      state <= refad;
      ref_sel <= '1';
    elsif (as_flag = '1') then
      state <= asdet;
      clr_as <= '1';
    end if;
  when asdet =>
    clr_as <= '0';
    if (match = '1') then
      state <= rasa;
      ras_en <= '0';
      col_sel <= '1';
      burst_flag <= burst_stored;
    else
      state <= idle;
    end if;
  when rasa =>
      state <= casa;
      cas_en <= '0';
      --back <= '0';

  when casa =>
      state <= w1;
      --back <= '1';

  when w1 =>
      state <= w2;

  when w2 =>
      state <= w3;
      back <= '0';
```

Appendix A. Complete Design Behavioral Description (continued)

```
      when w3 =>
        cas_en <= '1';
        back <= '1';
        if(burst_flag = '1' and bst_cnt /= "11") then
          state <= nocas;
          col_sel <= '1';
        else
          state <= idle;
          ras_en <= '1';
          col_sel <= '0';
        end if;

      when nocas =>
          state <= casa;
          cas_en <= '0';
      when refad =>
          state <= wr1;
          ras_en <= '0';

      when wr1 =>
          state <= wr2;
      when wr2 =>
          state <= idle;
          ref_sel <= '0';
          ras_en <= '1';

    end case;
    end if;
  end process;

-- Burst counter

  burst_count:process
  begin
    wait until clk_in = '1';
    if(state = idle) then
      bst_cnt <= "00";
    elsif(state = w3) then
      bst_cnt <= inc_bv(bst_cnt);
    end if;
  end process;

end behavior;
```

PCI Bus Applications on FPGAs

Introduction

The Peripheral Component Interconnect (PCI) bus is a high-bandwidth, "plug-and-play" bus designed to meet the performance demands of the peripherals of today's high-performance PCs and workstations and their large bandwidth applications. It is rapidly becoming widely accepted in the computer industry as it opens doors to performance demanding applications such as video and audio systems, graphics accelerator boards, 3D native signal processing, network adapters, data acquisition, and data storage devices. Development of PCI products requires strict adherence to the PCI Local Bus Specification.

Continuous evolution of the PCI specification and specific needs of each application demand a flexible PCI solution. This makes programmable logic in general and FPGAs in particular ideal candidates for the PCI interface. Designing a PCI interface can take several man-months. It is the intention of this application note to provide an overview of the PCI bus and its associated transactions, and to present an example design for a PCI target device that has been implemented in a Cypress FPGA. This note covers the basics of PCI, an example PCI target design, and design issues a PCI designer will encounter. The PCI design files may be obtained by contacting the Applications Group at (408) 943–2456.

PCI Bus

The PCI spec 2.1 specifies the PCI operating speed to be 0 to 33 MHz with 32-bit synchronous bus, expandable to 64 bits. 66-MHz PCI bus speed is also specified to allow future migration. The PCI has a potential transfer rate of 132 MB/s. This value will double/quadruple when the bus is expanded to 64 bits or/and the speed is increased to 66 MHz. PCI is also specified at both 5-volt and 3.3-volt operations, and is processor independent. All PCI devices have a "configuration space" that enables PCI to be a "plug-and-play" solution. Configuration of add-in boards and components is done automatically through software.

PCI Architecture

The PCI bus is the backbone of the I/O and memory devices of the computer (see *Figure 1*). Processor independent, the PCI bus is accessed by the CPU via a CPU local bus to PCI bridge device. The I/O and memory devices hang off the PCI bus and transact in an initiator/target (master/slave) relationship.

PCI Interface Signals

A PCI interface device must have 47 pins. A PCI initiator device has 2 additional pins, which brings the total number of required pins to 49. Optional pins provide 64-bit operation, JTAG boundary scan, target locking, cache support, and interrupt expandability to the PCI bus (see *Figure 2*).

There are five different types of PCI signals:

in	input only signal
out	output only signal
t.s.	bidirectional, three-state input/output pin
s.t.s.	sustained three-state signal; an active LOW signal driven by one agent at a time and must be precharged HIGH before floating. A pull-up resistor is provided by the central resource to sustain the signal in the HIGH state.
o.d.	open drain signal so multiple devices share this signal as a wired-OR

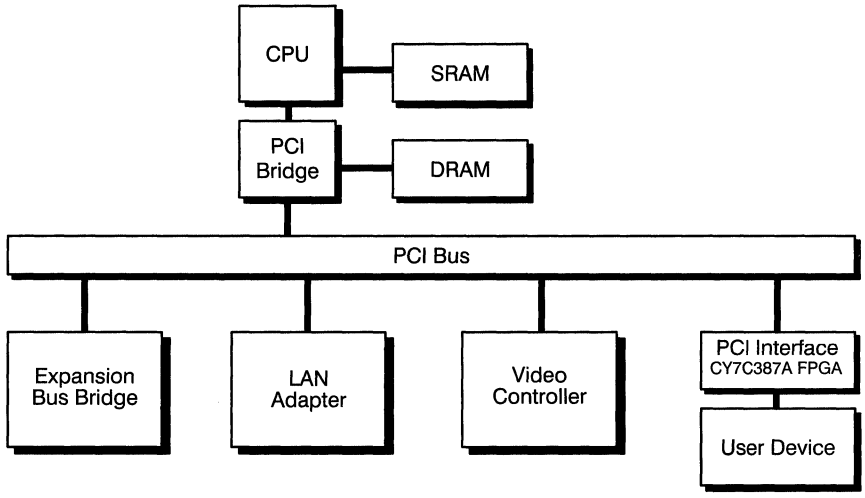


Figure 1. PCI Architecture

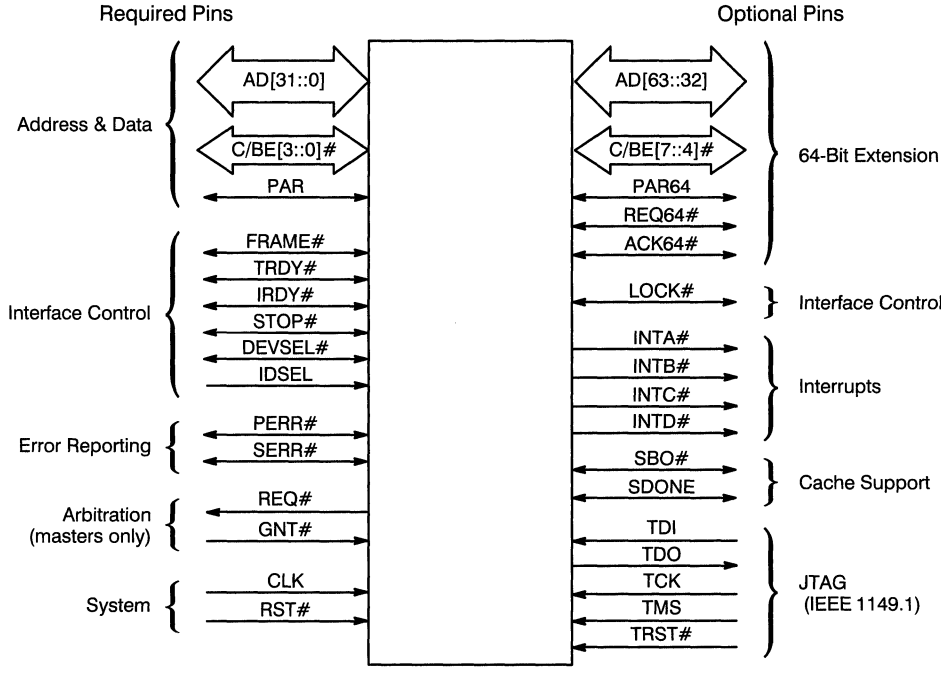


Figure 2. Pin Diagram

Table 1. Required Pins

Pin Name	Type	Description
AD[31:00]	t.s.	32-bit bidirectional multiplexed address/data bus
C/BE[3:0]#	t.s.	Byte enables for the four bytes of the 32-bit AD line
PAR	t.s.	Parity bit for even parity over AD and C/BE lines
FRAME#	s.t.s.	Indicates the duration of a transaction
TRDY#	s.t.s.	Target ready signal, indicates that the target is ready to perform a data transfer
IRDY#	s.t.s.	Initiator ready signal, indicates that the initiator is ready to perform a data transfer
STOP#	s.t.s.	Target signal to induce retry, disconnect, or abort
DEVSEL#	s.t.s.	Target signal to claim the current transaction on the bus
IDSEL	in	Individual device selector signal
PERR#	s.t.s	Parity error during the data phase
SERR#	o.d.	Parity error during the address phase or special cycle
REQ#	t.s.	Initiator bus request arbitration signal
GNT#	t.s.	PCI bus arbiter grant signal to requesting initiator
CLK	in	PCI system clock
RST#	in	PCI system reset signal

Table 2. Optional Pins

Pin Name	Type	Description
AD[63:32]	t.s.	64-bit address/data extension pins
C/BE[7:4]#	t.s.	64-bit byte enable extension pins
PAR64	t.s.	64-bit parity bit
REQ64#	t.s.	Initiator 64-bit bus request arbitration signal
ACK64#	t.s.	PCI bus arbiter 64-bit grant signal to requesting initiator
LOCK#	s.t.s.	Target locking signal
INTA–D#	o.d.	Interrupt pins

Note:

PCI also supports two optional pins for cache support and five optional pins for JTAG support.

PCI Bus Commands

PCI initiators begin a transaction by placing a command on the bus. This command defines what action will be performed during the current transaction. *Table 3* shows all PCI bus commands and their 4-bit values.

Table 3. PCI Bus Commands

C/BE[3:0]#	Command
0000	Interrupt Acknowledge
0001	Special Cycle
0010	I/O Read
0011	I/O Write
0100	Reserved
0101	Reserved
0110	Memory Read
0111	Memory Write
1000	Reserved
1001	Reserved
1010	Configuration Read
1011	Configuration Write
1100	Memory Read Multiple
1101	Dual Address Cycle
1110	Memory Read Line
1111	Memory Write and Invalidate

PCI Configuration Space

The configuration space, a required feature of all PCI devices, is what makes PCI a plug-and-play solution. During system configuration, the PCI bus is scanned to determine the configuration requirements for all agents on the bus. All PCI devices must implement 256 bytes of configuration space which holds configuration information such as device identification, device status, functionality enables, and base address registers for address space assignments.

Configuration Space Header

The first 64 bytes of the 256 byte configuration space are known as the configuration header. This application note describes the header currently used for most I/O and memory devices, the type 00 header (shown in *Figure 2*).

Device ID – Device identification number issued by the vendor.

Vendor ID – Vendor identification number issued by the PCI SIG.

Revision ID – Device-specific revision identification number issued by the vendor.

Header Type – Identifies the layout of the second part of the predefined 64-byte header.

Class Code – Identifies the generic function of the device.

Base Address Register – Register for address space location assignment.

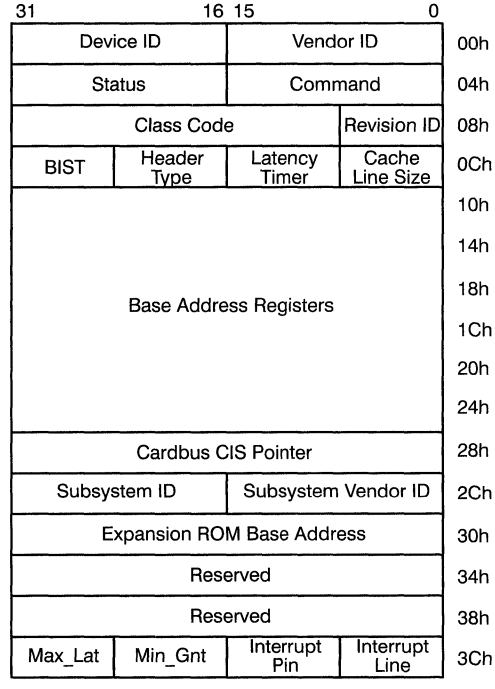


Figure 3. Type 00h Configuration Space Header

The latter 192 bytes of the 256 bytes of configuration space are a device-dependent region. A PCI-compliant device does not have to implement unused portions of the configuration space as registers. However, a value of zero must be returned when unused locations are read.

The 32-bit-wide register lines in the configuration space are addressed on 32-bit word boundaries. Hence the register sequence (see the right side of *Figure 2*) is 00h, 04h, 08h, 0Ch, etc. The 32-bit registers comprise four bytes, each of which may be accessed when the corresponding Byte Enable is asserted.

Address Space

A PCI device's address space is relocatable. The system assigns areas of address space by writing address values to the device's Base Address registers. The amount of address space a device needs is also determined by examination of the Base Address registers within the configuration space of that device. To determine how much address space a device on the bus requires, the system writes the value xFFFFFFFF to the Base register, and then reads the register. The number of zeroes returned in the least significant position determines how much address space the device requires. For example, if a device returns the value xFFFFFF80, the arbiter knows that this device requires 128 bytes (7 zero bits, 2 ^ 7 = 128).

The zeroes in the least significant positions of the Base Address registers should be implemented as hard-wired zeroes. More hard-wired zeroes provide a larger amount of address space with a smaller number of bits to compare to determine an address hit. In contrast, less hard-wired zeroes translate to a smaller amount of address space for the device, but more bits to compare to determine an address hit. For some devices, the number of bits to compare to determine an address hit affects how fast a PCI device can claim a transaction as a target.

Transaction Waveforms

All PCI read/write transactions are inherently burst transfers. The length of the burst is determined by the FRAME# signal provided by the initiator (master) of the transaction. Transactions begin with a single address phase followed by one or more data phases.

Figure 4 shows a basic read operation. Prior to clock 1, the initiator is assumed to have arbitrated for control of the bus, and has received permission to use the bus for a transaction. After clock 1, the initiator places the address of the desired device and the command for the device on the bus while asserting the FRAME# signal. On clock 2, because FRAME# is sampled LOW for the first time, all devices on the PCI bus are required to latch in the address and command on the bus, and begin decoding the address to determine transaction ownership. After clock 2, the initiator (master) waits for a target (slave) device to respond and claim the transaction by asserting the DEVSEL# signal.

Because this is a read transaction, the target is required to wait a clock to induce a turn around cycle on the A/D bus to prevent contention of the bus as control switches from initiator to target.

Data transactions are controlled by three signals: FRAME#, IRDY#, and TRDY# signals. (See signal description above for definition of signals.) The actual transfer of data occurs only on clocks where both IRDY# and TRDY# signals are asserted. If either or both signals are not asserted, then a wait state occurs.

After clock 3, the target is ready to provide the first piece of data, and the initiator is ready to receive it. Both the IRDY# and TRDY# signals are asserted, and on clock 4 a data transfer takes place. Also on this clock, because the target senses that the FRAME# signal is still asserted, it knows that the transaction is not complete and the initiator expects more data. On the next clock (clock 5), the target is not ready (TRDY# deasserted), and so a wait state is induced. On clock 6, a data transfer occurs since both ready signals are asserted. On clock 7 the initiator is not ready, so the IRDY# signal is deasserted, inducing a wait state. The initiator knows that it desires only one more piece of data, and when the IRDY# signal is asserted on clock 8, the FRAME# signal is deasserted. A data transfer takes place on this clock since the TRDY# signal is also asserted. Also on clock 8, the target samples the FRAME# signal. Since the FRAME# signal is deasserted, the

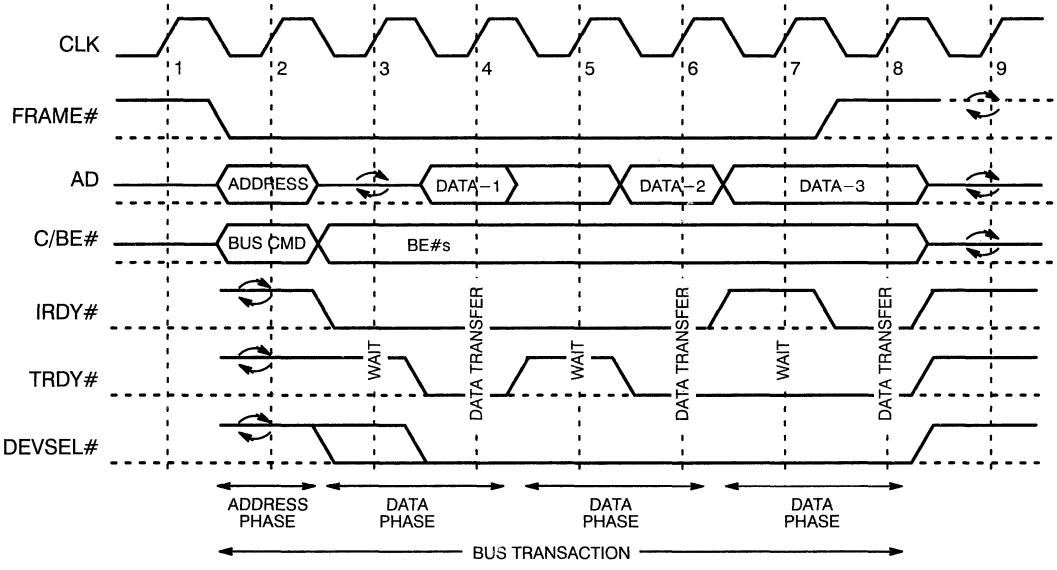


Figure 4. Read Transaction Waveform

target device is informed that the transaction is over. On clock 9, FRAME#, A/D bus, and C/BE bus are turned around for one cycle, and the control signals are precharged HIGH before being three-stated. This completes the read operation.

Once a ready signal is asserted, it may not be deasserted until the data transfer takes place or the transaction is aborted.

Figure 5 shows a basic write operation. The rules of transaction are exactly the same as the read operation, with the exception that a turn around cycle on the A/D bus right after the address phase is unnecessary since the same agent controls the bus for the entire duration of the transaction.

Claiming the Transaction

Not all PCI devices can capture the address, decode it, and claim the transaction within a single clock after an initiator begins a transaction. PCI targets may take up to 3 clocks after the initial address phase to assert the DEVSEL signal (see *Figure 6*). In *Figure 6*, the address phase occurs at clock 2. If a target can assert its DEVSEL# signal by clock 3, it is considered a "fast" response device. Assertion of DEVSEL# on clock 4 would be "medium" and clock 5 would be "slow." The sixth clock is reserved for subtractive decoding devices. If a target device has not asserted DEVSEL# by the sixth clock, the initiator may terminate the transaction.

Parity

Parity generation is required for all PCI devices. In general, parity checking is usually required. On read transactions, it is the responsibility of the target to generate parity. On write transactions, parity generation is the responsibility of the initiator.

Parity in PCI is even parity over the AD bus, C/BE bus, and the parity line. The generated parity bit is available one clock after valid values on the buses are transferred. A parity error is reported two clock cycles after the valid values have been transferred (i.e., one clock after the parity bit was available). Because parity is calculated over the entire AD bus, the signals on the AD bus must be held stable even if they are undefined.

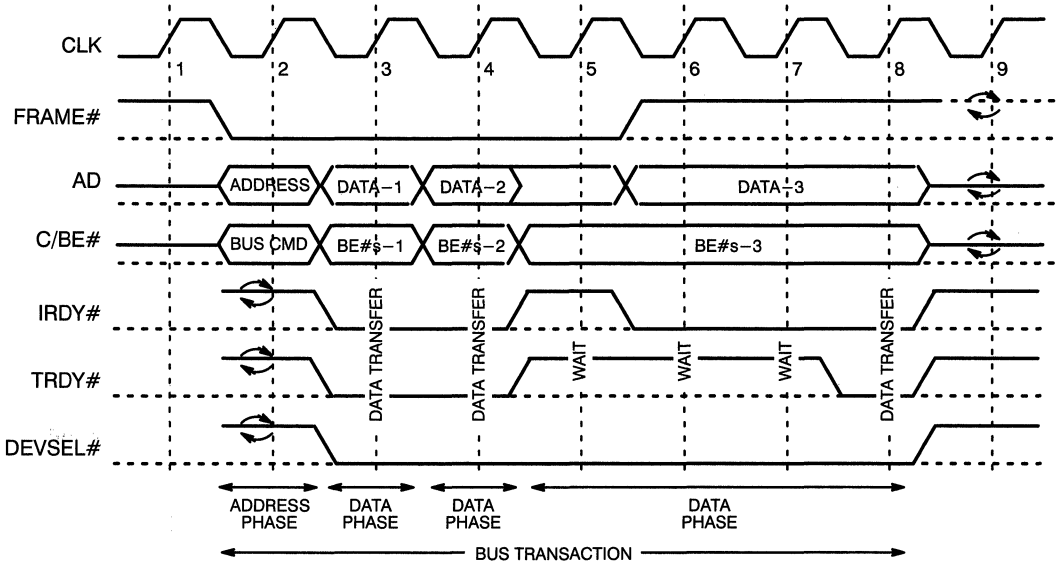


Figure 5. Write Transaction Waveform

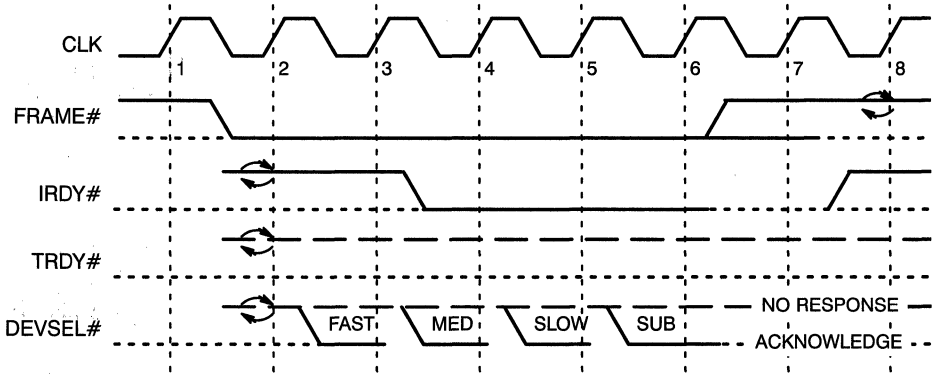


Figure 6. Transaction Claiming Speed

Aborting the Transactions

PCI provides a method for premature transaction termination.

There are three scenarios when an initiator may terminate a transaction.

1. The transaction has completed normally, and so the initiator ends the transaction.
2. The initiator's latency timer has expired and arbitrator has deasserted the initiator's GNT# signal. The initiator is allowed one last data transfer once the latency time-out is sensed.
3. No target has responded to an initiator request within five clock cycles after FRAME# was asserted. The initiator will end the transaction in the sixth clock.

There are three types of target terminations:

1. disconnect – When a data phase is very long, the target may induce a disconnect to free the bus. To signal a disconnect, the target must assert both STOP# and TRDY#. One last data transfer takes place, and the initiator ends the transaction.

2. retry – If a target cannot respond to the current transaction at the current time, the target may signal a retry, indicating to the initiator to try the transaction again at a later time. For example, if a target is currently locked for exclusive access by another initiator, then the target would signal a retry. In a retry, no data is transferred. A target can signal a retry by asserting the STOP# signal and keeping the TRDY# signal deasserted.

3. target-abort – If a target encounters a fatal error, then the device may signal an abort. The abort is signalled by asserting STOP# and deasserting DEVSEL#. The TRDY# signal must also be kept deasserted.

Recommended Device Pinout

The PCI spec recommends the pinout shown in *Figure 7.*

A PCI Target Application

To introduce designing for PCI applications, a target PCI implementation is presented. This example design can be modified to suit any specific needs.

Design Overview

1) The Features

The first step is to decide what features the PCI Target interface is going to have. A PCI-compliant interface with the following features is desired.

- 0 to 33 MHz bus clock speed operation
- 32-bit Addr/Data bus
- Burst cycles
- Wait state support
- Fully customizable address space size: 1 byte to 4 Gbytes
- Two base address registers (more may be implemented if necessary)
- Configuration, I/O, and Memory read and writes
- Parity generation, with checking option
- Target-abort and retry support

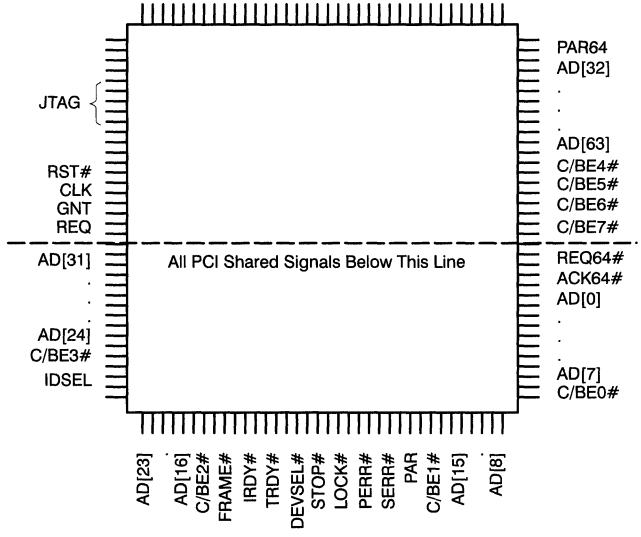


Figure 7. Recommended Pinout

- State machines and configuration space implemented in VHDL for easy high-level user modification
- Generic back-end user interface

2) Handling the Address Phase

The target needs to latch the data on the first cycle that the FRAME# signal is sampled LOW. The PCI specification allows both the address and FRAME# signals only a 7-ns set-up time. In some cases, the logic necessary to determine that FRAME# has transitioned to the asserted state and then enable all 36 bits to the register would take longer than 7 ns. To make it more robust, it was decided to put an input register on the A/D bus that would latch data every clock tick. In parallel, the asserted FRAME# signal would "wake" the PCI state machine. In this manner, the device will have the address stable for an entire clock cycle. A second register is needed to memorize the address, command, and IDSEL lines.

Address compare logic is needed to compare the latched address with all implemented base address registers from the configuration space. This means that the base address registers have to be directly connected to the inputs of the compare logic. For flexibility, the address compare logic is pipelined. In the event that the address from the PCI matches a base address register, a hit signal is asserted.

An asserted hit signal, or an asserted IDSEL signal for configuration transactions, will cause the control logic to claim the transaction by asserting the DEVSEL# signal. Concurrent to the address compares, the command will be decoded.

3) Handling the Data Phases

When PCI performs a write operation on the device, it is undesirable for the PCI bus to have to wait for the back-end user device to be ready to accept the data. Therefore, a FIFO-like structure is needed to reduce latency. The size of the FIFO structure should be customizable without affecting the rest of the design's logic. For this design, a single 36-bit register is used.

To handle burst cycles, an address counter for the back-end user device must be included. This counter is stepped on every data transfer. Since this design performs 32-bit transfers, addressing must be on double word aligned boundaries.

For read operations, parity must be generated and made available one clock after the data transfer takes place.

4) The Control Logic

The control logic must be able to handle the PCI protocols, support burst transfers, wait states, and still meet the 2- to 11-ns clock-to-out time, and 7-ns set-up time of PCI bus signals. It should also provide all the internal control signals and user interface signals. Because of its high-level nature, the control logic should be implemented in VHDL. To meet clock-to-out times, output should be registered.

5) The Configuration Space

Many designs will only use 00h to 0Bh configuration registers and Base Address registers. The configuration space implementation should have these registers and the mechanism for writing to and reading from those registers. In addition, the register information is used internally by direct means (as opposed to using a read operation) so the contents of the registers need to be accessible by the rest of the design. For example, the Base Address registers need to be connected directly to the inputs of the address comparator logic. Since customizing the design for real applications will involve modifications to the configuration space, the configuration space registers is implemented in VHDL.

Block Diagrams and Data Paths

Figure 8 shows the top-level block diagram of a PCI target interface developed using the criteria of the previous section.

CONTROL: (VHDL) This block contains the PCI and user state machines and the logic that determines the internal control signals as well as the bus signals.

C_SPACE: (VHDL) This block is the VHDL implementation of the configuration space. "Hardwired" values are easily set in the VHDL source code and registers are manipulated behaviorally.

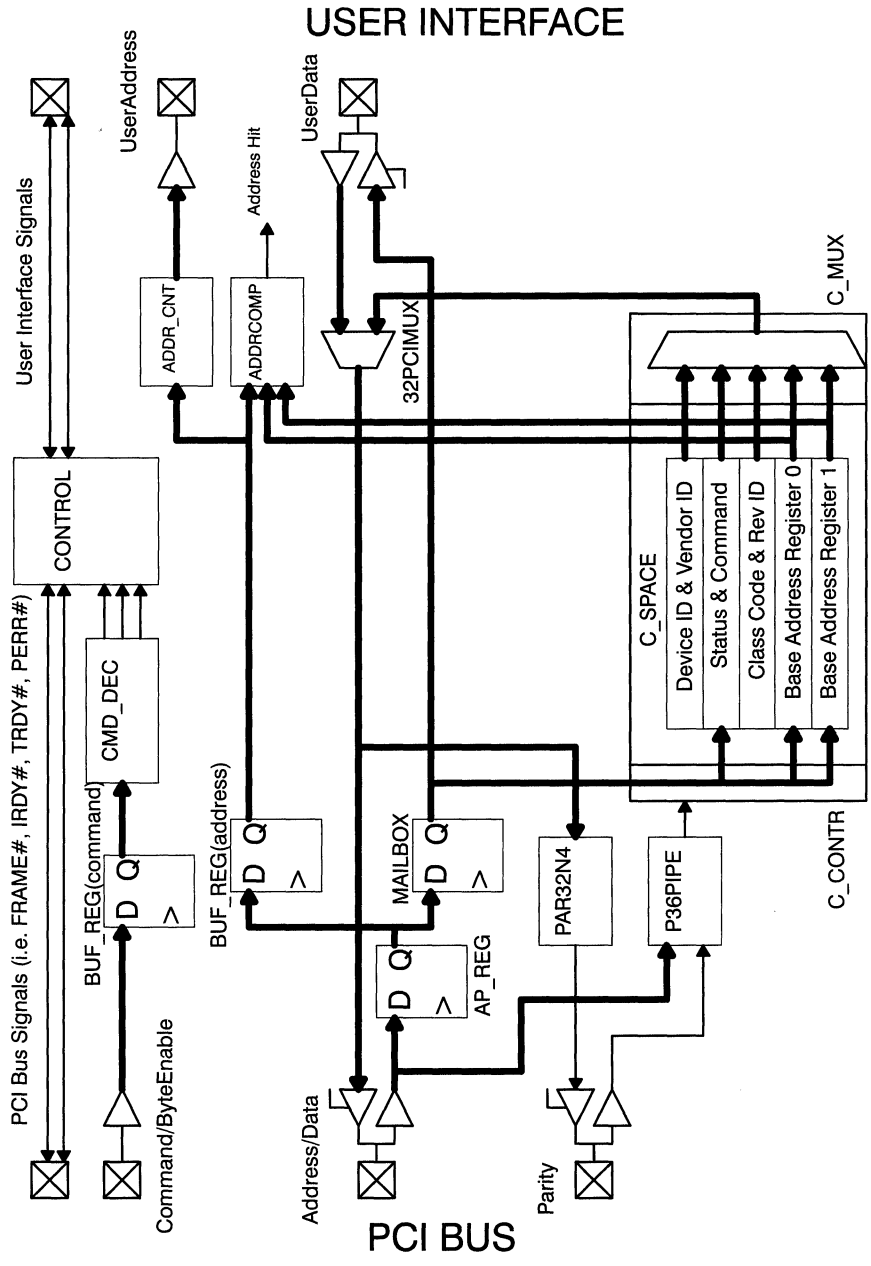


Figure 8. Target Design Block Diagram

AP_REG: (Schematic) This block acts as the input register for the A/D bus, C/BE bus, and the IDSEL signal. On every clock, the values on these signal lines are registered into this block.

BUF_REG: (Schematic) This is the storage register for the address and the command taken from the PCI bus at the beginning of each transaction. It is enabled by the CONTROL block and cleared upon reset or at the close of a transaction.

CMD_DEC: (Schematic) The command is decoded into single-bit enable lines. The decoded command, the I/O and Memory access enable, and IDSEL line determine which function signal will be raised. All unimplemented memory transactions are treated as either the respective mem read or write.

ADDRCOMP: (Schematic) This pipelined address comparator takes two cycles to determine an address hit. The number of bits compared can range from 1 to 32 bits. If less than 16 bits need to be compared, the pipelined configuration usually is not necessary since a hit can be determined within one clock cycle.

MAILBOX: (Schematic) This is a data-holding register for I/O or Memory writes. This block may be changed to a multileveled FIFO to reduce latency between burst transfers.

PAR32N4: (Schematic) This block calculates even parity over the 32-bit A/D bus and the 4-bit C/BE bus in one clock cycle. The output is registered to delay the valid parity bit one clock in accordance with the PCI spec.

ADDR_CNT: (Schematic) The initiator provides the beginning address for a transfer. On burst transfers, the target device is responsible for stepping the beginning address appropriately for its local user side. This block counts the address on data transfers.

C_CONTR: (Schematic) This block decodes the address and enables the addressed register within the configuration space.

C_MUX: (Schematic) This is a 32-bit, 4-to-1 mux, exclusively selecting configuration registers 00h, 04h, 08h, and 10h. If other configuration registers need to be addressed, a larger mux must be used. The 4-to-1 mux was chosen because it fits in a single level of logic cells. When nonimplemented registers are addressed, the block outputs zeroes.

32PCIMUX: (Schematic) This is a 32-bit, 2-to-1 mux, selecting between the local user data bus and the configuration space register output mux C_MUX.

State Machines

Within the CONTROL block, there are two state machines: the PCI state machine (*Figure 9*), which handles PCI bus protocols, and the User state machine, which handles transactions on the user interface (*Figure 10*).

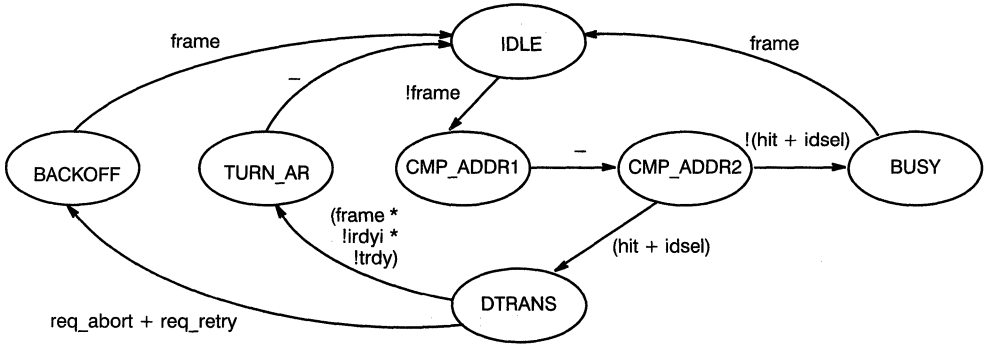


Figure 9. PCI Interface State Machine

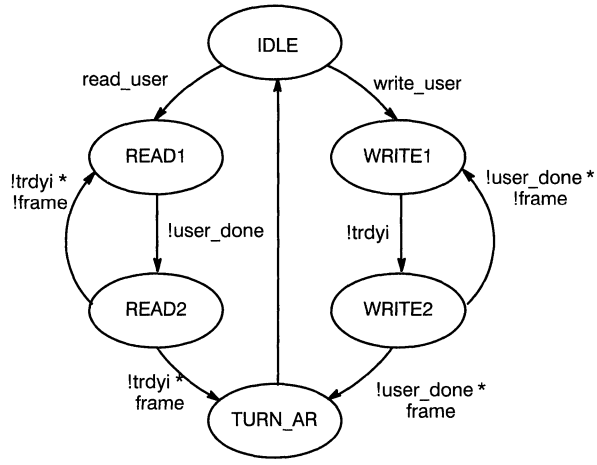


Figure 10. User Interface State Machine

PCI State Machine

IDLE: The device waits in this state while the PCI bus is idle. When the FRAME# signal indicates a transaction is beginning (becomes asserted), the FSM moves to the CMP_ADDR1 state.

CMP_ADDR1: This is the first stage of the address and command decoding pipeline. On the next clock, the FSM moves to the CMP_ADDR2 state.

CMP_ADDR2: This is the second stage of the decoding pipeline. At this point, the address hit and command are determined. If it is an address hit or a configuration operation is occurring on that device, then the FSM moves to the DTRANS state. Otherwise, it goes to the BUSY state.

BUSY: In this state, the PCI bus is engaged in a transaction that the device is not a part of. The device will wait in this state until the bus goes idle again.

DTRANS: All data transfers occur in this state. When the device determines that it is involved with the last data transfer of the transaction, the FSM will move to the TURN_AR state on the next clock. If an abort is sensed, then the FSM will move to the BACKOFF state.

TURN_AR: Signals on the PCI bus are precharged and three-stated, and the A/D bus is brought to high impedance. The FSM moves to the IDLE state on the next clock.

BACKOFF: The device induces a target abort or a retry in this state. When the transaction is closed, the FSM moves to the IDLE state.

User State Machine

IDLE: The user FSM stays in this state while the user interface is inactive. When the device is involved with either a read or a write transaction involving the user interface, the FSM moves to the READ1 or WRITE1 state appropriately.

READ1: In this state, the user interface prompts the user device for the requested piece of data. When the user device responds with valid data, the FSM moves to the READ2 state.

READ2: The device has the requested data ready, and waits for the initiator to pick it up. Once the transfer takes place, the FSM either moves to the READ1 state for burst transfers, or to the TURN_AR state.

WRITE1: In this state, the device receives the data from the initiator. Once the data transfer takes place, the FSM moves to the WRITE2 state.

WRITE2: Data is available in the data FIFO. The user device is prompted for a data write. When the

user device signals that the transfer is completed, the FSM moves back to the WRITE1 state for burst transactions, otherwise it moves to the TURN_AR state.

TURN_AR: This is the final state of the user FSM before going back to IDLE.

Design Interaction Description

To demonstrate the operation of this PCI target design, a description of the waveforms are analyzed.

Scenario 1: Configuration Write

[These scenario descriptions follow the simulation waveforms produced in ViewSim. Simulate design using command file PCICR.CMD with the PCI target design, 75 ns.]

At the beginning of the transaction (clock 0), this target device senses on the clock that FRAME# has been asserted. Because the AP_REG captures all information on the A/D and C/BE buses and the IDSEL line on every clock, the target knows that the address is held within the AP_REG.

On clock 1, the BUF_REG is enabled so that the address and command can be stored. Between this clock and the next, the IDSEL line is found to be asserted and the CONTROL logic determines that DEVSEL should be asserted. The command is also decoded to be a configuration write operation, and the bits [7:2] of the address are decoded by C_CONTR to enable the appropriate C_SPACE register.

On clock 2, the internal DEVSEL signal is captured by the DEVSEL output register to meet the 2- to 11-ns clock-to-out timing spec. The target samples IRDY# asserted, and knows that valid data is on the bus. The CONTROL block asserts the internal TRDY# signal.

On clock 3, the initiator samples the DEVSEL signal asserted and knows that the transaction has been claimed. The TRDY# output register asserts the PCI bus TRDY# signal. Valid data is contained in the AP_REG.

On clock 4, the data from the AP_REG is written to the C_SPACE register according to the byte enables also held within the AP_REG. The CONTROL logic samples FRAME# deasserted and knows that this is the last transaction. Both the TRDY# and DEVSEL# signals are deasserted.

On clock 5, the CONTROL logic three-states the bus signals, and resets the address compare and BUF_REG blocks. The transaction is complete.

Scenario 2: Configuration Read

[In ViewSim, use command file PCICR.CMD with the PCI target design, 315 ns.]

This transaction works like the Configuration Write transaction with a few differences:

On clock 1, the A/D bus is floated by the initiator to turn control of the data bus over to the target. When the CONTROL logic asserts DEVSEL, it turns on the output enable to the A/D bus.

The address held in BUF_REG causes C_MUX to select the appropriate configuration register bus. For this design, only 32-bit registers at 00h, 04h, 08h and 10h are selected. Other addresses will cause the C_MUX to randomly select one of the four buses.

The 32PCIMUX selects between configuration and IO/Memory reads. The output of 32PCIMUX goes directly to the output pins of the A/D bus.

Scenario 3: I/O or Memory Write

[In ViewSim, use command file PCIMW.CMD with the PCI target design, 560ns.]

At the beginning of the transaction (clock 0), this target device senses on the clock that FRAME# has been asserted. The AP_REG captures all information on the A/D bus every clock, therefore the target knows that the address is held within the AP_REG.

On clock 1, The BUF_REG is enabled so the address and command can be stored. While this happens, the address compare pipeline begins it's first phase. Between this clock and the next, an address hit is determined and the CONTROL logic determines that DEVSEL should be asserted. The command is also decoded at this time.

On clock 2, the DEVSEL signal is captured by the DEVSEL output register to meet the 2- to 11-ns clock-to-out timing spec. The user address counter

is loaded with the offset address from BUF_REG. The ADDR_CNT is enabled after this clock to load it with the offset address on the next clock.

On clock 3, the initiator samples the DEVSEL signal asserted and knows that the transaction has been claimed. The target waits with the TRDY# signal deasserted until an asserted IRDY# signal. The CONTROL logic senses that the IRDY# signal is asserted, and thus knows that valid data is on the bus. The CONTROL logic then prepares to assert the TRDY# signal on the next clock, and enables the MAILBOX register.

On clock 4, both IRDY# and TRDY# signals are asserted so both agents know that a data transfer took place. The enabled MAILBOX register collects the contents of the AP_REG (previous clock held valid data also since IRDY# was already asserted). The TRDY# signal is immediately deasserted. If the CONTROL logic sample FRAME# to be asserted (indicating a burst transfer), then the target would prepare to perform another data transfer. Otherwise, the CONTROL logic will end the transaction just like the Configuration Write transaction.

On clock 5, the CONTROL logic for the user interface side senses that the MAILBOX contains data to be written to the user device. The USR_WRITE strobe is asserted, and the CONTROL logic waits for the user device to respond with an asserted USER_DONE.

On clock 6, USER_DONE is sampled asserted, and the CONTROL logic 'clears' out the MAILBOX register and increments the ADDR_CNT.

Scenario 4: I/O or Memory Read

[In ViewSim, use command file PCIMR.CMD with the PCI target design, 560 ns.]

This operation works like the I/O or Memory Write with these differences:

On clock 1, the A/D bus is floated by the initiator to turn control of the data bus over to the target. When the CONTROL logic asserts DEVSEL, it turns on the output enable to the A/D bus.

After clock 2, the CONTROL logic prompts the user device with the USR_READ strobe.

On clock 3, the USER_DONE signal is sampled asserted. This is a 'pass-through' read design so the user device must hold the data values so the PCI bus can read them.

On clock 4, the CONTROL logic prepares the TRDY# signal so that a data transfer may take place on the next clock.

On clock 5, both IRDY# and TRDY# signals are asserted. If the CONTROL logic senses that FRAME# is asserted at this point (indicating a burst transfer), then the target prepares to perform another data transfer. Otherwise, it closes the transaction in the normal fashion.

PCI Target Interface Timing Specifications

Table 4. PCI Bus I/O Timing Specification

Symbol	Description	Min.	Max.
t_{val}	Clock to Data Valid	2	11
t_{on}	Float to Active Delay	2	–
t_{off}	Active to Float Delay	–	28
t_{su}	Input Set-Up Time	7	–
t_{h}	Input Hold Time	0	–
t_{cuc}	Clock Cycle Time	30	–
t_{high}	Clock High Time	12	–
t_{low}	Clock Low Time	12	–

Table 5. User Interface I/O Timing Specification

Symbol	Description	Min.	Max.
t_{val}	Clock to Data Valid	2	14
t_{su}	Input Set-Up Time	14	–

Critical PCI Design Issues

There are several considerations that arise when designing PCI applications using an FPGA. For an FPGA to be able to handle the demands of PCI, it must have several necessary characteristics. These

characteristics include: speed, generous routing, a large number of pins, a large amount of logic resources, and many registers. This section will describe some critical issues and possible solutions to implementing PCI applications with an FPGA.

(1) Bused signals set-up time is no greater than 7 ns. (PCI spec 7.6.4.2)

Problem 1: The Address/Data bus needs to be tapped by several blocks: the address decoders for each Base Address, the address registers, the data registers for memory and I/O transactions, and the data bus for the configuration space. This fanout can add considerable loading to the bus, thus inhibiting the input drivers and increasing the input delays beyond the 7-ns set-up time, even with the FPGA's short input delay.

Solution 1: A 36-bit input register can be implemented using D-type flip-flops. These flip-flops will latch whatever is on the Address/Data bus every clock. This increases the availability of the data from 7 ns to 30 ns, with the trade off of adding one clock cycle. This additional clock cycle, however, does not significantly impact the performance of the device for several reasons: during the address phase, addresses must be latched anyway; and during data phases, another PCI spec forces the extra wait state.

Problem 2: Some bused signals such as IRDY# and FRAME# are used combinatorially to determine other outputs. In some cases, the combinatorial delay to the registered outputs and states of the state machine take longer than 7 ns, thus giving an invalid registered output or state.

Solution 2: Remember that there is a clock input delay to the device. The actual set-up time is input delay minus the clock delay. In the event that this difference is still greater than 7 ns, then care should be taken to minimize fanout of the input signal, and to place the logic near the pin. In most cases, this is taken care of automatically by the place and route tool by placing a constraint on the signal. To do this, run SpDE and open up the design .CHP file. Run the path analyzer. Click on options, and display all paths that start from the critical input signals (i.e., IRDY# and FRAME#). Place constraints on the critical signals paths (e.g., type "5.0" ns in the constraint column for all critical paths) and rerun placer tools.

2) Bused signals must be driven valid between 2 and 11 ns after CLK. (PCI spec 7.6.4.2)

Problem: Many delays contribute to a signal's total delay. These delays include: the clock input to flip-flop delay, the clock to Q output delay, the combinatorial delay, all routing delays, and the final signal to output pin delay. Particularly for programmable logic, these delays are on the order of nanoseconds (as opposed to picoseconds, as is the case in ASICs). The total clock to output delay quickly passes the 11-ns spec.

Solution: The quickest, easiest, and most robust solution is to register the outputs. However, this solution has the trade off of adding one additional clock cycle. For long delay calculations, pipelining may be used to reduce variables. By doing the necessary calculations in a previous stage, the final stage can have a shorter total delay. Because PCI has wait states, the pipelining solution may be used.

3) All inputs require no more than 0 ns of hold time after CLK. (PCI spec 7.6.4.2)

Problem: Devices must have to be able to latch data with a 0-ns hold time.

Solution: It is necessary to use a part that can meet the 0-ns hold time spec. The Cypress 38x FPGA family meets the 0-ns hold time.

4) Configuration Space of PCI requires many registers.

Problem: PCI specifies that 256 bytes of register space be implemented.

Solution: Most of the 256 bytes of registers can be implemented as hard-wired zeroes. This reduces the need to use flip-flop resources to implement the configuration registers. In addition, some of the bits within the 32-bit registers may also be hard-wired to some permanent value. As a minimum, the configuration space will probably require a minimum of approximately 40 flip-flop registers for the simplest design.

5) Multiplexed Address/Data bus is routed to several places within the device.

Problem: PCI has a multiplexed address/data bus. The bus is accessed internally by several devices such as registers, FIFO, parity check/generators, comparators, and decoders. This requires the use of several 32-bit muxes.

Solution: Each logic cell of the 38x FPGAs has a cascaded muxing structure that can implement a 4-to-1 mux. By grouping signals into fours (along with their control signals), more optimal performance and utilization can be achieved.

6) PCI device must respond to a transaction within 3 clocks after the Address Phase.

Problem: After the first clock that the FRAME# signal has been asserted by the initiator, the addressed target must respond by asserting the DEVSEL# signal. If a target can respond within one clock, it is considered to be a "fast" response device. If it responds in two clocks it is a "medium" response device, and if it responds in three clocks it is "slow." The fourth clock after the asserted FRAME# signal is reserved for subtractive decoding devices such as bridges. If the initiator is not responded to within four clocks, it will abort the transaction. Therefore, most PCI devices must respond with the DEVSEL# signal within three clocks.

All targets have one clock (the first time FRAME# is sampled LOW) to latch the address and command from the PCI bus. They must immediately begin decoding the address to determine the recipient of the transaction. This is done by comparing the address to all implemented base address registers within the configuration space. If a hit is determined, then the target must assert its DEVSEL# signal. Parity (if enabled) is also checked on the second clock to determine if a parity error occurred during the address phase.

Solution: Pipelining the address compare function will allow the design to meet the timing. Remember that there is only a 7-ns set-up time on the bus, and therefore the first stage of the pipeline must be able to complete within this time (plus accounting for internal clock delay). Registering the DEVSEL# signal will insure the 2- to 11-ns clock-to-out time, but keep in mind that this will add an extra clock to your response time.

7) Parity

Problem: Even parity over the 32-bit A/D bus, 4-bit C/BE bus, and parity signal must be calculated and made available exactly one clock after a valid data transfer. Implementing the parity generator requires several levels of XOR logic. It should also have a small propagation delay to prevent excessive wait states during data transfers.

Solution: A single logic cell in the pASIC family can implement a 3-input XOR. Building the parity generation logic with 3-input XOR yielded a parity generator which utilized a minimal amount of logic cells and routing. The parity generator induced no extra necessary wait states.

8) High Fanout Signals

Problem: Several combinatorially produced signals have a very high fanout. For example, a signal will be used to enable 36 registers at once for data and byte enable latching. Signals with high fanout incur long propagation delays.

Solution: Inherent to all FPGAs, signal delays are often routing dependent. Reducing the number of loads on a signal, thus reducing the number of routing resources, can greatly improve performance. There are several methods for doing this: split buffering, selective buffering, paralleling, and double buffering. For more information on buffering techniques, see Chapter 4 "Design Techniques" of the *Warp3*™ User's Guide, SpDE/Warp System section.

Split buffering involves inserting another layer of logic between the signal source and all of its loads. For example, if a signal has 10 loads, the loads can be split into two groups of five. Each group would then be driven by a BUF component, which in turn are driven by the signal source. This reduces the load of the original signal to just two.

Selective buffering is similar to split buffering: an extra buffering layer is inserted. The difference between the two methods is that a few of the original load signals are more timing-critical than the others. In this case, those critical signals should be driven by the original source (the same level as the buffers).

This effectively reduces the load of the original signal, without adding extra logic between the source and the timing-critical signal.

Paralleling has the advantage of no extra layers of logic with the trade off being a complication of the design. This method involves repeating the signal's source logic. For example, if the signal is produced by an AND gate, this AND gate would be repeated (both with the same input values) and each gate would then drive its own group of logic.

Signals with larger fanouts or speed-critical signals should be buffered using the DOUBLE_BUFFER attribute in their design. Keep in mind that every time this technique is used, express wires in the device are used. Using this attribute without discretion can quickly exhaust all available express wires within the device. A second improvement to double buffering is to place the flip-flops in a single column. This has the advantage of shorter signal paths and uses less express wires. To place flip-flops, use the FIXED_FF attribute on the registered signals.

Making Modifications to the Design

Assigning Values to Configuration Registers

Configuration registers DEVICE ID, VENDOR ID, CLASSCODE, and REV ID must be assigned. To do so, edit the configuration space block: C_SPACE.VHD (VHDL). These registers are declared as constants and their assignments may be changed to the appropriate values.

Changing Address Space Size

Different applications will have different address space size demands. Modifications of the design to match size demands is expected. Since a device's address space is determined by the number of hardwired zeroes in the lower bit positions, decreasing the address space size increases the number of bits compared to determine an address hit. Likewise, if the address space size in increased, the number of bits compared goes up. To customize this design to meet an application's address space size demands:

1. Edit the configuration space VHDL.
   - Modify the signal declaration of BASE_ADDR_X to be the appropriate size bit vector.
   - Locate where the base address is assigned a value from the data bus and modify the BASE_ADDR_X and PCI_DATA vector sizes to the appropriate size.
   - Locate where the base address values are sent to the output pins of the configuration space block and modify the BASE_ADDR_X vector and number of concatenated zeroes to the appropriate size.
2. Modify the address compare logic
   - The schematic of the decode logic (xxP_DEC) is a nibble oriented design. The number of bits compared in this circuit should reflect the number of bits necessary to determine an address hit.
3. Modify the user address counter
   - The burst length of a device does not always reflect the size of the address space. In this design the user addressing counter allows double word burst lengths of 16. The size of the counter may be modified to meet the required burst length.
   - Regardless of the size of the burst length counter, all lower bit positions must be sent as output to the user address pins to cover all locations in the allocated address space.

Target Aborts and Retries

The control logic of this design is ready to handle target aborts and retries. However, as the design stands, no logic uses this functionality. (Notice REQ_ABORT and REQ_RETRY inputs to the CONTROL block are grounded.) Logic for determining target aborts or retries may be added, and used to signal the CONTROL block to perform the target abort/retry.

Increasing the Depth of the FIFO

The control logic of this design utilizes a 36-bit register for write operations. PCI interface side logic performs a data transfer when it sees that the regis-

ter is not full. The user interface side logic performs a data transfer when it sees that the register is full. Minor modifications to this logic should be done to support an internal FIFO.

Conclusion

Interfacing with the PCI bus is a task of intricate protocols, timing specs, and data handling. However, the PCI challenge can be met by using PLDs, and in particular, FPGAs. The flexibility, high density, and compliance of Cypress FPGAs make the FPGAs ideal candidates for PCI bus interface applications such as add-in cards.

PCI read and write transactions are inherently non-preempted burst transactions. The basic protocols are the same for configuration, I/O and Memory read and writes. All PCI bus devices implement configuration space registers which give PCI its plug-and-play nature.

Because of the many issues involved in PCI interfacing, a designer will inevitably run into a multitude of challenges. Careful planning and use of this application note and reference design can provide a head start in the design process.

CY7C380 Family Quick Power Calculator

This brief is intended to provide a rapid method of calculating the approximate power consumed by a CY7C380 family device. Because the intent is a first-estimate calculation, some details are neglected. The quiescent power of about 20 mW is not included. There is no estimate of the power for the number of columns and number of loads per column of the clock distribution tree. Wiring capacitance is neglected. High drive cell power is taken to be the same as a normal input cell. I/O cell power is averaged over input and output. The power calculation does not include the power of an output driving an external load. This approach was taken to simplify the calculation. The average toggle rate and per cent of the device used are assumed to be rough estimates, thus there is no need to strive for great accuracy. For detailed considerations refer to the application note "Power Characteristics of Cypress Products."

The equations used to create the curves are:

$P(I/O) = (\text{number of I/Os})*Fav*(0.3)$

$P(\text{cells}) = (\%\ \text{used})*F_{AV}*(0.38)$ for 7C381/C382 (*Figure 1*)

$P(\text{cells}) = (\%\ \text{used})*F_{AV}*(0.77)$ for 7C383/C384 (*Figure 2*)

$P(\text{cells}) = (\%\ \text{used})*F_{AV}*(1.54)$ for 7C385/C386 (*Figure 3*)

Where F_{AV} is the average toggle rate frequency

Quick Power Calculation Process

1. Estimate the toggle rate (frequency in MHz) for each of the major blocks of the design.
2. Select a CY7C380 family device.
3. Estimate the percent of the device that will be utilized to implement each block.
4. For each block, use the power vs. toggle rate curves for the selected device and read the power for the estimated toggle rate and percent utilization. Enter the power in the work sheet.
5. Sum the individual powers for an estimate of the total power.

Table 1. Power Calculations

Block	Percent of Device	Toggle Rate (MHz)	No. of I/Os switching at toggle rate	$Power_{I/O}$ (from eqn)	$Power_{cells}$ (from table)	$Power_{Block}$ ($Power_{I/O}$+ $Power_{cells}$)
Block 1						
Block 2						
Block 3						
Block 4						
	100%					

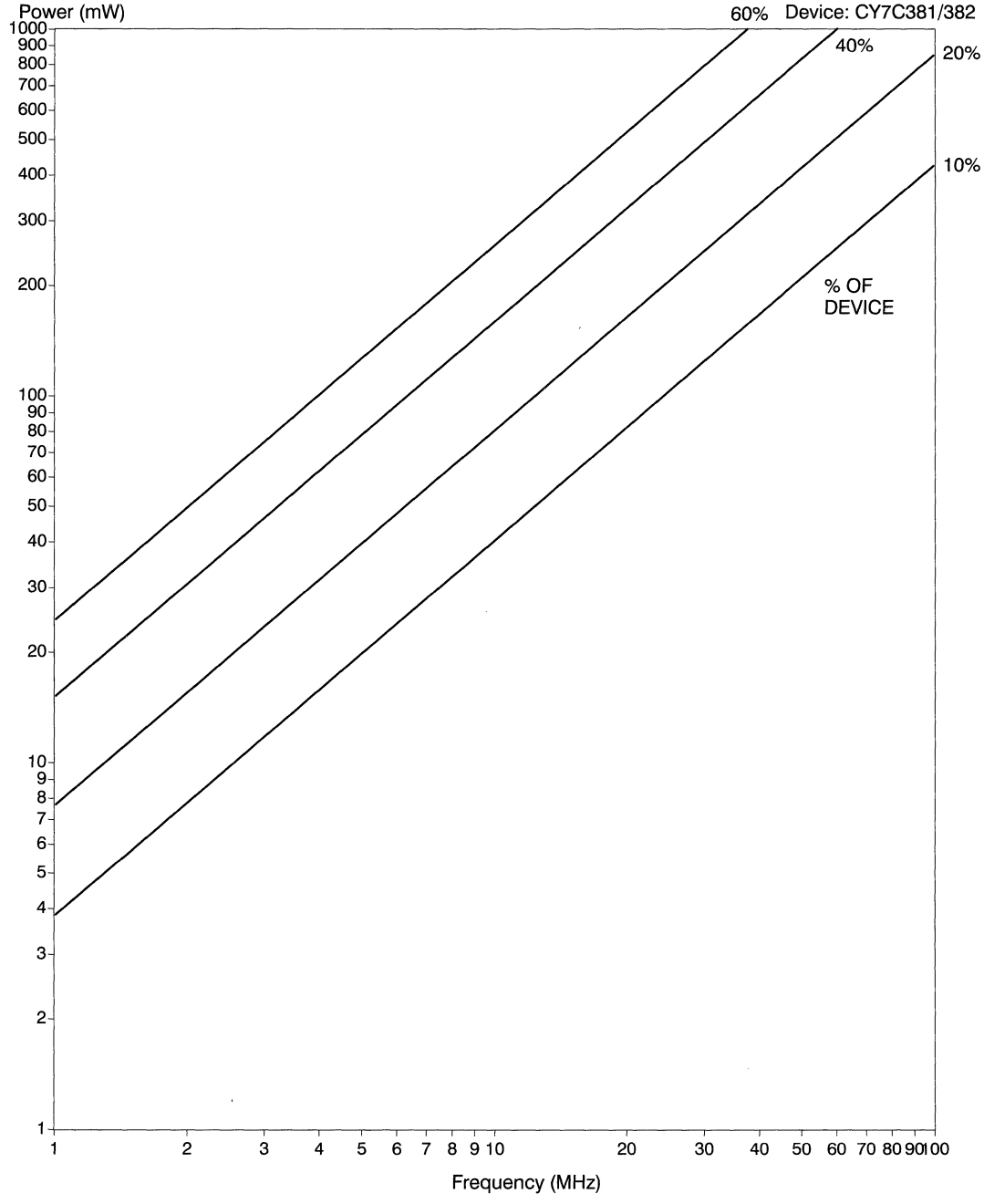


Figure 1. Average Toggle Range for CY7C381/2

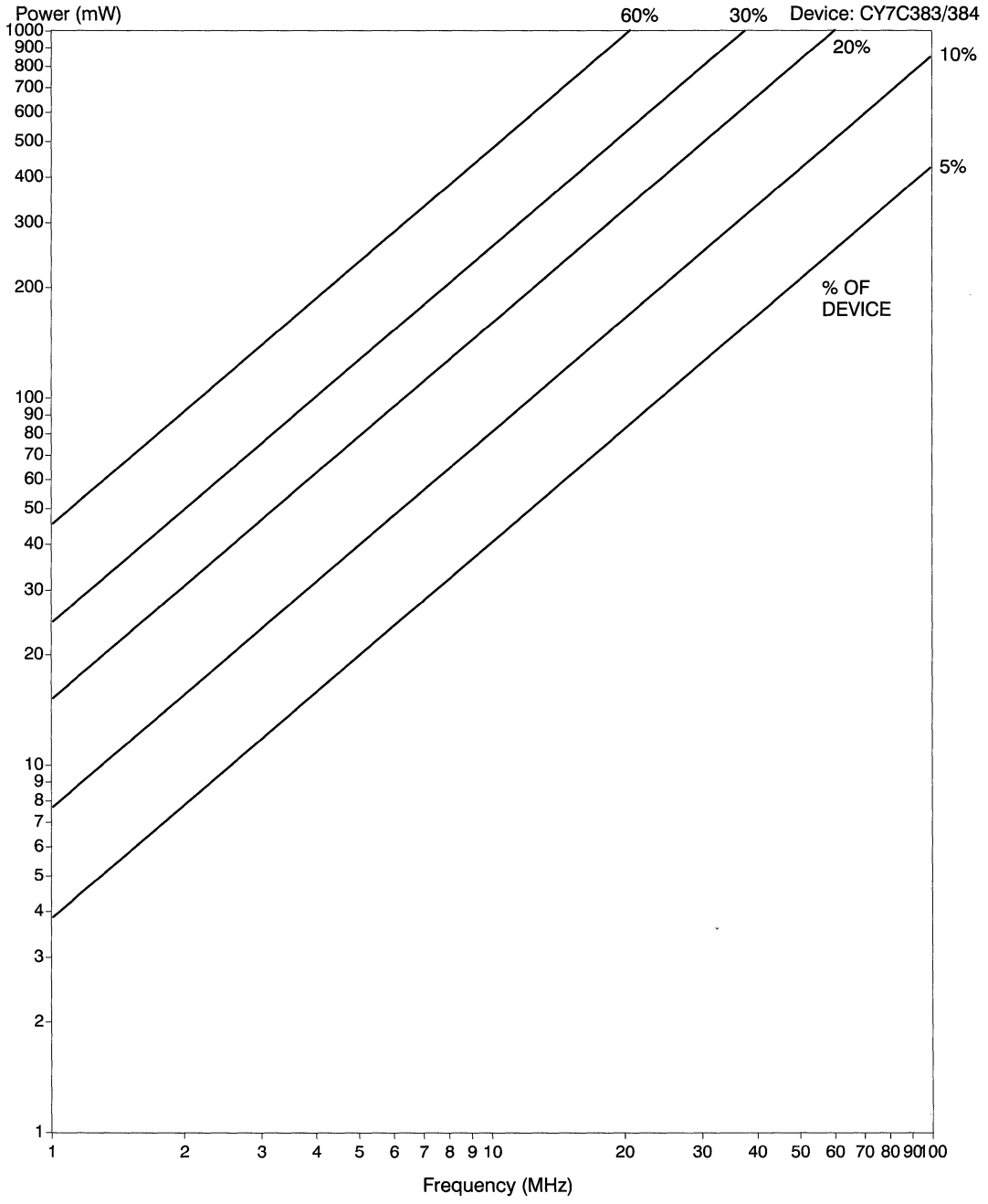


Figure 2. Average Toggle Range for CY7C383/4

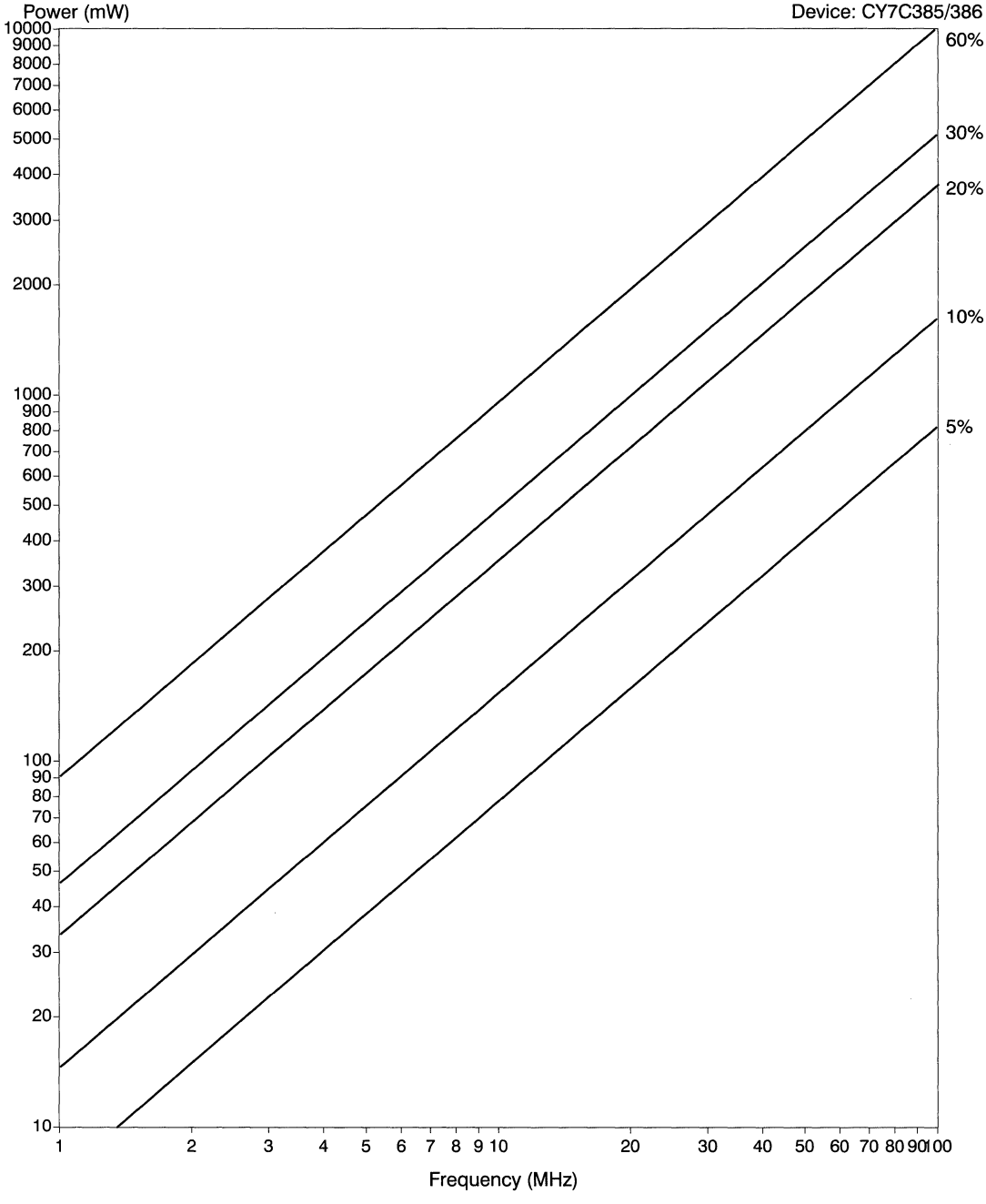


Figure 3. Average Toggle Range for CY7C385/6

Example

A CY7C382 FPGA is to be used with the following estimates:

- 32 I/Os are connected to a 40-MHz bus (half are assumed to be changing at this rate on the average).
- The remaining I/Os have a low duty cycle.
- 10% of the device is going to be toggling at the 40-MHz rate.
- 60% of the device is estimated to be toggling at 10 MHz.

The work sheet is filled in as shown below. The number of I/Os is taken to be 16 because half are assumed to change in any clock (on the average). The next two entries are taken from the graph for the 7C382 and entered into the Power column. Total power is summed at the bottom.

Table 2. Power Calculations – An Example

Block	Percent of Device	Toggle Rate (MHz)	No. of I/Os switching at toggle 191502rate	$Power_{I/O}$ (from eqn)	$Power_{cells}$ (from curve)	$Power_{Block}$ ($Power_{I/O}$ + $Power_{cells}$)
Block 1	10	40	16	192	150	342
Block 2	60	10	0	0	220	220
Block 3*	30	0	0	0	0	0
Block 4						
	100%					562

* Block 3 represents 30% of the device that goes unused.

FPGA Design Entry Using *Warp3* ™

This application note is intended to demonstrate hierarchical as well as mixed-mode design entry for FPGAs using the *Warp3* ™ software package. *Warp3* eases and speeds up the design process by featuring both schematic and VHDL design entry methods. Complex designs may be broken up into manageable pieces and each piece may be described behaviorally (VHDL) or structurally (VHDL and schematic). All the lower-level blocks are then put together to create the top level. In this application note, a general-purpose DMA controller is designed to further familiarize the reader with the *Warp3* design process.

Warp3 Interface and the Cockpit Overview

Running on both IBM PC/AT™-compatible platform and Sun SPARCstation™, *Warp3* provides all the tools necessary to quickly and efficiently convert complex designs into functional silicon. *Warp3* uses ViewLogic as its front-end. *Figure 1* shows the Powerview cockpit which appears when you invoke *Warp3* on Unix workstations. To the upper right is a collection of icons, one for each *Warp3* tool.

Viewdraw is used to create schematics, as well as symbols that can be instantiated on other schematics. It gives you the ability to capture schematics utilizing standard 74XXX TTL functions, generic logic gates, or user-defined custom functions.

VHDL designs may be entered using ViewText or any other text editor. VHDL can be used to describe the entire design or just a portion of it. It allows for state machine, Boolean equation, IF/ELSE type constructs, tabular, and many other design description styles. Packages allow designs to be integrated into higher levels of the hierarchy.

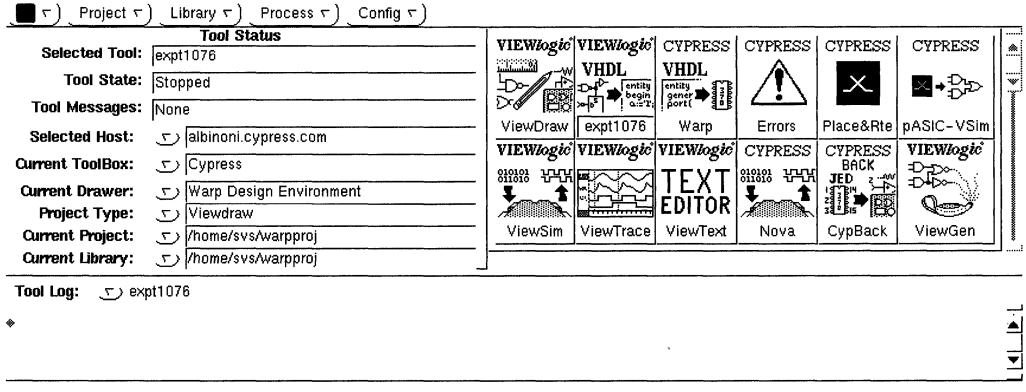


Figure 1. Powerview CockPit

ViewGen generates schematic symbols from schematic drawing. The resulting symbol could then be instantiated on other, higher-level schematics.

All designs (schematic and VHDL) are converted to VHDL, so for designs containing schematics, Expt1076 is run to translate the viewdraw schematic into one or more VHDL models.

The VHDL files are then compiled and synthesized using *Warp*™. *Warp* produces JEDEC files (used to program PLDs), HEX files (used to program PROMs), or QDIF files (used by the Place&Route tools when targeting pASIC™ FPGAs).

For FPGA devices, the Place&Rte tool is used to perform automatic place and route, delay modeling, critical-path timing analysis, automatic test vector generation, and device programming and test.

After compiling the design, ViewSim can be used to determine the design's functionality and worst case timing characteristics. ViewSim automatically brings up ViewTrace, which allows you to view the simulated waveforms.

After compilation, if the same pin assignment is desired to be kept, CypBack can be used for back annotation.

This section was intended to provide an overview of the Cockpit. For additional information please refer to the *Warp3* documentation.

Cypress pASIC380 Family FPGA Architectures

The previous architecture discussions have pointed out the strong relationship between the technology, the architecture of the FPGA, and the device characteristics. The 380 family possesses a unique technology which impacts all of the remaining architecture trade offs positively. The discussion of the 380 family begins, therefore, with a presentation of the interconnect technology.

pASIC380 Family Fuse Technology

In usual integrated circuits two crossing metal lines that are on different layers may be connected by a via. A via is a small hole in the insulating glass that lies between the two layers of metal. This small hole, which is about the size of the metal lines themselves, is filled with metal from above making the connection to the underlying metal line. The programmable via is a modified via used in standard CMOS semiconductor processing. The modification consists of depositing a thin layer of amorphous silicon in the via hole so that the silicon separates the two layers of metal. As manufactured, this special via has a resistance in excess of 1 gigaohm and an insignificantly small capacitance (about 1 fF). Its size is no larger than the standard via normally used to connect two layers of metal. A cross section of the programmable via is shown in *Figure 2*. A programming pulse applied across the programmable via causes a change in the characteristics of the silicon layer forming a bidirectional conductive link between the top and bottom metal. This programmed

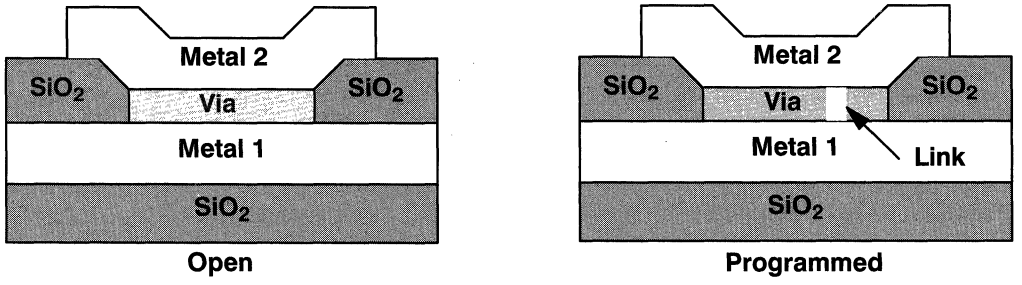


Figure 2. The ViaLink

link has a series resistance of about 52 ohms and in practice is no more than 65 ohms. The parasitic capacitance is no larger than a normal metal to metal via. The technology is appropriately termed "ViaLink™."

Routing

ViaLink technology has significant impact on FPGA architecture. Since the programmable site is no larger than the associated metal interconnect wires, there is no real restriction on the number of interconnect points (fuses) and no fuse related restrictions on the number of wires in the interconnect channels. The pASIC380 family takes advantage of this freedom with a generous routing structure.

Four types of signal wires are employed in the routing channels:

- segmented wires
- quad segmented wires
- express wires
- clock wires

Segmented wires are wires that extend only from one routing channel to the next, both vertically and horizontally. At the channel junction, a horizontal segmented wire may be programmed to interconnect to a vertical segmented wire at points called cross links. In *Figure 3*, programmable cross links are denoted by the open circle at intersections of vertical and horizontal wires. Also at the channel juncture, the segmented wire may be continued in the original horizontal or vertical direction by connection to another segmented wire running in the same channel. This connection is provided by a pass link. These links are denoted by an "x" in the figure. Segmented wires are most applicable for local wiring around or between adjacent logic cells.

Quad segmented wires are similar to the segmented wires described above except that the wire extends across four logic cells before it is segmented. Like segmented wires, the quad segmented wires may be continued to the next quad segmented wire by a pass link. The quad segmented wires are applicable to signal distribution over a larger but still local group of logic cells.

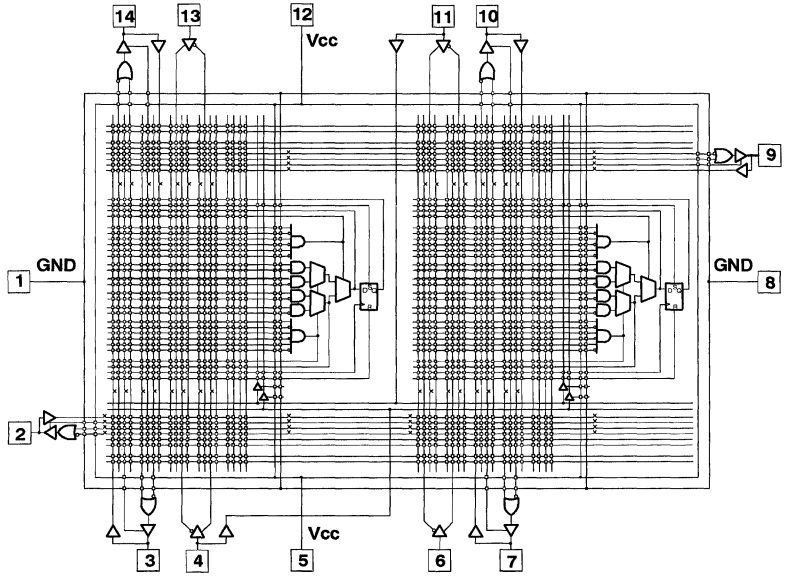


Figure 3. Simplified pASIC380 Family Model

Express wires are similar to segmented wires except they do not include pass links. An express wire will therefore run the entire length of the device. These wires are most suitable for global signals within the device.

Routing software with specific knowledge of the device architecture will automatically route signals over the appropriate wire type.

Clock wires are special signal lines that include an array of buffers for minimal skew. Clock wires are similar to express wires except that the cross links are limited. This is to insure that the clock wires are lightly loaded by programmable interconnects and can be used maximally in routing high-speed clocks or reset signals globally throughout the device with minimal skew. The source of the signal on the clock wires is specific device pins with the designation "I/CLK." After passing through the special input buffers, the signal is routed horizontally across the center of the die, as shown in *Figure 4*. There are four high drive buffers. One pair drive clock 1 and clock 2 to the upper half of the column of logic cells, and the other pair drive the two clocks to the lower half column of logic cells. There is a cluster of these buffers for each column of logic cells in the array. The buffers can be enabled to drive the clock lines or disabled if a clock is not required in a given column.

Vertical channels include all three wire types plus V_{CC} and ground wires. The V_{CC} and ground connections allow unused inputs of any logic cell to be tied to an appropriate logic level. The vertical channels run to the left of each logic cell column and extend the full height of the device. The I/O wires, which run from each of the logic cells to the right of the vertical channel, intersect the wires of the vertical channel with cross links at all segmented wires and at

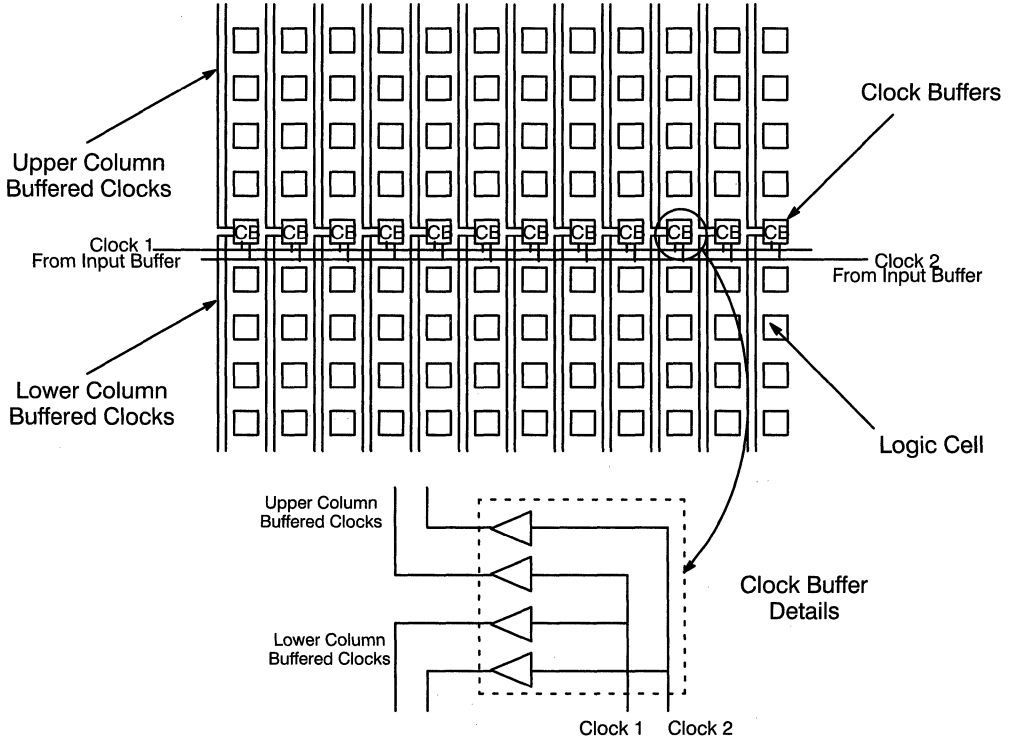


Figure 4. pASIC380 Family Clock Distribution

judicious points for express wires. At the extreme ends of the vertical channels are I/O cells that connect to the device pins. The number of wires in the vertical channel is chosen to be commensurate with the number of inputs and outputs of a logic cell, the added wires for V_{CC}, ground, and the I/O cells at the device periphery. There are 24 of these wires.

Horizontal channels provide connection by way of cross links from vertical channel to vertical channel and from the vertical channels to I/O cells on the left and right periphery of the device. All wire types are included in the horizontal channels (which contain 12 wires each) except for the clock wires. (These are the dedicated wires that carry the clocks to the buffers.)

I/O Cells

There are three types of interface buffers that connect the internal array to the device pins. The dedicated input buffer provides high drive internally and generates both true and complementary versions of the input signal. This high drive capability allows signals coming from these input only buffers to fan out to a larger number of cells than the normal I/O cell. The clock input buffer is similar to the dedicated input buffer except that it provides a third output that is routed to the internal clock distribution buffers described previously. The I/O cell provides a bidirectional connection to the devices pins. The cell can be used as input only, output only, or a bidirectional pin connection. Internally the cell has an output enable, an input data connection, and two output data connections which are ORed together to produce the output. This cell is shown schematically in *Figure 5*. The output driver provides 8 mA drive level (I_{OH} and I_{OL}).

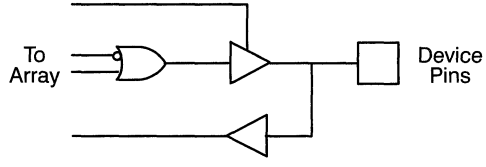


Figure 5. Bidirectional I/O Buffer

Logic Cells in the pASIC380 Family

Since the routing resources of the 380 family are abundant and without expectation of being interconnect constrained, there is freedom in the logic cell architecture to choose the optimum complexity. The 380 family logic cell is shown in *Figure 6*. This cell has been optimized to maintain the speed advantage of the ViaLink technology while insuring maximum logic flexibility.

The logic cell consists of two 6-input AND gates, four 2-input AND gates, three 2-to-1 multiplexers and a D flip-flop. This cell represents approximately 30 gate equivalents of logic capability. The cell has 23 logic and control inputs and 5 outputs. The arrangement of the gates permits 14-bit-wide gating functions and can realize all possible Boolean transfer functions of up to three variables. The D flip-flop possesses asynchronous set and reset inputs to independently control the output state. The multiplexer and logic feeding the D input allow the flip-flop to be configured as D, T, JK, or SR.

The outputs of the logic cell include the Q output of the flip-flop (QZ) plus four other outputs tapped at selected points within the logic cell. The OZ output is the same as the D input to the flip-flop. The OZ

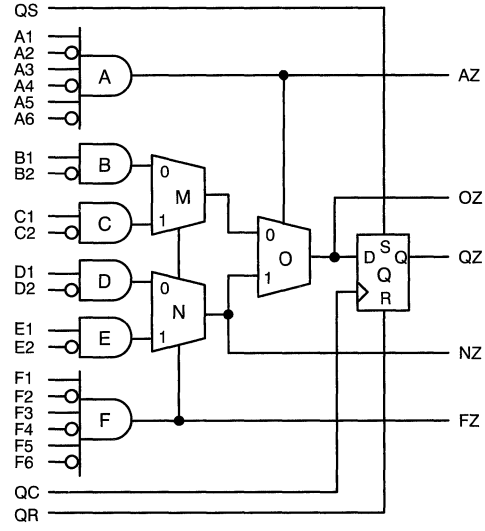


Figure 6. pASIC380 Internal Logic Cell

output facilitates combinatorial functions. The three other combinatorial outputs tap the logic cell at selected places. If simple logic functions are to be implemented, the multiple outputs permit more than one of these functions to be realized in a single logic cell. Maximum use of the available logic can be made. Note the ability to provide this multifunction utilization without any significant impact on routing. The additional utilization factor is obtained for free. When implementing multiple functions, the flip-flop may still be employed in many cases.

The logic cell is not so complex as to adversely impact propagation delay. The internal multiplexers are positioned to participate in implementing logic functions. Since the multiplexers are all in the path to the D input of the flip-flop, they contribute significantly to combinatorial logic function realization and are not expended on signal steering. The logic cell is also noticeably symmetric and regular. Combinatorial delays are thus also symmetric. That is, input to output delays tend to be roughly the same, although the AZ and FZ output will be faster than the others. Whereas some architectures bypass large sections of cell logic by the multiplexing, thereby making the cell delay dynamically changeable, the pASIC380 logic cell delay is not subject to this condition.

Design Example

The application example described here is a general-purpose, 16-bit direct memory access controller (DMAC). Direct Memory Access facilitates maximum I/O data rate and maximum concurrence. For DMA transfers, the Central Processing Unit (CPU) must have a DMA feature. Additional external logic is also necessary. This additional logic, the DMA controller, contains its own address register, word count register, and logic for reading or writing data to or from memory. *Figure 7* illustrates the basic components of a DMA controller.

The CPU loads the DMAC with a starting address for the memory transfer and the number of words to transfer. When an I/O device requires data from memory or needs to transfer data to memory, it must request service from the DMAC by asserting a DMA request (DREQ). The DMAC then activates its hold request (HLDREQ) output. The DMAC then waits until it receives a hold acknowledge (HLDA) signal from the CPU. At this time the CPU floats its address and data buses and appropriate control lines. It suspends any processing that re-

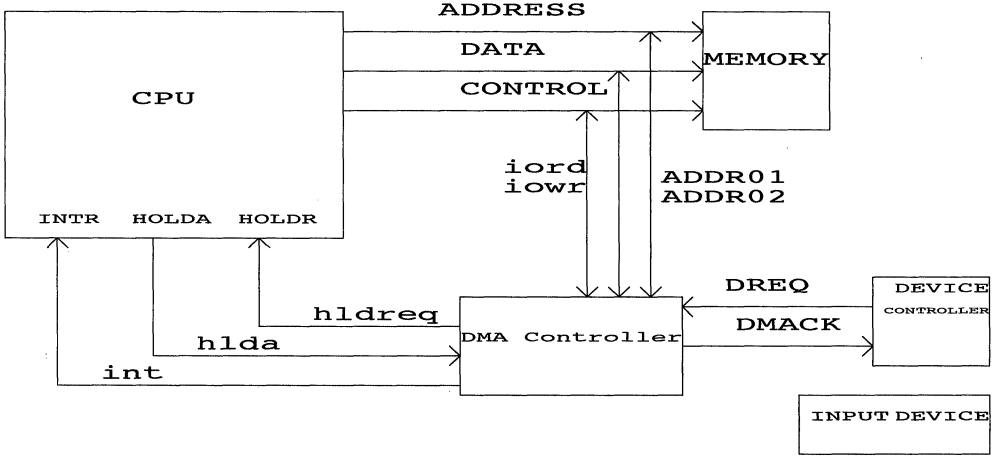


Figure 7. DMA Controller Controlling an Input Device

quires use of the address and data bus. The DMA controller then provides address and control strobes to read or write memory. The I/O device provides or accepts the data on the data bus.

Data transfers between memory and I/O devices can occur as single-word operations or as bursts of words under CPU program control. A 16-bit counter is decremented every transfer. When the required number of words have been transferred (a count of zero is reached), the DMAC terminates the DMA request and interrupts the CPU to indicate that the DMA transfer is complete.

In this implementation of the DMA controller, it is partitioned into six smaller blocks, as follows: the CPU Decoder, Control register, address generator, word counter, output multiplexer, and the DMA state machine. *Figure 8* shows the DMAC block diagram. Since some blocks are easier to describe in schematic and some others in VHDL, mixed mode design entry is selected here. For example state machine or the CPU decoder modules are easier to describe in VHDL using Behavioral and Tabular design entry methods.

The DMAC building blocks are described here in detail including an explanation of the design methodology chosen for each implementation.

CPU Decoder

The DMAC is configured by the CPU via address bits (A01 and A02), and control signals IOWRI, and IORDI. The CPU decoder receives these interface signals from the CPU and decodes them into internal write strobes and a read enable. The write strobes latch incoming parameters form the data bus into Control register, Word Counter, and Address registers. The read enable (RD_ENABL) sig-

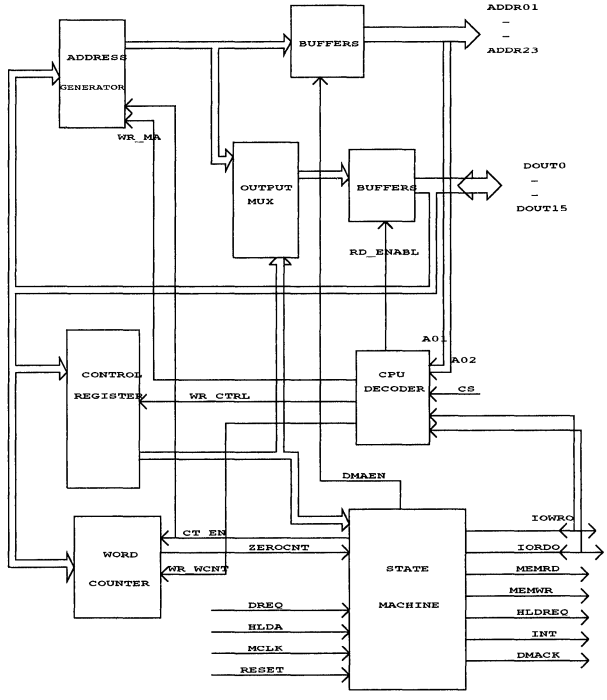


Figure 8. DMAC Block Diagram

nal and address line A01 (from the CPU) allow internally selected address registers to be multiplexed onto the data bus (during a CPU read operation). *Table 1* shows the decoding of the CPU address lines and I/O instructions by the DMAC.

Figure 9 shows the VHDL code for the CPU Decoder Block. The Entity section declares the design's inputs, outputs, and their types. VHDL provides several ways to specify a design's operation, a truth table is used to describe the CPU Decoder to express which outputs are active when specific inputs are asserted. In order to use the Tabular method of behavioral description, the "use work.table_bv.all" statement must be included. The body of architecture "arctbl" of entity "cpudec" contains the truth table. Signal TABLE_OUT is defined to hold the truth table's output signals values. Since there are 5 outputs, this signal is defined as BIT_VECTOR(0 to 4). The table is defined as constant "dectable," indicating the number of rows (0 to 4=5) and columns (0 to 8=9) it contains, followed by the bit values of the table itself.

The process "machine" then calls the TTF() function to produce outputs from the design's inputs. Since the CPUDEC.VHD (file name) is a lower-level piece of our DMA controller design, and it needs to be instantiated into our top-level DMA controller, it needs to be put in a Package. This is easily accomplished by copying and then slightly modifying the Entity section. The Package section is then placed at the top of CPUDEC.VHD file and is then recompiled. The last step is to run VHDL−> SYM (found in Viewdraw) which analyses our VHDL model and automatically generates a symbol. The symbol and the VHDL design file have the same name as the VHDL Entity name with an extension of ".1".

Control Register

The Control register configures the DMAC and controls the DMA controller's operation. The CPU writes to the control register block. The Control register has control bits to enable or disable the DMAC, enable an interrupt when the word count equals zero, clear the word counter, enable burst or single-byte transfers, and define the transfer direction (memory to I/O or I/O to memory). The bit definitions for each DMAC function appear in *Table 2*.

Warp3's schematic capture capability is used to implement the control register block. The registers can be cleared using the RESET or CLRENB signal from the state machine. The write control signal (WR_CTRL) from the CPU Decoder block clocks in the data bit values. After the design is entered in ViewDraw, Export1076 is run to convert the schematic to its VHDL model. The VHDL model is then compiled using the *Warp* compiler (Galaxy). Finally Viewgen is used to create a symbol for this lower-level design. The Control register schematic is shown in *Figure 10*.

Table 1. DMAC CPU Signals Decoding

A02	A01	CS	IORDI	IOWRI	Description
X	X	0	X	X	
0	0	1	0	1	Write Control Register
0	1	1	0	1	Write Word Count
1	0	1	0	1	Write Low Mem Address
X	0	1	1	0	Read Low Mem Address
1	1	1	0	1	Write High Mem Address
X	1	1	1	0	Read High Mem Address and DMAC Status

```
package cpu_dec is

component cpudec
  port(iowri,iordi,cs,a01,a02:in bit;
       wr_ctrl,wr_wcnt,wr_ma_0,wr_ma_1,rd_enabl:out bit);
end component;

end cpu_dec;

entity cpudec is
  port(iowri,iordi,cs,a01,a02:in bit;
      wr_ctrl,wr_wcnt,wr_ma_0,wr_ma_1,rd_enabl: out bit);

end cpudec;

use work.table_bv.all;

architecture arctbl of cpudec is

  signal table_out :bit_vector(0 to 4);
  constant dectable:x01_table(0 to 4, 0 to 8):= (
  -- inputs    outputs
  --
     "xx10" & "00001",  -- read status reg or i/o
     "0001" & "10000",  -- write control register
     "0101" & "01000",  -- write word count
     "1001" & "00100",  -- write low mem address
     "1101" & "00010"); -- write high mem address

begin

 machine: process (cs)
  begin
  if cs = '1' then
    table_out <= ttf(dectable,a02&a01& iordi& iowri);
  end if;
  end process;

wr_ctrl <= table_out(0);
wr_wcnt <= table_out(1);
wr_ma_0 <= table_out(2);
wr_ma_1 <= table_out(3);
rd_enabl <= table_out(4);

end arctbl;
```

Figure 9. CPU Decoder VHDL Design File

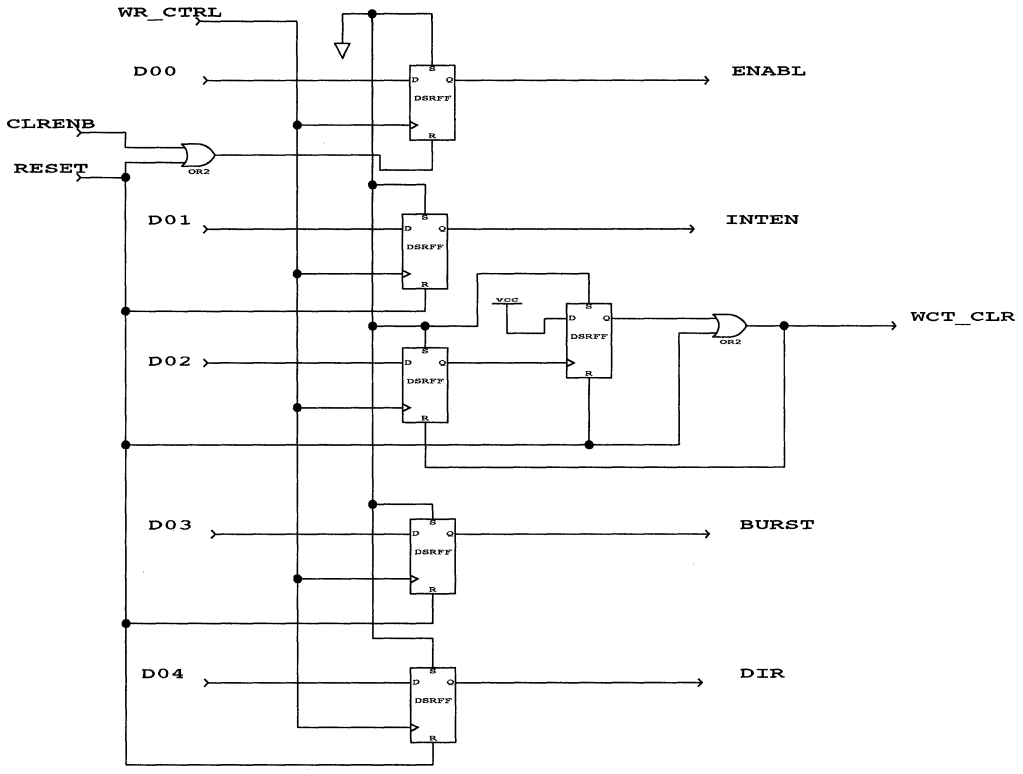


Figure 10. Control Register Schematic

Table 2. DMAC Control Register Bit Definitions

BIT	DEFINITION
0	DMA Controller Enabled (ENABL)
1	Interrupt Enabled (INTEN)
2	Clear Word Counter (1 clears Word Counter and bit 2 to zero)
3	Burst/Single Word Transfer Mode (0=single, 1=Burst)
4	Transfer Direction (0=Mem to I/O, 1=I/O to Mem)
5–15	Not Used

Address Generator

The Address Generator block is a 23-bit synchronous counter that provides the system memory address for the data transfer operation. The 23 address registers are initialized by loading the registers with the address of the first memory location to be accessed. The CPU places the 23-bit starting address on the 16-bit data bus in two operations, one for the lower 16 bits and one for the upper 7 bits. This is controlled by WR_MA_0 and WR_MA_1 signals. After each memory transaction, the state machine block asserts the CT_EN signal which enables the counter to increment. This guarantees that the address is set for the following transfer.

Figure 11 shows the Address Generator diagram. Using the 74XXX TTL functions available in *Warp3*, the address generation function is implemented with six 4-bit, 74161 counters. These counters are arranged so that when each 4-bit counter increments to a binary count of 1111, its ripple carry out output (RCO) enables the next higher 4-bit counter via the ENT and ENP inputs (tied together). The 23 address lines must be three-stated when the CPU has ownership of the system bus. The state machine block generates an output signal called DMAEN which at the appropriate time (during data transfer) enables the DMAC address lines. The three-state buffers must be implemented in the top-level design and they correspond to the internal three-state buffers of the pASIC devices.

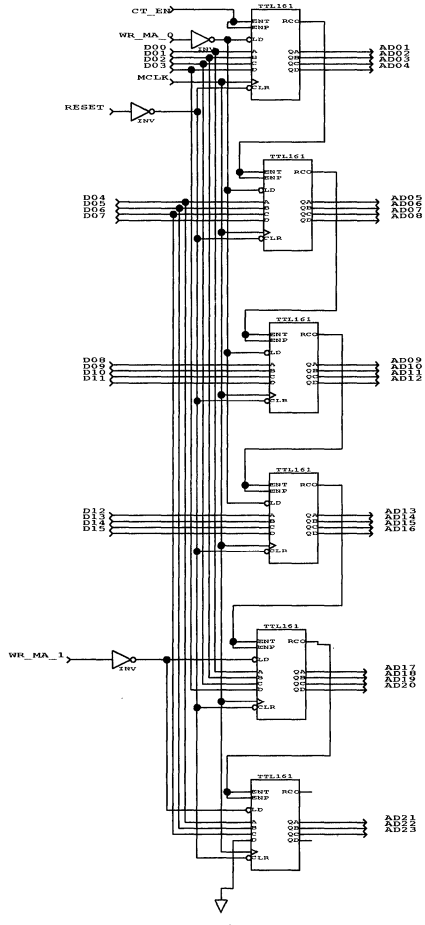


Figure 11. Address Generator Schematic

A symbol is generated for this block as was done for the Control register block.

Word Counter

Because each transfer operation requires a word count, a 16-bit counter monitors the number of words that are transferred. The CPU initializes the Word Counter to a value representing one less than the number of words to be transferred. This value allows the counter to reach zero before the last transfer and terminate the operation at the proper time. Four 74161 counters are used to construct this counter as shown in *Figure 12*. The WR_WCNT signal from the CPUDEC block and the data bits D00 through D15 initialize this counter. The data bits are inverted as they are loaded. Therefore this counter is actually decremented instead of incremented. At the end of each transfer (states *mem2* and *io2*) the CT_EN signal is asserted HIGH. This signal enables both the Address register and the Word Counter blocks. Each time the Address register is incremented, the Word Counter is decremented. The Word Counter is cleared using the RESET signal from the CPU or the WCT_CLR signal from the Control register block written by CPU address and control lines as shown in *Table 1*.

Output Multiplexer

The CPU must have access to the DMAC's internal registers to monitor operation. Therefore, the CPU has the capability of reading the DMAC's current status and configuration. This is signaled to the DMAC by asserting the A01 address line and the IORDI signal HIGH (see *Table 1*). When these two signals along with the CS signal go HIGH, the CPU decoder asserts the RD_ENABL signal HIGH. The required data is then driven to the CPU data bus. The bit definitions for the control signals are essentially the same as those for the control word (on different data bits) and are shown in *Table 3*. A multiplexing scheme is used here to enable the CPU to read either the address generator's lower 16 bits

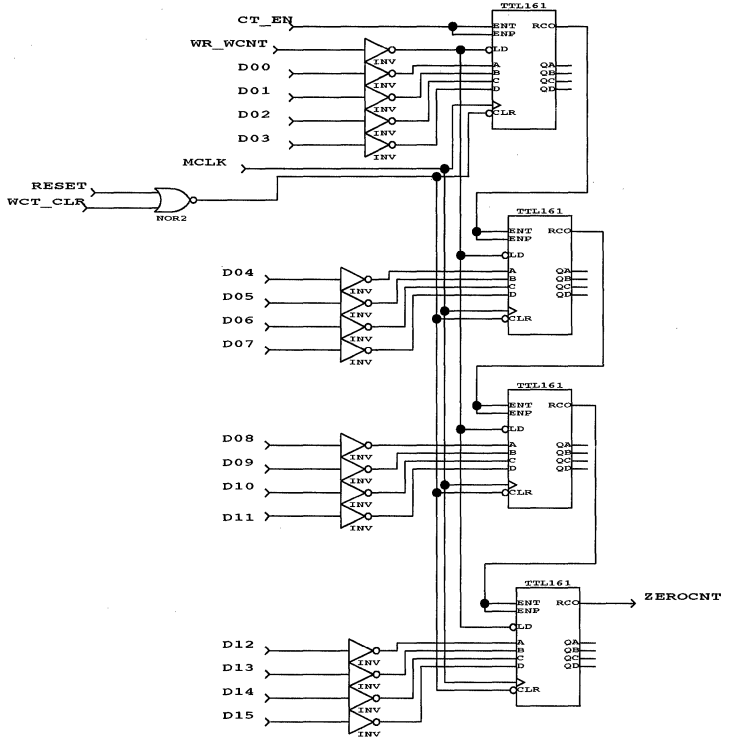


Figure 12. Word Counter Implemented in *Warp3* Schematic Capture Tool

(DATA00–DATA15 when A01=0) or upper 7 bits (DATA00–DATA06) and the DMAC status information (DATA08–DATA09, DATA11–DATA12 when A01=1). *Figure 13* shows the Output Multiplexer.

Table 3. DMAC Status Register Definition

BIT	DEFINITION
8	DMA Controller Enabled (ENABL)
9	Interrupt Enabled (INTEN)
10	Not Used
11	Burst/Single-Word Transfer Mode (0=Single, 1=Burst)
12	Transfer Direction (0=Mem to I/O, 1=I/O to Mem)
13–15	Not Used

DMA Control State Machine

Figure 14 shows the state diagram for the DMA controller. The state machine consists of 9 states: IDLE, HOLD, DIRCT, MEM, IO, ENDST, ENDHLD, CLENB, INTRPT. In IDLE state, the controller waits for an ENABL signal from the Control Register Module. Upon receiving this signal, it goes to the HOLD state and waits for the HLDA signal from the CPU. In DIRCT state, the DMAEN signal is asserted, which enables the three-state buffers that control the address lines. This signal stays asserted through state ENDST. Depending on the Control register content (written by the CPU), data is transferred between the memory and the I/O device. In states IO2 and MEM2, CT_EN is asserted, which in turn increments the Address registers and decrements the Count registers after each transfer. In state ENDST, if all words have been

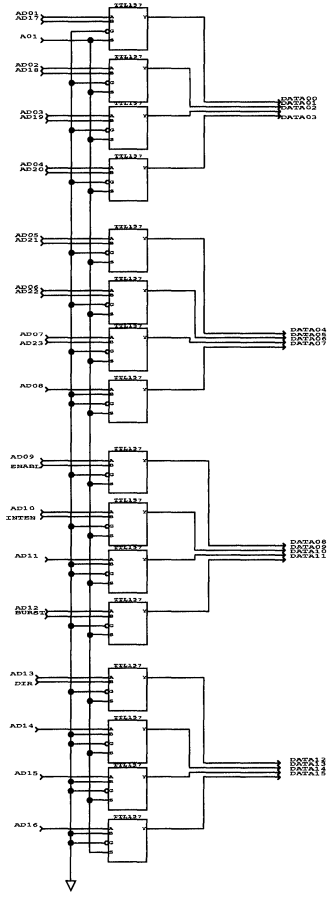


Figure 13. The Output Multiplexer

transferred and there is no Interrupt enable signal from the Control register, then the Control register is cleared.

Figure 15 shows the behavioral description of the DMAC state machine implemented in VHDL. This is a Moore state machine, since the outputs are only a function of the states.

In the Architecture section, we have declared a signal which is a vector that is 11 bits wide. It is called STATE. In this state machine, all the outputs are encoded within state bits. Since there are 10 outputs,

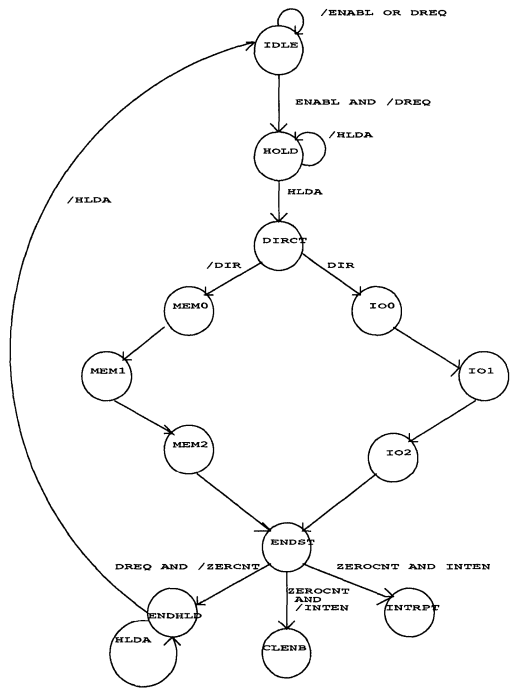


Figure 14. DMA Control State Machine

we need at least 10 state variables. The 11th bit is used to make all state definitions unique. The operation of the state machine is described in the Process section. Notice that behavioral description uses a combination of CASE-WHEN and IF-THEN-ELSE statements. The state machine can asynchronously go to state IDLE. All of the inputs and outputs are defined as BITS in the Entity section. *Warp* assumes that for BIT types '1' is true and '0' is false. After the Process section, all the outputs are assigned to state bits.

Top-LEVEL DMAC Design

After creating lower-level block, each design was compiled and a symbol was created. It's time now to incorporate all the symbols in the DMAC's top-level schematic (*Figure 16*). To accomplish this, each symbol is called and placed on the schematic. To

```
package dma_ctrl is
component  dmas
  port(reset,dreq,hlda,zerocnt,enabl,inten,dir,burst,mclk:in bit;
       ct_en,memw,memr,iowr,iord,dack,dmaen,hreq,setint,clrenb:out bit);

end component;
end dma_ctrl;

entity dmas is
  port(reset,dreq,hlda,zerocnt,enabl,inten,dir,burst,mclk:in bit;
       ct_en,memw,memr,iowr,iord,dack,dmaen,hreq,setint,clrenb:out bit);

end dmas;

architecture machin of dmas is

signal state:bit_vector(10 downto 0);

constant idle   :bit_vector(10 downto 0)   := "00000000000";
constant hold   :bit_vector(10 downto 0)   := "00000001000";
constant dirct  :bit_vector(10 downto 0)   := "00000011000";
constant mem0   :bit_vector(10 downto 0)   := "00100111000";
constant mem1   :bit_vector(10 downto 0)   := "00110111000";
constant mem2   :bit_vector(10 downto 0)   := "00000111001";
constant io0    :bit_vector(10 downto 0)   := "00001111000";
constant io1    :bit_vector(10 downto 0)   := "01001111000";
constant io2    :bit_vector(10 downto 0)   := "10000111001";
constant endst    :bit_vector(10 downto 0)   := "10000011000";
constant intrpt   :bit_vector(10 downto 0)   := "00000001100";
constant endhld   :bit_vector(10 downto 0)   := "10000000000";
constant clenb  :bit_vector(10 downto 0)   := "00000000010";

begin

dma: process (mclk,reset)

  begin

    if reset = '1' then
      state <= idle;
    elsif (mclk'event and mclk = '1') then
```

Figure 15. VHDL Code for DMAC State Machine

```
case state is
  when idle =>
     if (enabl='1' and dreq ='0')  then
    state <= hold;
     end if;

  when hold =>
     if hlda ='1'  then
    state <= dirct;
     end if;

  when dirct =>
     if dir= '1'  then
    state <= io0;
     else
    state <= mem0;
     end if;

  when mem0 =>
    state <= mem1;

  when mem1 =>
    state <= mem2;

  when mem2 =>
    state <= endst;

  when io0 =>
    state <= io1;

  when io1 =>
    state <= io2;

  when io2 =>
    state <= endst;

  when endst =>
     if (dreq='0' and zerocnt='0' and burst='1') then
    state <= dirct;
     elsif (dreq='1'  and zerocnt='0') then
    state <= hold;
     elsif (zerocnt='1' and inten='1') then
    state <= intrpt;
     elsif (zerocnt='1' and inten='0')  then
    state <= clenb;
     end if;
```

Figure 15. VHDL Code for DMAC State Machine (continued)

```
          when intrpt =>
            state <= clenb;

          when clenb =>
            state <= endhld;

          when endhld =>
             if (hlda='0')  then
            state <= idle;
             end if;

          when others =>
            state <= idle;

       end case;
     end if;
end process;

-- assign state outputs to state bits

ct_en <= state(0);
clrenb <= state(1);
setint <= state(2);
hreq <= state(3);
dmaen <= state(4);
dack <= state(5);
iord <= state(6);
iowr <= state(7);
memr <= state(8);
memw <= state(9);

-- bit 10 is to make all state definitions unique.
end machin;
```

Figure 15. VHDL Code for DMAC State Machine (continued)

connect signals, it is sufficient to give them the same names rather than connecting them by wires. The external inputs and outputs are connected to input and output ports. Since the RESET signal is a high fanout signal, an HDPAD is used for distributing this signal across the device. Using an HDPAD insures usage of a dedicated input pin for the signal, giving it twice the current drive capability of the I/O pads. In addition a CKPAD is used for the clock input (MCLK). This uses a clock pin for this signal. The Clock/input pin drives a low-skew, fan-out independent clock tree that can connect to clock, set, or reset inputs of the logic-cell flip-flops. Next *triout* and *bufoe* components are used to implement three-state buffers. The *triout* component has three ports: DATA_IN, ENABLE, and DATA_OUT. The *bufoe* component has four ports: DATA_IN, ENABLE, DATA_OUT, and FEEDBACK. These two types of buffers must be connected to bidirectional pins. In this design, when the CPU has ownership of the system bus, the DMAC's address, memory and I/O control lines are in a high-impedance state. The data bus must also remain in a high-impedance state unless the CPU is reading the DMAC's internal reg-

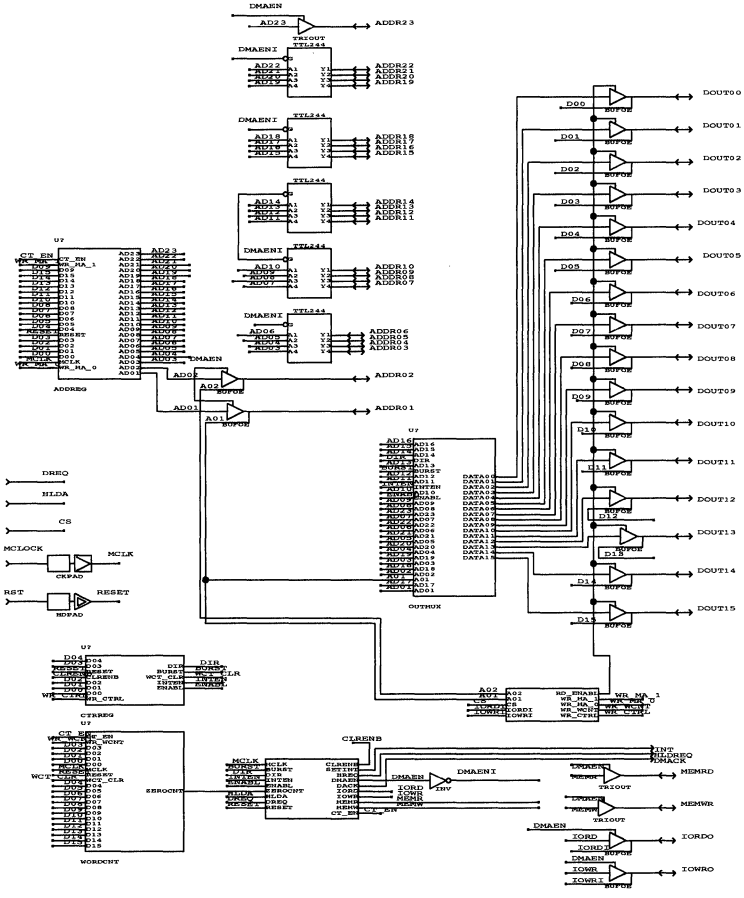


Figure 16. DMAC Top-Level Schematic

isters. The state machine's output, DMAEN, enables the address bus, IORDO, IOWRO, MEMRDO, and MEMWRO outputs. The data outputs are enabled by a RD_ENABL signal from the CPU decoder module. Since signals A01, A02, IORDO, IOWRO, and DATA bus (DOUT00–DOUT15) may be driven by the CPU to initialize the DMAC, *bufoes* (rather than *triout*) are used to connect these signals to bidirectional pins.

A VHDL model for the top-level schematic is then created using EXPT1076 from the *Warp3* Cockpit. The final task remaining is to compile the overall DMAC design and automatically place and route it into a pASIC device. This design easily fits into a CY7C383A. It uses 81 percent of the Logic Cells and 79 percent of the Pad cells.

State Machine Design Considerations and Methodologies

The use of state machines provides a systematic way to design complex sequential logic circuits—an increasingly popular approach since the advent of PLD (Programmable Logic Device) circuitry. This application note describes the many options encountered during the state machine design cycle. By exhaustively walking through the PLD-based design example presented here, you can weigh the merits of several design approaches.

Definitions of Commonly Used Terms

External input vector—External signals (stimulus) applied to the state machine.

System outputs—Signals generated by the state machine that are explicitly designed for availability to the external system (hardware outside of the state machine). Registered system outputs can also be fed back into the state machine as part of the State Vector, which is then used in the decode of the state machine's next state.

State registers—Registers used exclusively for determining the next state of the machine (feedback).

State outputs—Outputs of the state registers that are available to the external system. (They are typically available to the external machine for debug or due to the lack of buried registers.)

State vector or machine state—The registered feedback information defining the present state of the machine and required to determine the next state of the machine.

State path—The transitional condition that must be met for the state machine to progress from one state to another. The state path typically consists of one or more product terms generated from external inputs, although other state paths are possible.

Total input vector—The combination of the external input vector and the state vector. The total input vector is decoded to generate the next state of the machine.

State Machine Entry Methods

There are many ways of describing a state machine, each with distinct advantages and disadvantages. Three popular description methods are state diagrams, state tables, and high-level languages (HLLs). The state diagram provides an easily observable flow description of the state machine. Because the ability to view the flow of states provides distinct documentation advantages, state diagrams will be used throughout this application note to describe the example state machine.

Upon completing a state diagram, you can easily convert the diagram's visual information into the other types of state machine description or directly into Boolean equations. Several available software programs accept their own forms of state table, HLL, and/or Boolean entry. You can enter all these formats easily via your favorite text editor. The software then translates the inputs into suitable forms (usually a JEDEC map) for hardware implementation.

Another method of describing a state machine, the state table, offers perhaps the most concise description. Its major advantage over the other entry methods is the availability of state table reduction methods (see Reference 1). When applied to your state

table definition, a reduction program generates a minimal model for the function. The software used for state machine synthesis throughout this application note uses the state table method of entry. The program is called LOG/iC™ from Isdata Corporation.

Finally, high-level language (HLL) state machine entry is probably the most popular form of state machine design. HLLs typically offer C-language-like instructions (e.g., case, if-then-else, etc.) to describe the machine.

A Sample State Machine

The sample state machine is a clock generator for a pipelined (three system execution stages), bit-slice-based, central processing unit (CPU). Each of the three system execution stages contains two clocks for a total of six system clocks for every instruction execution. With pipelining enabled, each instruction takes an average of two clock periods. Further, external hardware unaffected by CPU wait and stop states (e.g., cache memory) needs both polarities of an additional free-running clock.

To minimize clock edge skew, the state machine provides both versions of the clock. To put the timing of this application into perspective, executing each pipeline stage in an 80-ns period (or 12.5 MHz) requires the state machine to run at 25 MHz. This speed is well within the range of the available PALs, EPLDs and PROMs that can be used to implement the state machine.

Each of the pipeline's three execution stages has a specific function. Briefly, the first stage of the pipeline accesses the Writable Control Store (WCS) RAM. The Arithmetic Logic Unit (ALU) execution occurs during the second stage of the pipeline. Finally, the third pipeline stage clocks status and memory address registers. The function(s) performed during each of the three stages are described in greater detail in the State Machine Output Definition section of this application note.

If this design only generates a simple set of pipelined clocks, why not use shift registers and miscellaneous glue logic instead of a state machine? There are two reasons to consider a state machine. First, it is usually desirable to minimize the number of chips required; the state machine in PLD form might need external glue logic, but significantly less than the shift register solution.

The second reason for considering a state machine is that this application requires more then just a simple set of pipeline clocks. The function of the clock signals is to provide control of the CPU in multiple modes of operation. The desired modes of operation follow.

PIPELINED RUN Mode

In this mode, the CPU simultaneously performs the instructions in all three stages of the pipeline. For example, while instruction n does an ALU operation, instruction n+1 accesses WCS, and instruction n−1 clocks ALU status.

NONPIPELINED RUN Mode

NONPIPELINED RUN mode performs all three stages of instruction execution without overlap. The time to complete one nonpipelined instruction equals the average of three pipelined instructions.

CPU STOP

The system must have a way to perform an orderly stop of CPU execution from both of the above run modes. This stop might be the result of several possible conditions, including a utility stop from a system control unit, a single step, a breakpoint, or a response to external hardware (e.g., a logic analyzer). The free-running clocks continue to run during the CPU STOP mode and remain running at all times, except during a reset condition.

CPU WAIT

In CPU WAIT mode, an external condition causes a delay in an instruction's execution. The instruction pauses until the external condition is removed. One application for the CPU WAIT mode is to handle a cache miss. When a cache miss occurs, the CPU remains in the CPU WAIT mode until the cache completes its memory transfer.

SINGLE STEP

The ability to execute one instruction at a time is needed to debug the CPU. You can easily imple-

ment SINGLE STEP external to the clock state machine by pulsing the RUN signal. SINGLE STEP mode is described further in the State Machine Input Definition section of this application note.

INTERRUPT

A variety of system conditions can interrupt the CPU out of its normal execution sequence and immediately start the execution of the interrupt handler. The influence of the INTERRUPT mode on the system clocks will be discussed in greater detail later in this application note.

REPEAT INSTRUCTION

The REPEAT INSTRUCTION mode is a CPU debug feature. It is a good idea to implement this mode external to the clock state machine. By dubbing the clock to the instruction register and the interrupt line to the clock state machine, the CPU continually executes the instruction in the instruction register.

Synchronous vs. Asynchronous Machine

At this point in the state machine design, an appropriate type of state machine must be chosen to match the application. Two major types are the asynchronous and the synchronous implementations. The asynchronous machine changes state when one or more of its inputs changes from a previously stable input state. After a state change, the outputs of the state machine settle, while the machine stabilizes once again. A basic example of an asynchronous state machine would be a simple SR latch built from two NAND gates (*Figure 1*). For the clocking application considered in this application note, the asynchronous state machine implementation would be a poor choice, due to the instability of the system outputs.

The synchronous state machine offers a better choice. A synchronous state machine block diagram appears in *Figure* 2. Generally, a synchronous state machine samples the total input vector at specific periods to determine the machine's next state. When designing synchronous state machines, it is important to avoid state register metastability. External inputs to the machine must be synchronized to guarantee stable state register inputs, and the feedback time plus data set-up time to the state register clock must be less then or equal to the state clock period.

The modern theory of synchronous state machines was pioneered by Mealy and Moore (see Reference 1). Mealy and Moore machines differ slightly from each other in the way they control the system outputs. During a specific machine state, a Mealy machine allows the input conditions to alter the system outputs (the outputs depend on the "total" input state). In contrast, a Moore machine system outputs depend only on the present machine state. Thus, the system outputs remain stable until the next time period, when the Moore machine samples the total input vector to determine the next state. If all design conditions are met (external inputs are stable prior to the next state clock), the Moore machine provides glitch-free system outputs—a desirable characteristic for the CPU system clock. The design described here is therefore implemented as a Moore machine.

Clock Generator Output Definition

As explained earlier, each of the three system execution stages contains two clocks for a total of six system clocks for every instruction execution. The naming convention for these clocks is

```
CLK_xy
```

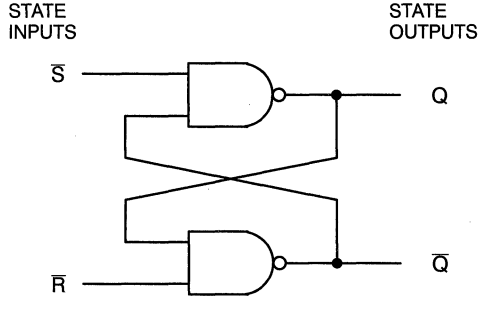


Figure 1. SR Latch, Asynchronous State Machine Example

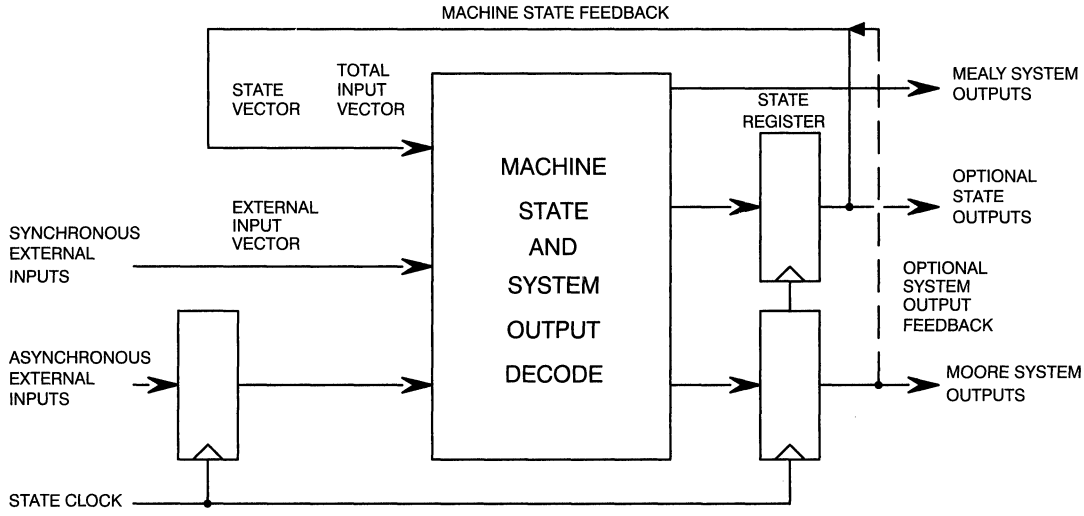


Figure 2. Synchronous State Machine Block Diagram

where x = 1, 2, or 3, representing the first, second, or third stage of the instruction execution and y = A or B, representing the first or second half of the execution stage.

Following this convention, the state machine's two free-running clocks are named CLK_A and CLK_B. These clocks run at half the state clock frequency and 180 degrees out of phase. The free-running clocks occur at the same time as their respective CLK_xA and CLK_xB clocks.

The major clock functions for this application are:

CLK_1B: The leading edge of this clock updates the instruction register.

CLK_2A: This clock's leading edge marks the start of ALU execution. The information on the ALU input bus clocks into the appropriate input registers at this time. The instruction cycle is considered recoverable up through and including CLK_2A (i.e., the status of the machine from the previous instruction has not been altered).

CLK_2B: Used to control the second half of the ALU execution stage, this clock initiates a write to RAM, triggers counters, gates ALU output into its latch, and clocks the ALU output information into any of the distributed destination registers.

CLK_3A: On this clock the memory address register can be updated. The ALU output bus status and ALU status is also clocked into the CPU status register.

Clock Generator Inputs

A set of inputs (external stimulus to the state machine) controls the state machine. The clock state machine described here has eight external inputs, including the state machine clock. These inputs are:

STATECLK: The state machine clock.

RESET: An asynchronous or synchronous reset input that can be connected directly to the state registers' preset or clear or to all clocked register inputs (D or T input). If connected to the preset or clear, RESET need not be synchronized. In this case, RESET forces the state machine into the machine's initial state, regardless of the present state. RESET can result from any combination of the following sources:

- Power up circuit (system reset)
- System controller software decodes system reset

- System controller software decodes module reset
- CPU software decodes module reset

RUN: This signal controls the start and stop sequence of the CPU clocks. In PIPELINE RUN mode, the start sequence generates the proper clock progression to fill up the pipeline registers, and the stop sequence empties the pipeline. RUN is externally manipulated to implement the single step and breakpoint functions.

NPL: Used to select NONPIPELINED RUN vs. PIPELINED RUN modes, this signal must be set to the selected mode prior to activating the RUN signal. Setting NPL = 1 selects NONPIPELINED RUN mode, and NPL = 0 selects PIPELINED RUN mode. The single step function operates properly in NONPIPELINED RUN mode only.

INTR: This signal indicates an external interrupt. When INTR is received, and IEN (interrupt enable, described below) is active, the CPU executes its interrupt handler. An interrupt inhibits the instruction register update clock (CLK_1B) and the ALU update clock (CLK_2B). CLK_1A for the interrupt instruction executes on the next cycle. The interrupt condition has priority over a wait condition and therefore starts generating clocks to permit execution of the interrupt instructions.

IEN: This interrupt enable signal qualifies INTR. IEN is likely to be a bit in the instruction word, allowing the user to define sections of un-interruptable code.

WAIT: The wait condition is initiated when both WAIT and WEN (wait enable, described below) are active. The CPU remains in the wait condition until WAIT goes inactive.

WEN: This wait enable signal qualifies WAIT for entrance into the wait condition. Like IEN, WEN is usually a bit in the instruction word, allowing the user to define sections of wait-sensitive code.

State Machine Partitioning

When architecting a state machine, it is generally a good practice to break up large machines into workable blocks, with each of the smaller machines containing states that require common inputs and generate common outputs. The example clock state machine is small enough to be designed as a single state machine, although it would be trivial to design logic to generate the free-running clocks as a separate machine from the rest of the clock state machine. Equations for the free-running clocks are:

$$CLK\_A := \overline{RESET} \bullet \overline{CLK\_A}$$

$$CLK\_B := \overline{RESET} \bullet CLK\_A$$

where ":=" indicates a registered output.

By examining these output equations, you can see that the free-running clocks have only two dependencies in common with the remaining portion of the clock state machine, i.e., RESET and STATECLK. The free-running clocks are required as inputs to the other state machine to synchronize the additional system outputs, however.

The example presented here implements the free-running clocks and the other system outputs within the same state definition. The resulting output equations can be verified against the equations for the free-running clocks alone.

The Initial Machine State

Regardless of the preferred state machine entry method, attacking the problem starts with defining the initial state of the machine. This initial state (INIT in the example) must be consistent with the power-on condition and/or an external input used to initialize the machine (RESET).

The state of the machine can be decoded from the present values of the system outputs, state registers, or a combination of the two. (The advantages and disadvantages of the state definition options will be discussed in greater detail later in this application note.) The initial machine state is generally, but not always, a decode of all 0s or all 1s. In the example design, INIT is the decode of all 0s.

Naming the States

With the exception of INIT, each state in the example design is named to indicate the active system clocks occurring during that state. For example, during state A, only CLK_A is active. Similarly,

state 123B has only CLK_1B, CLK_2B, CLK_3B, and CLK_B active. Additionally, an "N" suffix designates a nonpipelined state and a "W" suffix designates a wait condition state; this convention differentiates between states with identical active system outputs.

CPU Inactive States

The RESET input causes the state machine to enter the INIT state from any state in the machine. From the INIT state, the machine unconditionally starts to generate the free-running clocks. As shown in *Figure 3*, a line pointing from the INIT state to the A state, with a path equation equal to 1, indicates an unconditional branch. The state machine progression continues from the A state unconditionally into the B state. In the B state a multi-branch condition exists. If the RUN input remains inactive, then the A and B states continue to toggle, generating only the free-running clocks. Hence the INIT, A, and B states are referred to as "CPU inactive states."

Nonpipelined States

If the NPL input is active while the RUN input becomes active, the state machine operates in NONPIPELINED RUN mode and follows the model portrayed in *Figure 4*.

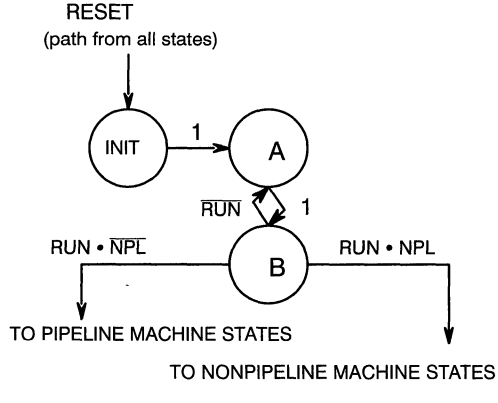


Figure 3. CPU Inactive States

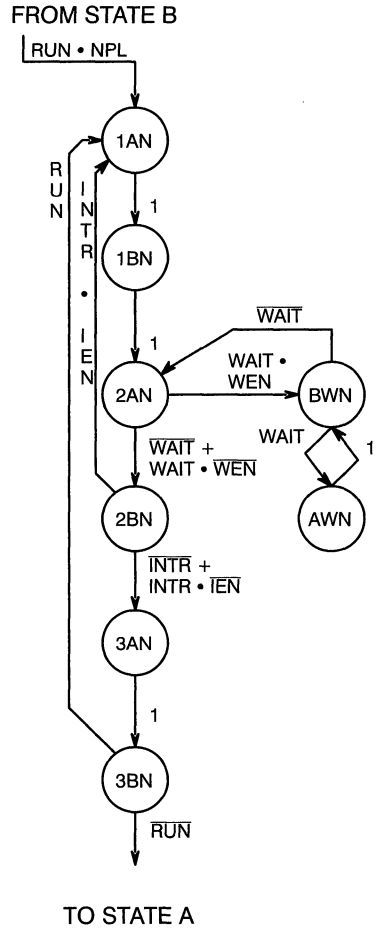


Figure 4. Non-Pipelined States

Pipelined States

If the NPL input is inactive when the RUN input goes active, thus indicating PIPELINED RUN mode, the state machine operates as depicted in *Figure 5*.

Unique States

When the RUN input goes active, the next state executed is either the 1A or the 1AN state, depending upon the value of the NPL input (refer to *Figures 4* and 5). Notice that the active system outputs in these two states are identical. Why generate two

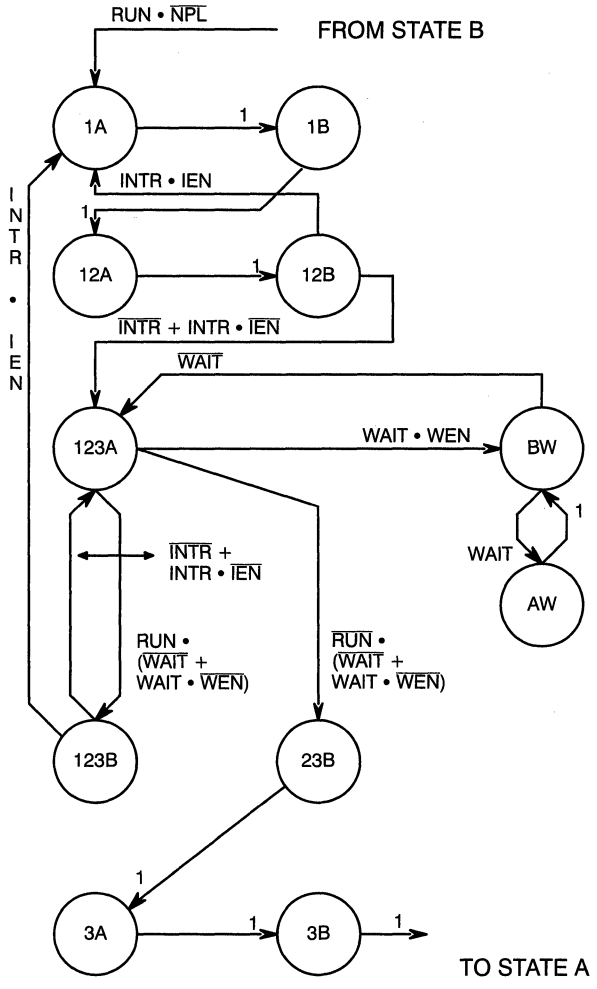


Figure 5. Pipelined States

identical states—when an additional state register might be required to differentiate between the states? (This assumes you use the system outputs to decode the machine's states.) The redundant states are not a problem because the additional state register needed to differentiate between the states is not an issue. There are two reasons for this. First, if you eliminate the redundant states, the state machine would require at least one additional state register anyway to differentiate between the B and the BW or BWN states, which would be needed without 1A and 1AN. (Separation of states BW and BWN from state B is required for correct functionality.) Second, adding another state only increases the number of state registers if the new total number of states exceeds an additional binary boundary (2, 4, 8, 16, ...). This is not a problem here.

You might also choose to widen your state machine (increase the number of state registers) to reduce

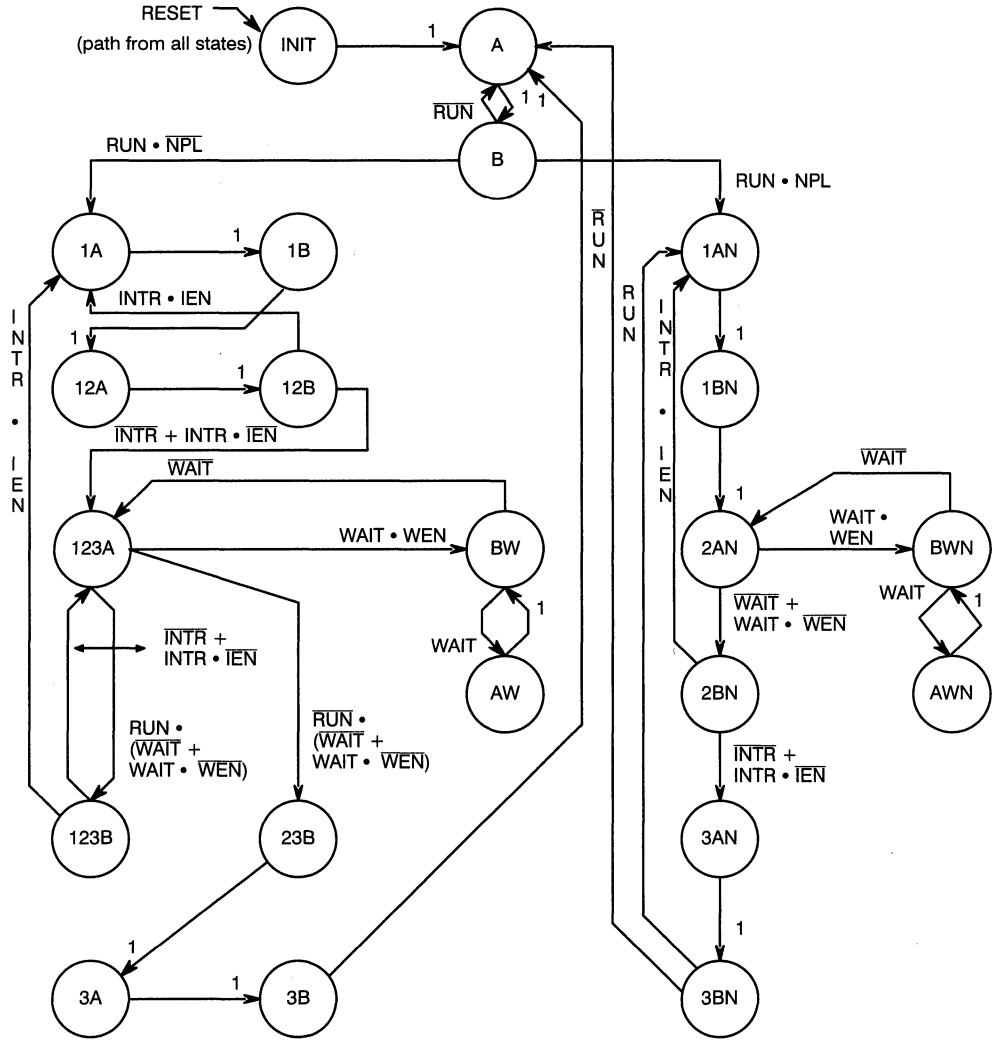


Figure 6. CPU Clock State Machine

the number of product terms to the state or system output registers. This decision should take into account the desired circuit implementation (PLDs, PROMS, discrete hardware, etc.) and is often an iterative process. In general, you can initially architect the state machine in the manner that is the easiest for you to understand, then make additional changes or small adjustments later if they become necessary.

State Description Verification

Now that all the pieces of the state machine are functionally defined (refer to *Figure 6* for the com-

pleted state diagram), consider methods for verifying the validity of the design. Some software you can use to describe and implement state machines would already offer verification at this point in a design. For other methods, read on!

One way to verify a state machine design is to recognize a rule of thumb: Out of every state, there should be a state path to another state for every possible combination of relevant external inputs. For example, there are two paths out of state 123B, with INTR and IEN as the relevant external inputs:

Path 1 = INTR • IEN

Path 2 = $\overline{\text{INTR}}$ + INTR • $\overline{\text{IEN}}$

If there are no known restrictions on the external inputs, a simple method of verifying the above rule of thumb is to generate an equation where all of the paths out of a state are ORed together as follows:

OUT_STATE_123B = Path 1 + Path 2;
OUT_STATE_123B = (INTR • IEN)
+ $\overline{\text{INTR}}$
+ (INTR • $\overline{\text{IEN}}$);
= OUT_STATE_123B = 1

If the equation's terms equal 1 after Boolean reduction, then every state path out of the state is accounted for. The main advantage to this verification method is that you can easily do it using readily available Boolean reduction software.

If there are known restrictions to the external inputs, you can use this information to reduce the complexity of the machine. If it is impossible for the INTR • $\overline{\text{IEN}}$ condition to occur externally, for example, then you can leave this condition out of the Path 2 equation. In that case, the reduction of the OUT_STATE_123B equation yields a non-1 result.

Because the method of verification just described does not detect redundant path equations, it is useful to revise the original rule of thumb to: Out of every state, there should be one and only one state path to another state for every possible combination of relevant external inputs.

This revised condition is not as easily verified as the original statement. The easiest way to verify the more restrictive case is to simulate the state machine. To do this, you must generate a test vector for every possible external input that is relevant to each state simulated. Automatic test vector generation programs are available that produce every possible combination. After running the vectors against the design, you must visually inspect the output to verify that the machine never enters an illegal state.

System and State Register Output Generation

The model defining the clock state machine is complete, but there are still quite a few important decisions to be made regarding the final circuit implementation. Some of the major alternatives for final implementation are:

- System output vs. exclusive state register state decode
- D flip-flop vs. T flip-flop implementation
- PLD vs. PROM implementation

To gain some insight into these choices, consider how the output or feedback equations are assembled. Take, for example, the generation of CLK_3A using a D flip-flop (FF) implementation. By referring to *Figure 6*, you can find all the states in which CLK_3A is active. These are 123A, 3A, and 3AN. The CLK_3A output is generated by ORing the state decodes that, when ANDed with their respective state paths, advance the state machine into the three states listed above. Specifically:

CLK_3A :=
(Decode of 12B)•($\overline{\text{INTR}}$+INTR•$\overline{\text{IEN}}$) ;−123A
+(Decode of BW)•($\overline{\text{WAIT}}$) ;−123A
+(Decode of 23B)•(1) ;−3A
+(Decode of 2BN)•($\overline{\text{INTR}}$+INTR•$\overline{\text{IEN}}$) ;−3AN

When you define the state decodes, the CLK_3A equations are completely specified in terms of the state machine inputs (state path), state registers, and/or system outputs (state decode). Typically, you then multiply the equation out to form a sum of products. This format provides for easy implementation in a PLD, which has a sum-of-products architecture, and also provides a useful foundation for further equation reduction.

State Decode

As discussed earlier, the next state of the machine can be decoded from the present values of the system outputs, the state registers, or a combination of the two. The choice typically comes down to weighing the maximum number of product terms verses the maximum number of flip-flops available in an implementation. For a Moore machine with registered system outputs, using the system outputs to uniquely define the states uses the smallest number of flip-flops to define the state machine. However, it is often necessary to add one or more state registers to uniquely define the states.

State assignment for this state decoding method is quite simple, but also rigidly defined, allowing limited flexibility when assigning the additional state registers. After reduction, the feedback and output equations of this "narrow" state machine might contain too many product terms to be implemented in a specific PLD, although product term complexity is never a problem with a PROM implementation.

Exclusive State Registers

Another consideration in state machine design is that you might be able to distribute the number of product terms more evenly among the equations implementing the state machine by using state registers exclusively to decode the states. Because the state decodes in the state registers can be selected to assist in Boolean reduction, proper state assignment enables the more complex equations to fit into a specific implementation.

This type of decode is useful in a PLD implementation, where there is a shortage of product terms for a specific state flip-flop, but extra flip-flops are available. Adding an extra state register can simplify the decode logic enough to fit the design in a single PLD.

The total number of exclusive state registers required to implement a state machine varies from a minimum of LOG(2)X (rounded up to the nearest integer) to a maximum of X, where X is the total number of states in the machine. You can iteratively change this number, along with the state assignment, to obtain a suitable solution.

The state assignment itself is a non-trivial issue, with almost limitless possibilities and no known method of obtaining the optimal solution. There are, however, some guidelines that can be used to obtain workable solutions:

1. Two or more states that potentially enter the same state with identical path equations should be adjacent (their binary codes differ in exactly one position). As an example, refer to *Figure* 5. States 12B and 123B both proceed into state 1A if the path condition INTR • IEN is true. When generating the CLK_1A equation, two of the terms of the equation look like this:

```
CLK_1A :=
   (Decode of 12B) • (INTR • IEN)          ;−1A
+ (Decode of 123B) • (INTR • IEN)          ;−1A
   . . .
```

If the decode of 12B and 123B differ in exactly one position, then Boolean reduction (which uses the $A \cdot B + \overline{A} \cdot B = B$ relationship) converts the two product terms into one smaller product term.

2. Two or more states that might proceed into different states with identical path equations, and an identical active output, should be adjacent. This situation occurs in the previous CLK_3A equation, shown again here:

```
CLK_3A :=
(Decode of 12B)•(INTR+INTR•IEN)      ;−123A
+(Decode of BW)•(WAIT)               ;−123A
+(Decode of 23B)•(1)                   ;−3A
+(Decode of 2BN)•(INTR+INTR•IEN);    −3AN
```

Note that if states 12B and 2BN are adjacent, then you can reduce the CLK_3A equation to three product terms.

Clock Generator Implementation

As mentioned earlier, there are many ways to implement state machines. The following sections discuss some of the pros and cons associated with some of the more common state machine implementations.

D Flip-Flop Implementation

There are more products available that support a D flip-flop solution than any other implementation. Therefore, it is usually the most cost-effective solution for a state machine.

Table 1 lists the number of product terms per output obtained by compiling the clock generator state machine definition with the LOG/iC software, using D flip-flops. The compiler input file appears in Appendix A. Optimizing the design (*Table 2*) significantly reduces the number of product terms needed.

Table 1. Optimized Results for Clock Generator: T Flip-Flop Implementation

LOG/iC Optimization Summary (FACT)				
CPU Time Quota per Function: 100 sec				
Function	**INV**	**P-Terms**	**CPU-Time**	**Flags**
CLK_1A.T	No	6	<1	
	Yes	7	1	
CLK_1B.T	No	4	1	
	Yes	3	1	
CLK_2A.T	No	5	1	
	Yes	4	<1	
CLK_2B.T	No	4	1	
	Yes	3	<1	
CLK_3A.T	No	5	<1	
	Yes	6	2	
CLK_3B.T	No	4	<1	
	Yes	2	<1	
CLK_A.T	No			C
	Yes			C
CLK_B.T	No	2	1	
	Yes	1	<1	
QQ1.T	No	3	<1	
	Yes	5	1	
QQ2.T	No	6	<1	
	Yes	11	2	

C: Constant function
FACT Minimization: 11 sec

Table 2. Non-optimized Results for Clock Generator: D Flip-Flop Implementation

Log/IC Optimization Summary (FACT)				
CPU Time Quota per Function: 100 sec				
Function	**INV**	**P-Terms**	**CPU-Time**	**Flags**
CLK_1A.D	No	12	<1	N
	Yes	27	<1	N
CLK_1B.D	No	5	<1	N
	Yes	34	1	N
CLK_2A.D	No	8	<1	N
	Yes	31	<1	N
CLK_2B.D	No	7	<1	N
	Yes	32	<1	N
CLK_3A.D	No	8	<1	N
	Yes	31	<1	N
CLK_3B.D	No	6	<1	N
	Yes	33	<1	N
CLK_A.D	No			NT
	Yes			NT
QQ1.D	No	6	<1	N
	Yes	5	<1	N
QQ2.D	No	10	<1	N
	Yes	9	<1	N

N: No Optimization
T: Trivial Function
FACT Minimization: 11 sec

T Flip-Flop Implementation

Even though D flip-flop solutions are more widely available, there are times when the logic needed for this implementation is prohibitively complex. Under these circumstances, a T flip-flop implementation might be more cost effective, because using T flip-flops reduces the logic significantly.

The best example of this situation is a simple synchronous binary counter. While the most significant bit (MSB) of an N-bit counter in a D flip-flop implementation requires N product terms, the T flip-flop solution requires only one product term. Note that the Cypress family of CY7C33x devices offers you a configurable T- or D-type implementation if you

place an XOR gate prior to the D flip-flop; route the AND/OR array to one of the XOR's inputs and the flip-flop's Q output (via an additional product term) to the other XOR input.

It isn't clear from simple observation, however, whether the T flip-flop implementation is beneficial for the clock generator state machine. One way to clarify this question is to change three command lines in the state machine description shown in Appendix A and recompile to produce a T flip-flop implementation. *Table 3* contains the product term results using T flip-flops. A quick study of the results reveals that the optimized version using D flip-flops (*Table 2*) requires fewer product terms than the T flip-flop version.

PLD Implementation

With the LOG/iC PLD Database option, the software assists in selecting a PLD, and it shows that the non-optimized version of the clock state machine fits in a PALC22V10 without further reduction. If the equations are reduced using Boolean reduction, however, a lower-cost solution is available. The results shown in *Table 3* indicate that the less expensive PALC20G10 would work. Appendix A shows the listing for the 20G10 LOG/iC implementation. Waveforms for the completed design appear in Appendix B. You can verify the CLK_A and CLK_B equation results against the equations generated in the State Machine Partitioning section of this application note.

PROM Implementation

You can obtain very high speed solutions by implementing state machines using PROMs. A PROM uses a look-up table to decode the machine's next state, as opposed to the AND/OR array in a PLD. The main advantage of using a look-up table to decode the next state is that every combination of the inputs can be decoded. Thus, you can create an extremely complex machine, without equation reductions.

The look-up table's drawback is that the PROM's depth grows exponentially (2N, where N = # of inputs to the look-up table) with every additional input to the look-up table. To determine the depth required, notice that the present total input vector provides the inputs to the look-up table. The clock generator state machine has seven external inputs, six system outputs, and two state outputs, which indicates a feasible implementation using the CY7C277 (32K x 8) registered PROM.

Table 3. Optimized Results for Clock Generator: D Flip-Flop Implementation

Log/IC Optimization Summary (FACT)				
CPU Time Quota per Function: 100 sec				
Function	**INV**	**P-Terms**	**CPU-Time**	**Flags**
CLK_1A.D	No	6	1	
	Yes	11	2	
CLK_1B.D	No	3	1	
	Yes	4	<1	
CLK_2A.D	No	4	1	
	Yes	7	<1	
CLK_2B.D	No	3	1	
	Yes	4	<1	
CLK_3A.D	No	4	1	
	Yes	9	1	
CLK_3B.D	No	3	<1	
	Yes	3	1	
CLK_A.D	No	1	<1	
	Yes	2	<1	
CLK_B.D	No	1	1	
	Yes	2	<1	
QQ1.D	No	3	<1	
	Yes	3	1	
QQ2.D	No	6	16	
	Yes	6	2	

FACT Minimization: 29 sec

Using a registered PROM such as the CY7C277 to implement the machine also helps to reduce the parts count, because the PROM implements both the state and system output registers. LOG/iC offers support for implementing state machines in PROMs, and only a few minor changes to the state machine description shown in Appendix A are re-

quired. *PROM replaces the *PAL command, some simple statements indicating the CY7C277 architecture (INPUTS = 15 AND OUTPUTS = 8) replaces the TYPE = statement, and PROGFORMAT = INTEL–HEX.

Reference

1. Donald D. Givone, Introduction to Switching Circuit Theory (New York: McGraw-Hill, Inc., 1970)

Appendix A. LOG/iC PLD Source Code: Clock State Machine

```
LOG/iC-PAL              Rel 3.2/2-2328-1721/00034 #32-5955 90/03/15  23:49:45
-------------------------------------------------------------------------------
LOG/iC - COPYRIGHT (C) 1985,1988 BY ISDATA GMBH, 7500 KARLSRUHE WEST-GERMANY
Cypress Semiconductor                                  LICENCE FOR IBM-PC/XT/AT
Data Set: OD20G10.DCB
  1     1: *IDENTIFICATION
  2     2: PIPELINED CLOCKING SYSTEM OD20G10
  3     3: ERIC B.  ROSS
  4     4: CYPRESS SEMICONDUCTOR
  5     5: NAMING CONVENTION
  6     6:  OD      = SYSTEM OUTPUTS ARE DFLOPS AND ARE USED FOR STATE DEF
  7     7:  20G10   = PALC20G10 IMPLEMENTATION
  8     8: *PAL
  9     9:  TYPE=PALC20G10
 10 I  10:
 11    11: *X-NAMES
 12 I  12:
;-------------------------------------------------------------------------
 13 I  13: ;INPUT DEFINITIONS :
 14 I  14: ; RUN     = START & STOP EXECUTION OF OUTPUT CLOCKS (NORMAL, SINGLE
 15 I  15: ;           STEP, & BREAK PT.  EXECUTION
 16 I  16: ; NPL     = PIPELINED VS NON-PIPELINED MODE OF EXECUTION
 17 I  17: ; INTR    = EXTERNAL INTERRUPT CONDITION (TLB MISS, PARITY ERROR,...)
 18 I  18: ; IEN     = INTERRUPT ENABLE
 19 I  19: ; WAIT    = WAIT ENABLE (CACHE MISS)
 20 I  20: ; WEN     = WAIT ENABLE
 21 I  21:
;-------------------------------------------------------------------------
 22 I  22: ;
 23    23:  RUN, NPL, INTR, IEN, WAIT, WEN, RESET;
 24 I  24:
 25    25: *Z-NAMES
 26 I  26:
;-------------------------------------------------------------------------
 27 I  27: ;OUTPUT DEFINITIONS :
 28 I  28: ;
 29 I  29: ; 3 CLOCK STAGES 1, 2, 3
 30 I  30: ; 2 CLOCKS PER STATE A, B
 31 I  31: ;    CLK_XX WHERE XX = 1A,1B,2A,2B,3A,3B
 32 I  32: ;
 33 I  33: ; 2 FREE RUNNING CLOCKS
 34 I  34: ;    CLK_A, CLK_B
 35 I  35: ;
 36 I  36: ; ADDITIONAL REGISTERS FOR STATE DEFINITION
 37 I  37: ;    QQ1, QQ2
 38 I  38:
;-------------------------------------------------------------------------
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 39 I  39: ;
 40    40: CLK_1A, CLK_1B, CLK_2A, CLK_2B, CLK_3A, CLK_3B, CLK_A, CLK_B, QQ1,
QQ2;
 41 I  41:
 42    42: *Z-VALUES
 43 I  43:
 44 I  44: ;                              ADDITIONAL OUTPUTS
 45 I  45: ;       SYSTEM OUTPUTS         FOR STATE DEFINITION
 46 I  46: ;       ________________       ____________________
 47 I  47: ;
 48 I  48: ;       C C C C C C C C   Q Q
 49 I  49: ;       L L L L L L L L   Q Q
 50 I  50: ;       K K K K K K K K   1 2
 51 I  51: ;       1 1 2 2 3 3 A B
 52 I  52: ;       A B A B A B
 53 I  53:
 54    54:  S1  = 0 0 0 0 0 0 0 0   - -          ; INIT  COMMON STATES
 55    55:  S2  = 0 0 0 0 0 0 1 0   0 -          ; SA       - INACTIVE
 56    56:  S3  = 0 0 0 0 0 0 0 1   0 -          ; SB         MODE STATES
 57 I  57:
 58    58:  S4  = 1 0 0 0 0 0 1 0   - 0          ; S1A    PIPELINE STATES
 59    59:  S5  = 0 1 0 0 0 0 0 1   - 0          ; S1B
 60    60:  S6  = 1 0 1 0 0 0 1 0   - -          ; S12A
 61    61:  S7  = 0 1 0 1 0 0 0 1   - -          ; S12B
 62    62:  S8  = 1 0 1 0 1 0 1 0   - -          ; S123A
 63    63:  S9  = 0 1 0 1 0 1 0 1   - -          ; S123B
 64    64:  S10 = 0 0 0 1 0 1 0 1   - -          ; S23B 65    65:  S11 = 0 0 0
0 1 0 1 0   - 0           ; S3A
 66    66:  S12 = 0 0 0 0 0 1 0 1   - 0          ; S3B
 67    67:  S13 = 0 0 0 0 0 0 1 0   1 0          ; SAW
 68    68:  S14 = 0 0 0 0 0 0 0 1   1 0          ; SBW
 69 I  69:
 70    70:  S15 = 1 0 0 0 0 0 1 0   - 1          ; S1AN   NON-PIPELINE
 71    71:  S16 = 0 1 0 0 0 0 0 1   - 1          ; S1BN
 72    72:  S17 = 0 0 1 0 0 0 1 0   - -          ; S2AN
 73    73:  S18 = 0 0 0 1 0 0 0 1   - -          ; S2BN
 74    74:  S19 = 0 0 0 0 1 0 1 0   - 1          ; S3AN
 75    75:  S20 = 0 0 0 0 0 1 0 1   - 1          ; S3BN
 76    76:  S21 = 0 0 0 0 0 0 1 0   1 1          ; SAWN
 77    77:  S22 = 0 0 0 0 0 0 0 1   1 1          ; SBWN
 78 I  78:
 79    79: *STRING
 80    80:  INIT    = 1                          ;  COMMON STATES
 81    81:  SA      = 2                          ;   -INACTIVE MODE
 82    82:  SB      = 3                          ;    STATES
 83 I  83:
 84    84:  S1A     = 4                          ;  PIPELINE STATES
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 85    85:  S1B     = 5                                ;
 86    86:  S12A    = 6                                ;
 87    87:  S12B    = 7                                ;
 88    88:  S123A   = 8                                ;
 89    89:  S123B   = 9                                ;
 90    90:  S23B    = 10                               ;
 91    91:  S3A     = 11                               ;
 92    92:  S3B     = 12                               ;
 93    93:  SAW     = 13                               ;
 94    94:  SBW     = 14                               ;
 95 I  95: 96    96:  S1AN    = 15                                ;  NON-PIPELINE
 97    97:  S1BN    = 16                               ;
 98    98:  S2AN    = 17                               ;
 99    99:  S2BN    = 18                               ;
100   100:  S3AN    = 19                               ;
101   101:  S3BN    = 20                               ;
102   102:  SAWN    = 21                               ;
103   103:  SBWN    = 22                               ;
104   104:  LASTSTATE = 22;
105 I 105:
106   106: *FLOW-TABLE
107 I 107: ;
108 I 108:
;-------------------------------------------------------------------------
109 I 109: ;RESET STATE
110 I 110: ;ALL STATES MUST RESET TO THE INITIAL STATE (ALL OUTPUTS REGISTERS 0)
UPON
111 I 111: ;AN ACTIVE RESET INPUT.  SINCE THE 20G10 HAS NO GLOBAL OR INDIVIDUAL
112 I 112: ;RESETS TO THE OUTPUT REGISTERS, RESET TO INITIAL STATE MUST BE EM-
BEDDED
113 I 113: ;INTO THE STATE MACHINE
114 I 114: ;
115   115: RELEVENT = RESET                            ;
116   116: S[1..'LASTSTATE'], X 1      , F 'INIT'   ;ALL STATE > INIT UPON RESET
117   138: RELEVENT = RESET = 0                       ;
118 I 139: ;
119 I 140:
;-------------------------------------------------------------------------
120 I 141: ;INACTIVE MODE STATES
121   142: RELEVANT = RUN, NPL                        ;
122   143: S 'INIT'    , X - -          , F 'SA'     ;INITIAL STATE AFTER RESET
123 I 144:
124   145: S 'SA'      , X - -          , F 'SB'     ;INACTIVE MODE STATE, ONLY
125 I 146:
126   147: S 'SB'      , X 0 -          , F 'SA'     ;FREE RUN CLKS A & B ARE AC-
TIVE
127   148:               X 1 0          , F 'S1A'    ; PIPELINE VS.
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
128   149:              X 1 1          , F 'S1AN'   ; NON-PIPELINE DECISION
129 I 150:
130 I 151:
;-------------------------------------------------------------------------
131 I 152: ;PIPELINE MODE STATES
132 I 153:
133   154: RELEVANT = INTR, IEN                      ;*PRIMING THE PIPELINE*
134   155: S 'S1A'     , X - -          , F 'S1B'    ;
135 I 156:
136   157: S 'S1B'     , X - -          , F 'S12A'   ;
137 I 158:
138   159: S 'S12A'    , X - -          , F 'S12B'   ;
139 I 160:
140   161: S 'S12B'    , X 1 1          , F 'S1A'    ;   INTERRUPT CONDITION ? YES
141   162:               X 1 0          , F 'S123A'  ;                           NO
142   163:               X 0 -          , F 'S123A'  ;                           NO
143 I 164:
144   165: RELEVANT =  RUN, INTR, IEN, WAIT, WEN; *FULL PIPELINE*
145   166: S 'S123A'   , X - - - 1 1 , F 'SBW'      ;   WAIT CONDITION
146   167:               X 0 - - 0 - , F 'S23B'     ;   /RUN COND., EMPTY PIPELINE
147   168:               X 0 - - 1 0 , F 'S23B'     ;   /RUN COND., EMPTY PIPELINE
148   169:               X 1 - - 0 - , F 'S123B'    ;   RUN CONDITION
149   170:               X 1 - - 1 0 , F 'S123B'    ;   RUN CONDITION
150 I 171:
151   172: S 'S123B'   , X - 1 1 - - , F 'S1A'      ;   INTERUPT CONDITION
152   173:               X - 0 - - - , F 'S123A'    ;   RUN CONDITION
153   174:               X - 1 0 - - , F 'S123A'    ;   RUN CONDITION
154 I 175:
155   176: RELEVANT = RUN                            ; *EMPTY PIPELINE*
156   177: S 'S23B'    , X -            , F 'S3A'    ;
157 I 178:
158   179: S 'S3A'     , X -            , F 'S3B'    ;
159 I 180:
160   181: S 'S3B'     , X -            , F 'SA'     ;   BACK TO INACTIVE STATE
161 I 182:
162   183: RELEVANT = WAIT                           ; *PIPELINE WAIT STATES*
163   184: S 'SBW'     , X 1            , F 'SAW'    ;   WAIT
164   185:               X 0            , F 'S123A'  ;   /WAIT
165 I 186:
166   187: S 'SAW'     , X -            , F 'SBW'    ;
167 I 188:
168 I 189:
;-------------------------------------------------------------------------
169 I 190: ;NON-PIPELINE MODE STATES
170 I 191:
171   192: S 'S1AN'    , X -            , F 'S1BN'   ;
172 I 193:
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
173   194: S 'S1BN'   , X -           , F 'S2AN'  ;
174 I 195:
175   196: RELEVANT = WAIT, WEN                   ;
176   197: S 'S2AN'   , X 1 1       , F 'SBWN'    ; WAIT CONDITION
177   198:              X 0 -       , F 'S2BN'    ; /WAIT CONDITION
178   199:              X 1 0       , F 'S2BN'    ; /WAIT CONDITION
179 I 200:180   201: RELEVANT = INTR, IEN                    ;
181   202: S 'S2BN'   , X 1 1       , F 'S1AN'    ; INTERRUPT CONDITION
182   203:              X 0 -       , F 'S3AN'    ; /INTERRUPT CONDITION
183   204:              X 1 0       , F 'S3AN'    ; /INTERRUPT CONDITION
184 I 205:
185   206: RELEVANT = RUN                         ;
186   207: S 'S3AN'   , X -           , F 'S3BN'  ;
187 I 208:
188   209: S 'S3BN'   , X 1           , F 'S1AN'  ;
189   210:              X 0           , F 'SA'    ; BACK TO INACTIVE STATE
190 I 211:
191   212: RELEVANT = WAIT                        ;*NON-PIPELINED WAIT STATES*
192   213: S 'SBWN'   , X 1           , F 'SAWN'  ;  REMAIN IN WAIT
193   214:              X 0           , F 'S2AN'  ;  END OF WAIT CONDITION
194 I 215:
195   216: S 'SAWN'   , X -           , F 'SBWN'  ;  REMAIN IN WAIT
196 I 217:
197   218: *STATE-ASSIGNMENT
198   219:  Z-VALUES
199 I 220:
200 I 221:
201   222: *PIN
202   223:  STATECLK = 1, RUN = 2, NPL = 3, INTR = 4, IEN = 5, WAIT = 6, WEN =
7,
203   223:  RESET = 8, CLK_1A = 14, CLK_1B = 15, CLK_2A = 16, CLK_2B = 17,
204   223:  CLK_3A = 18, CLK_3B = 19, CLK_A = 20, CLK_B = 21, QQ1 = 22, QQ2 =
23;
205 I 224:
206   225: *RUN-CONTROL207   226:  LISTING= LONG,SYMBOL-TABLE,EQUATIONS,PIN-
OUT;208   227:  PROGFORMAT= L-EQUATIONS
209   228:  OPTIMIAZATION= P-TERMS;
210   229: *END
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

LOG/IC SYMBOL TABLE				
SYMBOL	TYPE	REG	LEVEL	PIN/NODE
GND	LOCAL	-	HIGH	
VCC	LOCAL	-	HIGH	
RUN	X-VARIABLE	-	HIGH	2
NPL	X-VARIABLE	-	HIGH	3
INTR	X-VARIABLE	-	HIGH	4
IEN	X-VARIABLE	-	HIGH	5
WAIT	X-VARIABLE	-	HIGH	6
WEN	X-VARIABLE	-	HIGH	7
RESET	X-VARIABLE	-	HIGH	8
CLK_1A	X-VARIABLE	-	HIGH	14
CLK_1B	X-VARIABLE	-	HIGH	15
CLK_2A	X-VARIABLE	-	HIGH	16
CLK_2B	X-VARIABLE	-	HIGH	17
CLK_3A	X-VARIABLE	-	HIGH	18
CLK_3B	X-VARIABLE	-	HIGH	19
CLK_A	X-VARIABLE	-	HIGH	20
CLK_B	X-VARIABLE	-	HIGH	21
QQ1	X-VARIABLE	-	HIGH	22
QQ2	X-VARIABLE	-	HIGH	23
CLK_1A.D	Z-VARIABLE	DFF	HIGH	14
CLK_1B.D	Z-VARIABLE	DFF	HIGH	15
CLK_2A.D	Z-VARIABLE	DFF	HIGH	16
CLK_2B.D	Z-VARIABLE	DFF	HIGH	17
CLK_3A.D	Z-VARIABLE	DFF	HIGH	18
CLK_3B.D	Z-VARIABLE	DFF	HIGH	19
CLK_A.D	Z-VARIABLE	DFF	HIGH	20
CLK_B.D	Z-VARIABLE	DFF	HIGH	21
QQ1.D	Z-VARIABLE	DFF	HIGH	22
QQ2.D	Z-VARIABLE	DFF	HIGH	23

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
EXPANDED FUNCTION TABLE (INCLUDING LOCAL VARIABLES):
----------------------------------------------------
                                  :  CCCC CC
                                  :  LLLL LLCC
                 CCC CCC          :  KKKK KKLL
                RLLL LLLC C       :  _____ __KK QQ
      I W       EKKK KKKL L       :  1122 33__ QQ
  GVRN NIAW     S___ ___K KQQ     :  ABAB ABAB 12
  NCUP TEIE     E112 233_ _QQ     :  ....  ....  ..
  DCNL RNTN     TABA BABA B12     :  DDDD DDDD DD
  ---------------------------------------------
  ---- ----  1000 0000 0--   :  0000 0000 --;     1/  116
  ---- ----  0000 0000 0--   :  0000 0010 0-;     2/  143
  ---- ----  1000 0001 00-   :  0000 0000 --;     3/  117
  ---- ----  0000 0001 00-   :  0000 0001 0-;     4/  145
  ---- ----  1000 0000 10-   :  0000 0000 --;     5/  118
  --0- ----  0000 0000 10-   :  0000 0010 0-;     6/  147
  --10 ----  0000 0000 10-   :  1000 0010 -0;     7/  148
  --11 ----  0000 0000 10-   :  1000 0010 -1;     8/  149
  ---- ----  1100 0001 0-0   :  0000 0000 --;     9/  119
  ---- ----  0100 0001 0-0   :  0100 0001 -0;    10/  155
  ---- ----  1010 0000 1-0   :  0000 0000 --;    11/  120
  ---- ----  0010 0000 1-0   :  1010 0010 --;    12/  157
  ---- ----  1101 0001 0--   :  0000 0000 --;    13/  121
  ---- ----  0101 0001 0--   :  0101 0001 --;    14/  159
  ---- ----  1010 1000 1--   :  0000 0000 --;    15/  122
  ---- 11--  0010 1000 1--   :  1000 0010 -0;    16/  161
  ---- 10--  0010 1000 1--   :  1010 1010 --;    17/  162
  ---- 0---  0010 1000 1--   :  1010 1010 --;    18/  163
  ---- ----  1101 0101 0--   :  0000 0000 --;    19/  123
  ---- --11  0101 0101 0--   :  0000 0001 10;    20/  166
  --0- --0-  0101 0101 0--   :  0001 0101 --;    21/  167
  --0- --10  0101 0101 0--   :  0001 0101 --;    22/  168
  --1- --0-  0101 0101 0--   :  0101 0101 --;    23/  169
  --1- --10  0101 0101 0--   :  0101 0101 --;    24/  170
  ---- ----  1010 1010 1--   :  0000 0000 --;    25/  124
  ---- 11--  0010 1010 1--   :  1000 0010 -0;    26/  172
  ---- 0---  0010 1010 1--   :  1010 1010 --;    27/  173
  ---- 10--  0010 1010 1--   :  1010 1010 --;    28/  174
  ---- ----  1000 1010 1--   :  0000 0000 --;    29/  125
  ---- ----  0000 1010 1--   :  0000 1010 -0;    30/  177
  ---- ----  1000 0101 0-0   :  0000 0000 --;    31/  126
  ---- ----  0000 0101 0-0   :  0000 0101 -0;    32/  179
  ---- ----  1000 0010 1-0   :  0000 0000 --;    33/  127
  ---- ----  0000 0010 1-0   :  0000 0010 0-;    34/  181
  ---- ----  1000 0001 010   :  0000 0000 --;    35/  128
  ---- ----  0000 0001 010   :  0000 0001 10;    36/  187
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
---- ---- 1000 0000 110  :  0000 0000 --;    37/  129
---- --1- 0000 0000 110  :  0000 0010 10;    38/  184
---- --0- 0000 0000 110  :  1010 1010 --;    39/  185
---- ---- 1100 0001 0-1  :  0000 0000 --;    40/  130
---- ---- 0100 0001 0-1  :  0100 0001 -1;    41/  192
---- ---- 1010 0000 1-1  :  0000 0000 --;    42/  131
---- ---- 0010 0000 1-1  :  0010 0010 --;    43/  194
---- ---- 1001 0001 0--  :  0000 0000 --;    44/  132
---- --11 0001 0001 0--  :  0000 0001 11;    45/  197
---- --0- 0001 0001 0--  :  0001 0001 --;    46/  198
---- --10 0001 0001 0--  :  0001 0001 --;    47/  199
---- ---- 1000 1000 1--  :  0000 0000 --;    48/  133
---- 11-- 0000 1000 1--  :  1000 0010 -1;    49/  202
---- 0--- 0000 1000 1--  :  0000 1010 -1;    50/  203
---- 10-- 0000 1000 1--  :  0000 1010 -1;    51/  204
---- ---- 1000 0101 0-1  :  0000 0000 --;    52/  134
---- ---- 0000 0101 0-1  :  0000 0101 -1;    53/  207
---- ---- 1000 0010 1-1  :  0000 0000 --;    54/  135
--1- ---- 0000 0010 1-1  :  1000 0010 -1;    55/  209
--0- ---- 0000 0010 1-1  :  0000 0010 0-;    56/  210
---- ---- 1000 0001 011  :  0000 0000 --;    57/  136
---- ---- 0000 0001 011  :  0000 0001 11;    58/  216
---- ---- 1000 0000 111  :  0000 0000 --;    59/  137
---- --1- 0000 0000 111  :  0000 0010 11;    60/  213
---- --0- 0000 0000 111  :  0010 0010 --;    61/  214
REST                     :  ---- ---- --;    62
-----------------------------------------
1234 5678 9012 3456 789     1234 5678 90
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
  STATE ASSIGNMENT:
-----------------
    CCCC CC
    LLLL LLCC
    KKKK KKLL
    ____ __KK QQ
    1122 33__ QQ
    ABAB ABAB 12
    ------------
    0000 0000 --; 1
    0000 0010 0-; 2
    0000 0001 0-; 3
    1000 0010 -0; 4
    0100 0001 -0; 5
    1010 0010 --; 6
    0101 0001 --; 7
    1010 1010 --; 8
    0101 0101 --; 9
    0001 0101 --; 10
    0000 1010 -0; 11
    0000 0101 -0; 12
    0000 0010 10; 13
    0000 0001 10; 14
    1000 0010 -1; 15
    0100 0001 -1; 16
    0010 0010 --; 17
    0001 0001 --; 18
    0000 1010 -1; 19
    0000 0101 -1; 20
    0000 0010 11; 21
    0000 0001 11; 22
EXPANDED FUNCTION TABLE (LOCAL VARIABLES REMOVED):
--------------------------------------------------
                           :  CCCC CC
                           :  LLLL LLCC
          C CCCC C         :  KKKK KKLL
         RL LLLL LCC       :  ____ __KK QQ
    I  W EK KKKK KLL       :  1122 33__ QQ
  RNNI AWS_ ____ _KKQ Q    :  ABAB ABAB 12
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
    UPTE IEE1 1223 3__Q Q  :  ....  ....  ..
    NLRN TNTA BABA BAB1 2  :  DDDD DDDD DD
    ---------------------------------------
    ---- --10 0000 000- -  :  0000 0000 --;     1/  116
    ---- --00 0000 000- -  :  0000 0010 0-;     2/  143
    ---- --10 0000 0100 -  :  0000 0000 --;     3/  117
    ---- --00 0000 0100 -  :  0000 0001 0-;     4/  145
    ---- --10 0000 0010 -  :  0000 0000 --;     5/  118
    0--- --00 0000 0010 -  :  0000 0010 0-;     6/  147
    10-- --00 0000 0010 -  :  1000 0010 -0;     7/  148
    11-- --00 0000 0010 -  :  1000 0010 -1;     8/  149
    ---- --11 0000 010- 0  :  0000 0000 --;     9/  119
    ---- --01 0000 010- 0  :  0100 0001 -0;    10/  155
    ---- --10 1000 001- 0  :  0000 0000 --;    11/  120
    ---- --00 1000 001- 0  :  1010 0010 --;    12/  157
    ---- --11 0100 010- -  :  0000 0000 --;    13/  121
    ---- --01 0100 010- -  :  0101 0001 --;    14/  159
    ---- --10 1010 001- -  :  0000 0000 --;    15/  122
    --11 --00 1010 001- -  :  1000 0010 -0;    16/  161
    --10 --00 1010 001- -  :  1010 1010 --;    17/  162
    --0- --00 1010 001- -  :  1010 1010 --;    18/  163
    ---- --11 0101 010- -  :  0000 0000 --;    19/  123
    ---- 1101 0101 010- -  :  0000 0001 10;    20/  166
    0--- 0-01 0101 010- -  :  0001 0101 --;    21/  167
    0--- 1001 0101 010- -  :  0001 0101 --;    22/  168
    1--- 0-01 0101 010- -  :  0101 0101 --;    23/  169
    1--- 1001 0101 010- -  :  0101 0101 --;    24/  170
    ---- --10 1010 101- -  :  0000 0000 --;    25/  124
    --11 --00 1010 101- -  :  1000 0010 -0;    26/  172
    --0- --00 1010 101- -  :  1010 1010 --;    27/  173
    --10 --00 1010 101- -  :  1010 1010 --;    28/  174
    ---- --10 0010 101- -  :  0000 0000 --;    29/  125
    ---- --00 0010 101- -  :  0000 1010 -0;    30/  177
    ---- --10 0001 010- 0  :  0000 0000 --;    31/  126
    ---- --00 0001 010- 0  :  0000 0101 -0;    32/  179
    ---- --10 0000 101- 0  :  0000 0000 --;    33/  127
    ---- --00 0000 101- 0  :  0000 0010 0-;    34/  181
    ---- --10 0000 0101 0  :  0000 0000 --;    35/  128
    ---- --00 0000 0101 0  :  0000 0001 10;    36/  187
    ---- --10 0000 0011 0  :  0000 0000 --;    37/  129
    ---- 1-00 0000 0011 0  :  0000 0010 10;    38/  184
    ---- 0-00 0000 0011 0  :  1010 1010 --;    39/  185
    ---- --11 0000 010- 1  :  0000 0000 --;    40/  130
EXPANDED FUNCTION TABLE (LOCAL VARIABLES REMOVED)- continued :
    ---- --01 0000 010- 1  :  0100 0001 -1;    41/  192
    ---- --10 1000 001- 1  :  0000 0000 --;    42/  131
    ---- --00 1000 001- 1  :  0010 0010 --;    43/  194
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
      ---- --10 0100 010- -  :  0000 0000 --;    44/  132
      ---- 1100 0100 010- -  :  0000 0001 11;    45/  197
      ---- 0-00 0100 010- -  :  0001 0001 --;    46/  198
      ---- 1000 0100 010- -  :  0001 0001 --;    47/  199
      ---- --10 0010 001- -  :  0000 0000 --;    48/  133
      --11 --00 0010 001- -  :  1000 0010 -1;    49/  202
      --0- --00 0010 001- -  :  0000 1010 -1;    50/  203
      --10 --00 0010 001- -  :  0000 1010 -1;    51/  204
      ---- --10 0001 010- 1  :  0000 0000 --;    52/  134
      ---- --00 0001 010- 1  :  0000 0101 -1;    53/  207
      ---- --10 0000 101- 1  :  0000 0000 --;    54/  135
      1--- --00 0000 101- 1  :  1000 0010 -1;    55/  209
      0--- --00 0000 101- 1  :  0000 0010 0-;    56/  210
      ---- --10 0000 0101 1  :  0000 0000 --;    57/  136
      ---- --00 0000 0101 1  :  0000 0001 11;    58/  216
      ---- --10 0000 0011 1  :  0000 0000 --;    59/  137
      ---- 1-00 0000 0011 1  :  0000 0010 11;    60/  213
      ---- 0-00 0000 0011 1  :  0010 0010 --;    61/  214
      REST                   :  ---- ---- --;    62
      ---------------------------------------
      1234 5678 9012 3456 7     1234 5678 90
  PIPELINED CLOCKING SYSTEM OD20G10
  CYPRESS SEMICONDUCTOR
  90/03/15  23:49:45
  ************************************************
  ***  NET DESCRIPTION TABLE FOR AND/OR STRUCTURE  ***
  ************************************************
                        :  CCCC CC
                        :  LLLL LLCC
        C CCCC C        :  KKKK KKLL
       RL LLLL LCC      :  ____ __KK QQ
     I  W EK KKKK KLL      :  1122 33__ QQ
RNNI AWS_ ____ _KKQ Q   :  ABAB ABAB 12
UPTE IEE1 1223 3__Q Q   :  ....  ....  ..
NLRN TNTA BABA BAB1 2   :  DDDD DDDD DD
---------------------------------------
                     INV   ....  ....  ..
                     REG   DDDD DDDD DD
---------------------------------------
---- 0-0- --0- 0-11 0   :  A...  ....  ..  ;   1
1--- --0- --0- 1--- 1   :  A...  ....  ..  ;   2
---- --0- 1--- ---- 0   :  A...  ....  ..  ;   3
---- --0- 1-1- ---- -   :  A...  ....  ..  ;   4
--11 --0- --1- 0--- -   :  A...  ....  ..  ;   5
1--- --0- 0-0- 0-10 -   :  A...  ....  ..  ;   6
---- --01 ---0 ---- -   :  .A..  ....  ..  ;   7
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

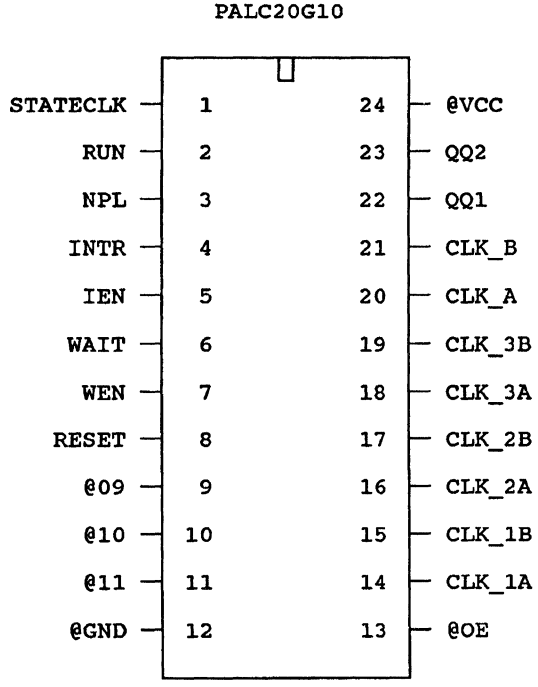
```
 1---  -001  ----  ----  -   :  .A..  ....  ..  ;   8
 1---  0-01  ----  ----  -   :  .A..  ....  ..  ;   9
 ---0  --0-  1---  ----  -   :  ..A.  ....  ..  ;  10
 --0-  --0-  1---  ----  -   :  ..A.  ....  ..  ;  11
 ----  0-0-  --0-  0-11  -   :  ..A.  ....  ..  ;  12
 ----  --0-  1-0-  ----  -   :  ..A.  ....  ..  ;  13
 ----  0-0-  -1--  ----  -   :  ...A  ....  ..  ;  14
 ----  -00-  -1--  ----  -   :  ...A  ....  ..  ;  15
 ----  --01  -1-0  ----  -   :  ...A  ....  ..  ;  16
 ---0  --0-  --1-  ----  -   :  ....  A...  ..  ;  17
 --0-  --0-  --1-  ----  -   :  ....  A...  ..  ;  18
 ----  --0-  0-1-  1---  -   :  ....  A...  ..  ;  19
 ----  0-0-  0-0-  0-11  0   :  ....  A...  ..  ;  20
 ----  0-0-  ---1  ----  -   :  ....  .A..  ..  ;  21
 ----  -00-  ---1  ----  -   :  ....  .A..  ..  ;  22
 ----  --0-  -0-1  ----  -   :  ....  .A..  ..  ;  23
 ----  --0-  ----  -0--  -   :  ....  ..A.  ..  ;  24
 ----  --0-  ----  -1--  -   :  ....  ...A  ..  ;  25
 ----  ----  ----  -1-1  -   :  ....  ....  A.  ;  26
 ----  ----  ----  0-11  -   :  ....  ....  A.  ;  27
 ----  ----  -1--  ----  -   :  ....  ....  A.  ;  28
 ----  ----  --0-  1---  -   :  ....  ....  .A  ;  29
 ----  ----  0-1-  0---  -   :  ....  ....  .A  ;  30
 ----  ---0  -1--  ----  -   :  ....  ....  .A  ;  31
 ----  ----  -0--  -1--  1   :  ....  ....  .A  ;  32
 -1--  ---0  0--0  0--0  -   :  ....  ....  .A  ;  33
 ----  ----  00--  0--1  1   :  ....  ....  .A  ;  34
 ---------------------------------------
1234 5678 9012 3456 7   :  1234 5678 90
   PIPELINED CLOCKING SYSTEM OD20G10
   CYPRESS SEMICONDUCTOR
   90/03/15  23:49:45
   ************************************************
   ***              BOOLEAN EQUATIONS             ***
   ************************************************
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 CLK_1A.D      :=
           /WAIT    & /RESET   & /CLK_2B        & /CLK_3B        &  CLK_B
                    &  QQ1     & /QQ2
         +  RUN     & /RESET   & /CLK_2B        &  CLK_3B        &  QQ2
         + /RESET   &  CLK_1B         & /QQ2
         + /RESET   &  CLK_1B         &  CLK_2B
         +  INTR    &  IEN     & /RESET   &  CLK_2B        & /CLK_3B          +  RUN
 & /RESET   & /CLK_1B         & /CLK_2B         & /CLK_3B
                    &  CLK_B   & /QQ1     ;
 CLK_1B.D      :=
           /RESET   &  CLK_1A         & /CLK_3A
         +  RUN     & /WEN     & /RESET   &  CLK_1A
         +  RUN     & /WAIT    & /RESET   &  CLK_1A        ;
 CLK_2A.D      :=
           /IEN     & /RESET   &  CLK_1B
         + /INTR    & /RESET   &  CLK_1B
         + /WAIT    & /RESET   & /CLK_2B        & /CLK_3B        &  CLK_B
                    &  QQ1
         + /RESET   &  CLK_1B         & /CLK_2B        ;
 CLK_2B.D      :=
           /WAIT    & /RESET   &  CLK_2A
         + /WEN     & /RESET   &  CLK_2A
         + /RESET   &  CLK_1A         &  CLK_2A        & /CLK_3A        ;
 CLK_3A.D      :=
           /IEN     & /RESET   &  CLK_2B
         + /INTR    & /RESET   &  CLK_2B
         + /RESET   & /CLK_1B         &  CLK_2B        &  CLK_3B
         + /WAIT    & /RESET   & /CLK_1B         & /CLK_2B        & /CLK_3B
                    &  CLK_B   &  QQ1     & /QQ2     ;
 CLK_3B.D      :=
           /WAIT    & /RESET   &  CLK_3A
         + /WEN     & /RESET   &  CLK_3A
         + /RESET   & /CLK_2A         &  CLK_3A        ;
 CLK_A.D       :=
           /RESET   & /CLK_A   ; CLK_B.D       :=
           /RESET   &  CLK_A   ;
 QQ1.D   :=  CLK_A  &  QQ1
         + /CLK_3B         &  CLK_B  &  QQ1
         +  CLK_2A         ;
 QQ2.D   := /CLK_2B        &  CLK_3B
         + /CLK_1B         &  CLK_2B        & /CLK_3B
         + /CLK_1A         &  CLK_2A
         + /CLK_2A         &  CLK_A  &  QQ2
         +  NPL     & /CLK_1A         & /CLK_1B         & /CLK_3A
                    & /CLK_3B         & /QQ1
         + /CLK_1B         & /CLK_2A         & /CLK_3B         &  QQ1     &  QQ2     ;
```

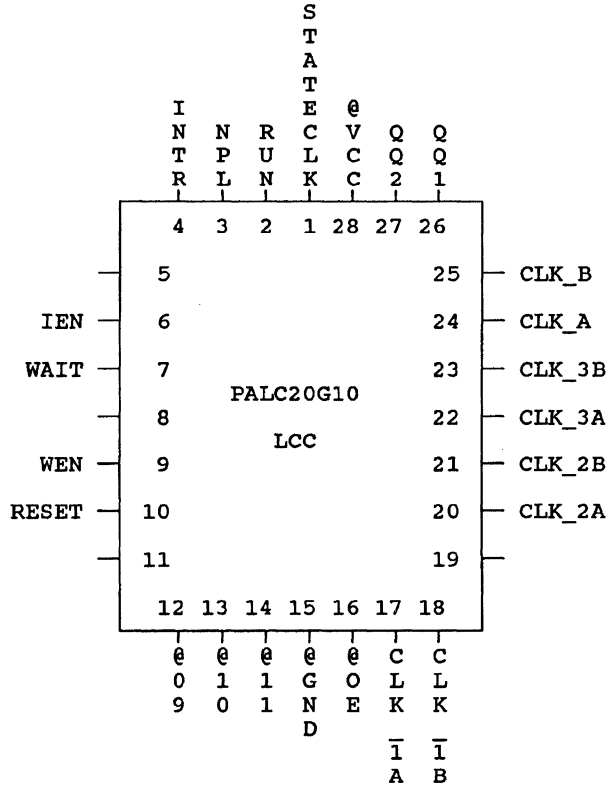
Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 PIPELINED CLOCKING SYSTEM OD20G10
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45
```



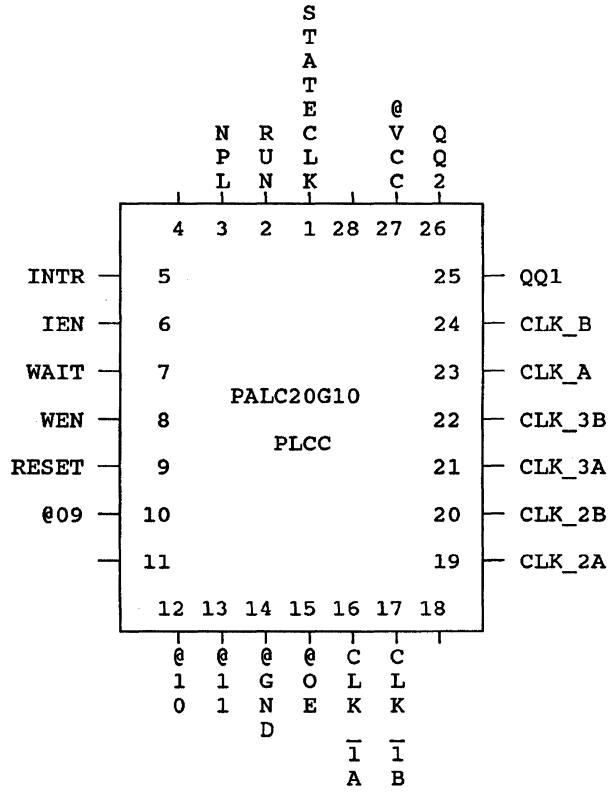
Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
PIPELINED CLOCKING SYSTEM OD20G10
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45
```



Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

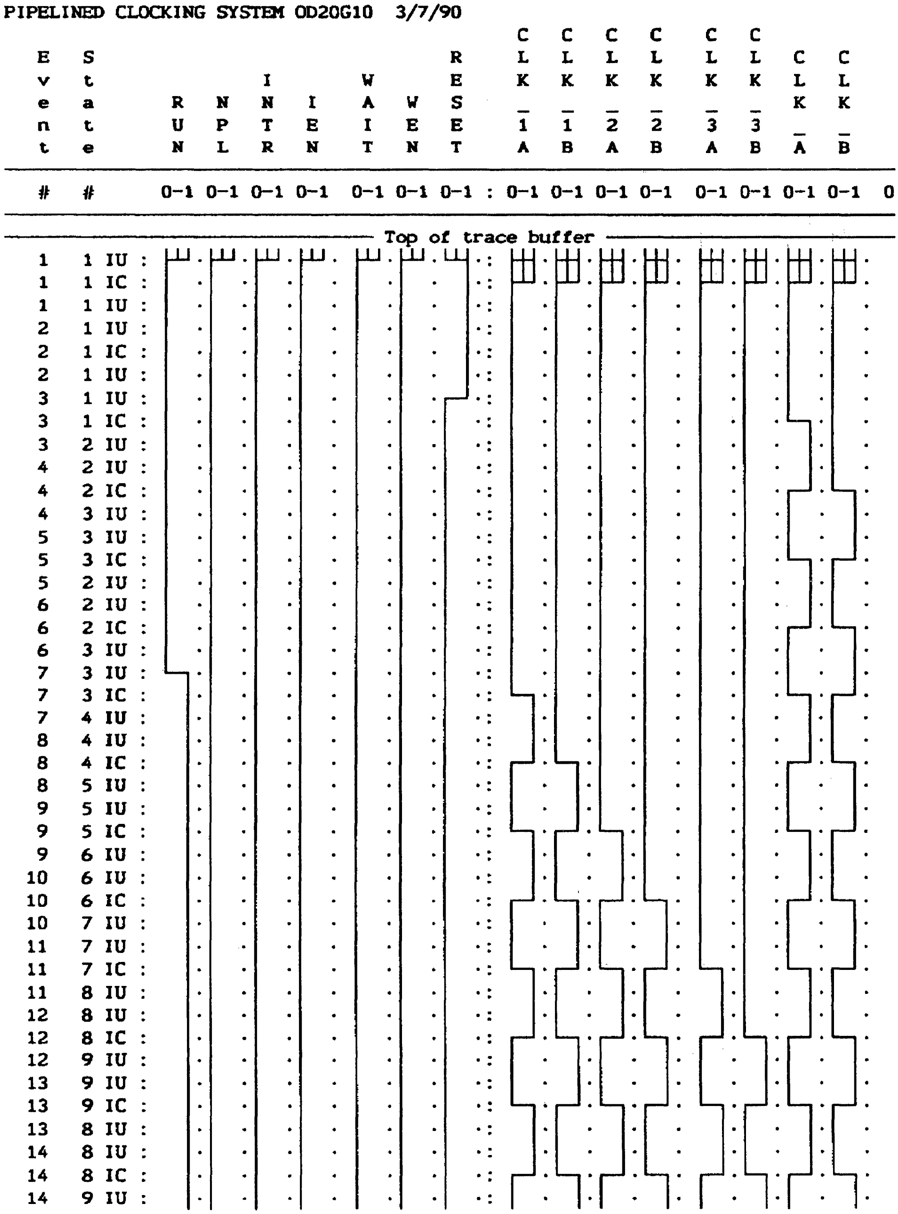
```
PIPELINED CLOCKING SYSTEM OD20G10
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45
```




```
-------------------------------------------------------
LOG/iC  -  PAL              CPU TIME USED:      45 SEC
```

Appendix B. LOG/iC Simulation: Clock State Machine

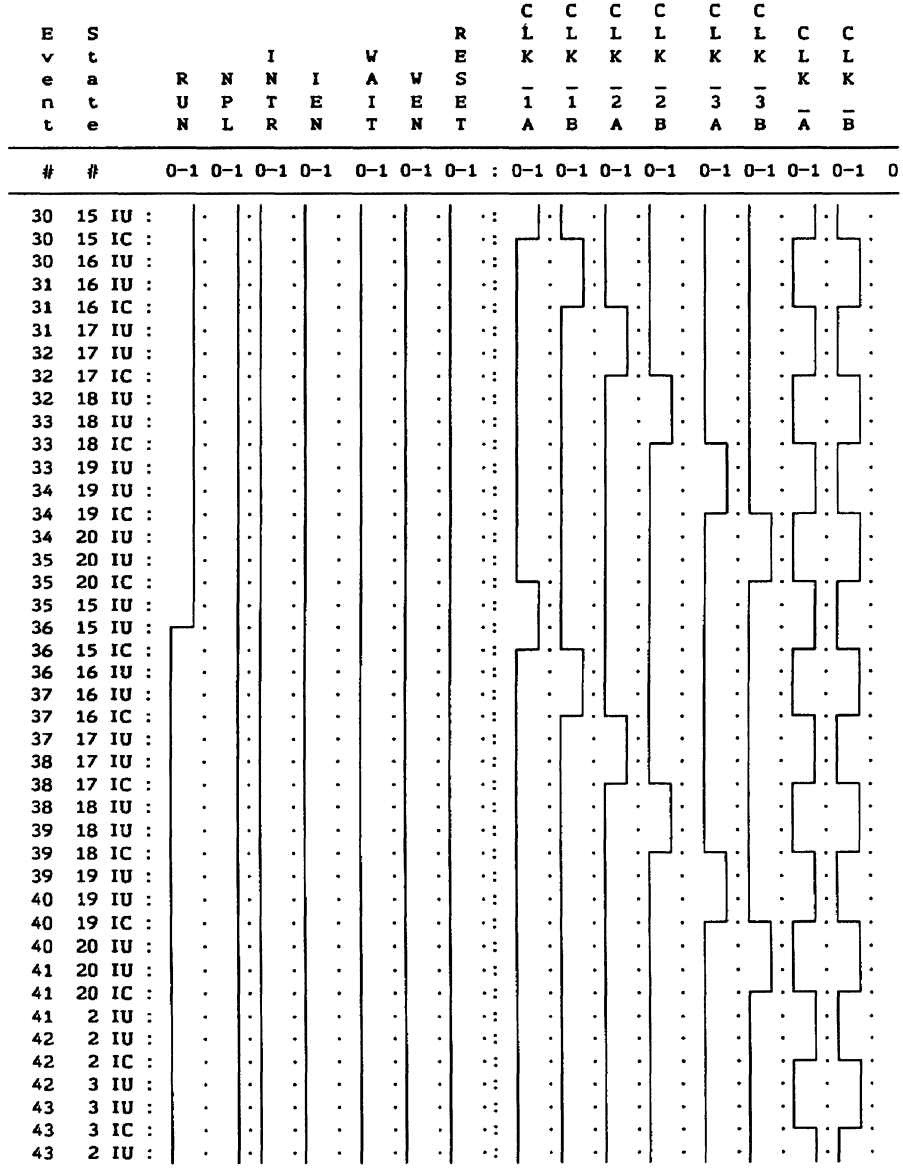
PIPELINED CLOCKING SYSTEM OD20G10 3/7/90



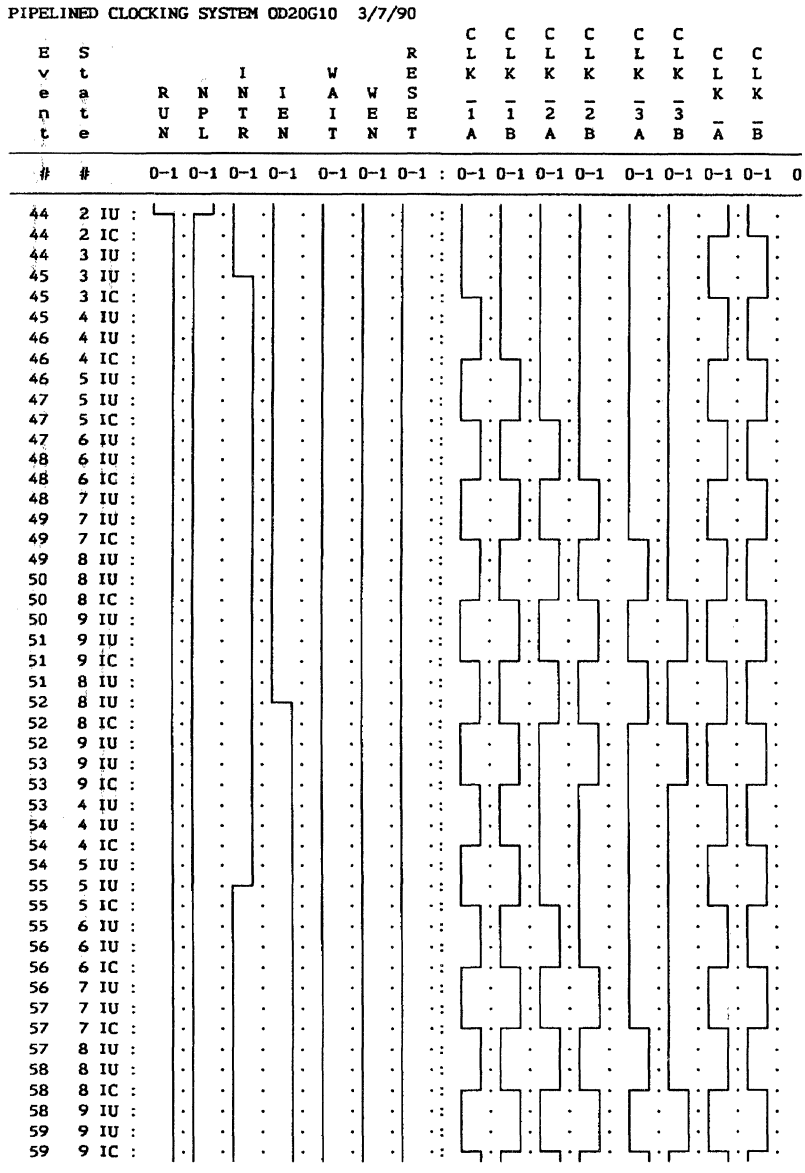
Appendix B. LOG/iC Simulation: Clock State Machine (continued)

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90

                                                                  C   C   C   C   C   C
E   S                                                     R       L   L   L   L   L   L   C   C
v   t                                                     E       K   K   K   K   K   K   L   L
e   a                 R   N   N   I   A   W   S           _       _   _   _   _   _   _   K   K
n   t                 U   P   T   E   I   E   E           1       1   2   2   3   3   _   _
t   e                 N   L   R   N   T   N   T           A       B   A   B   A   B   A   B
-------------------------------------------------------------------------------------------------
#   #                 0-1 0-1 0-1 0-1 0-1 0-1 0-1 :  0-1 0-1 0-1 0-1 0-1 0-1 0-1 0-1  0
-------------------------------------------------------------------------------------------------
15   9 IU :
15   9 IC :
15   8 IU :
16   8 IU :
16   8 IC :
16  10 IU :
17  10 IU :
17  10 IC :
17  11 IU :
18  11 IU :
18  11 IC :
18  12 IU :
19  12 IU :
19  12 IC :
19   2 IU :
20   2 IU :
20   2 IC :
20   3 IU :
21   3 IU :
21   3 IC :
21   2 IU :
22   2 IU :
22   2 IC :
22   3 IU :
23   3 IU :
23   3 IC :
23  15 IU :
24  15 IU :
24  15 IC :
24  16 IU :
25  16 IU :
25  16 IC :
25  17 IU :
26  17 IU :
26  17 IC :
26  18 IU :
27  18 IU :
27  18 IC :
27  19 IU :
28  19 IU :
28  19 IC :
28  20 IU :
29  20 IU :
29  20 IC :
29  15 IU :
```

Appendix B. LOG/iC Simulation: Clock State Machine (continued)



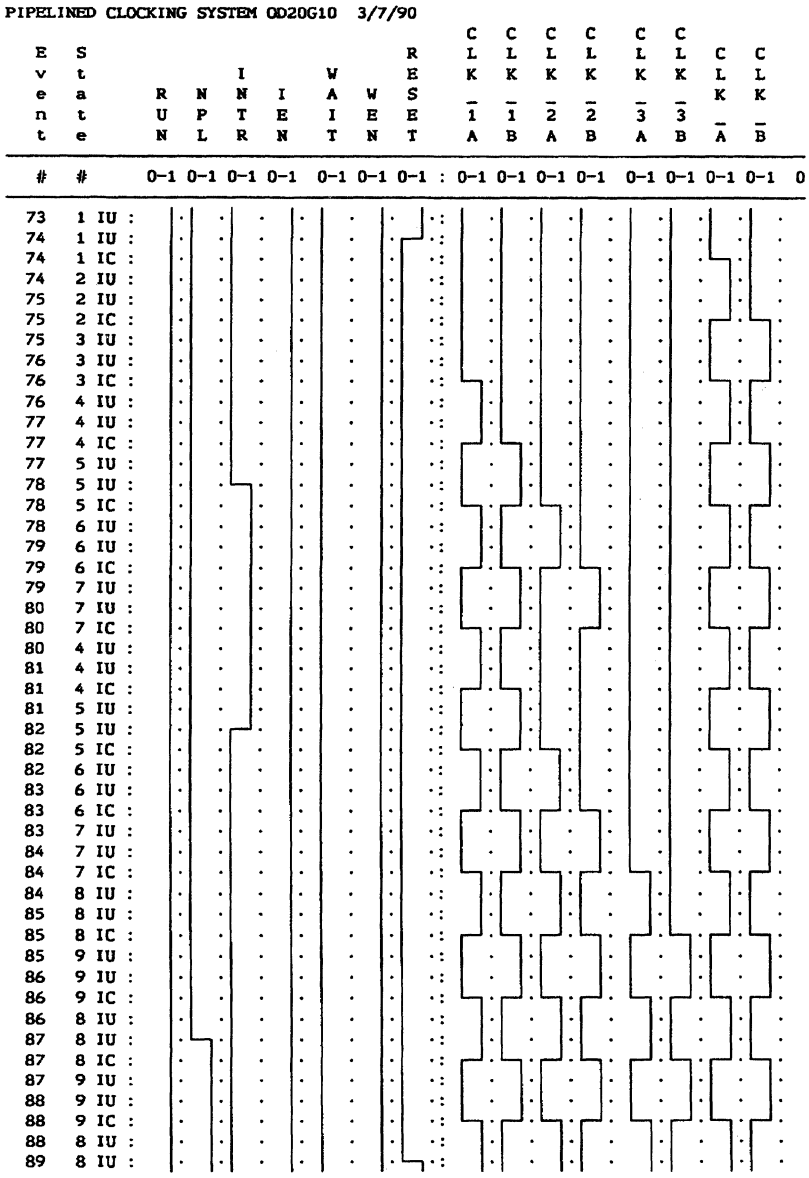
Appendix B. LOG/iC Simulation: Clock State Machine (continued)



Appendix B. LOG/iC Simulation: Clock State Machine (continued)

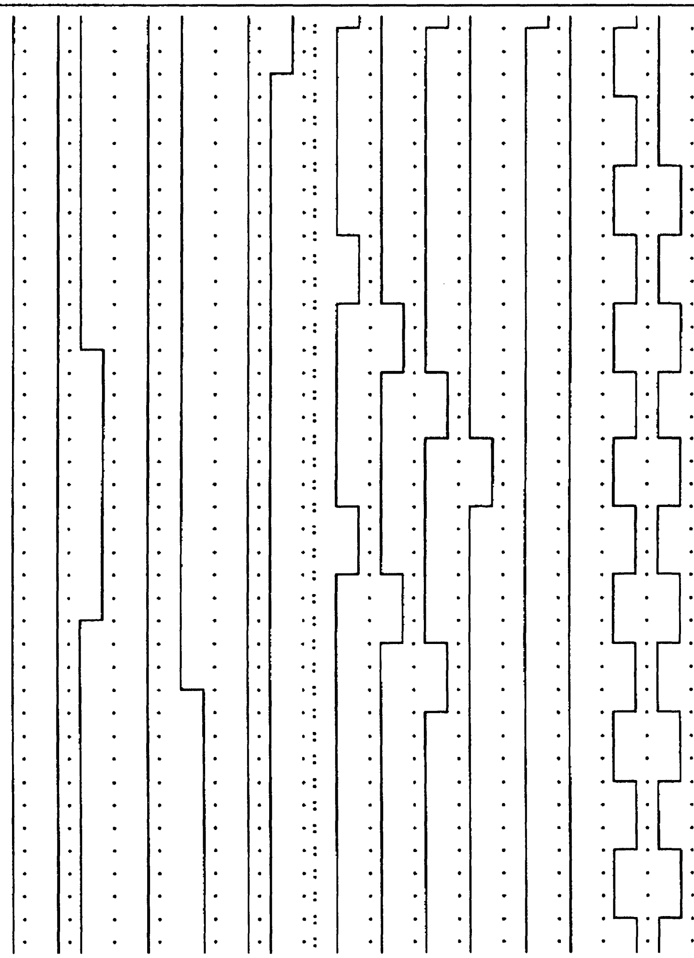
```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90

                                                                  C   C   C   C   C   C
 E   S                                                        R   L   L   L   L   L   L   C   C
 v   t                                    I           W       E   K   K   K   K   K   K   L   L
 e   a          R   N   N   I   A   W   S                       _   _   _   _   _   _   K   K
 n   t          U   P   T   E   I   E   E                       1   1   2   2   3   3   _   _
 t   e          N   L   R   N   T   N   T                       A   B   A   B   A   B   A   B

 #   #         0-1 0-1 0-1 0-1 0-1 0-1 0-1 : 0-1 0-1 0-1 0-1 0-1 0-1 0-1 0-1 0
 59  8 IU  :
 60  8 IU  :
 60  8 IC  :
 60  9 IU  :
 61  9 IU  :
 61  9 IC  :
 61  8 IU  :
 62  8 IU  :
 62  8 IC  :
 62 14 I   :
 63 14 I   :
 63 14 IC  :
 63 13 I   :
 64 13 I   :
 64 13 IC  :
 64 14 I   :
 65 14 I   :
 65 14 IC  :
 65 13 I   :
 66 13 I   :
 66 13 IC  :
 66 14 I   :
 67 14 I   :
 67 14 IC  :
 67  8 IU  :
 68  8 IU  :
 68  8 IC  :
 68  9 IU  :
 69  9 IU  :
 69  9 IC  :
 69  8 IU  :
 70  8 IU  :
 70  8 IC  :
 70  9 IU  :
 71  9 IU  :
 71  9 IC  :
 71  1 IU  :
 72  1 IU  :
 72  1 IC  :
 72  1 IU  :
 73  1 IU  :
 73  1 IC  :
```

Appendix B. LOG/iC Simulation: Clock State Machine (continued)



Appendix B. LOG/iC Simulation: Clock State Machine (continued)

```
PIPELINED CLOCKING SYSTEM OD20G10   3/7/90
                                                    C   C   C   C     C   C
 E   S                                      R       L   L   L   L     L   L   C   C
 v   t                I           W         E       K   K   K   K     K   K   L   L
 e   a        R   N   N   I       A   W     S       _   _   _   _     _   _   K   K
 n   t        U   P   T   E       I   E     E       1   1   2   2     3   3   _   _
 t   e        N   L   R   N       T   N     T       A   B   A   B     A   B   A   B
-----------------------------------------------------------------------------------
 #   #       0-1 0-1 0-1 0-1     0-1 0-1 0-1  :  0-1 0-1 0-1 0-1   0-1 0-1 0-1 0-1   0
-----------------------------------------------------------------------------------
 89    8 IC :
 89    1 IU :
 90    1 IU :
 90    1 IC :
 90    2 IU :
 91    2 IU :
 91    2 IC :
 91    3 IU :
 92    3 IU :
 92    3 IC :
 92   15 IU :
 93   15 IU :
 93   15 IC :
 93   16 IU :
 94   16 IU :
 94   16 IC :
 94   17 IU :
 95   17 IU :
 95   17 IC :
 95   18 IU :
 96   18 IU :
 96   18 IC :
 96   15 IU :
 97   15 IU :
 97   15 IC :
 97   16 IU :
 98   16 IU :
 98   16 IC :
 98   17 IU :
 99   17 IU :
 99   17 IC :
 99   22 I  :
100   22 I  :
100   22 IC :
100   21 I  :
101   21 I  :
101   21 IC :
101   22 I  :
102   22 I  :
102   22 IC :
102   21 I  :
```

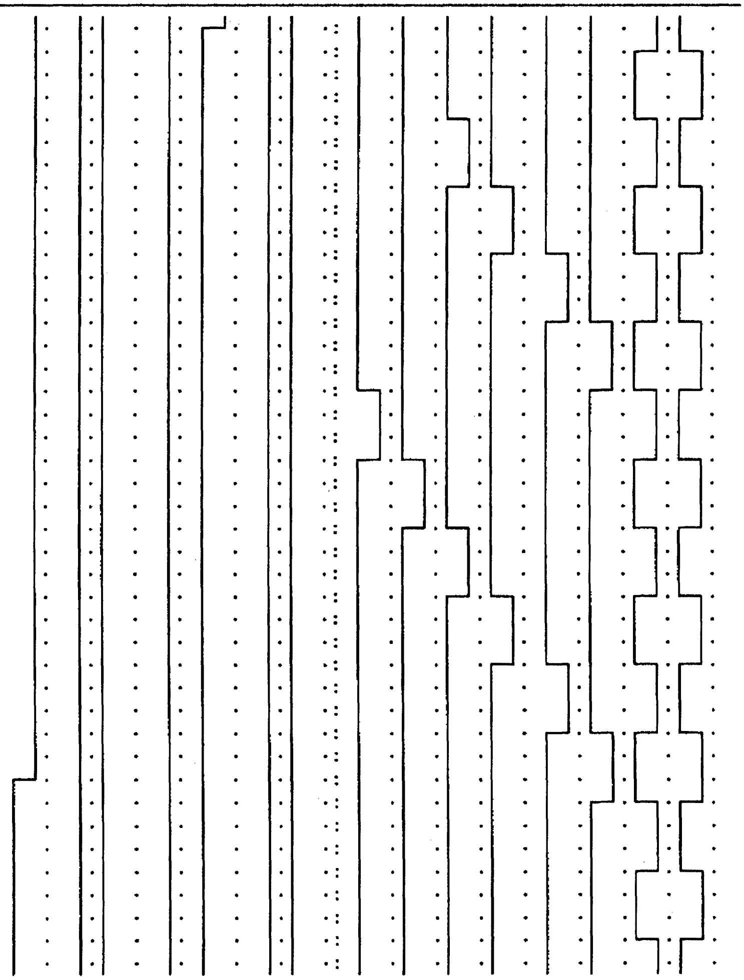


Appendix B. LOG/iC Simulation: Clock State Machine (continued)

PIPELINED CLOCKING SYSTEM OD20G10 3/7/90

Signals (0-1): RUN, NPL, INTR, IEN, WAIT, WEN, RESET : CLK_1A, CLK_1B, CLK_2A, CLK_2B, CLK_3A, CLK_3B, CLK_A, CLK_B

Event #	State #	
103	21	I
103	21	IC
103	22	I
104	22	I
104	22	IC
104	17	IU
105	17	IU
105	17	IC
105	18	IU
106	18	IU
106	18	IC
106	19	IU
107	19	IU
107	19	IC
107	20	IU
108	20	IU
108	20	IC
108	15	IU
109	15	IU
109	15	IC
109	16	IU
110	16	IU
110	16	IC
110	17	IU
111	17	IU
111	17	IC
111	18	IU
112	18	IU
112	18	IC
112	19	IU
113	19	IU
113	19	IC
113	20	IU
114	20	IU
114	20	IC
114	2	IU
115	2	IU
115	2	IC
115	3	IU
116	3	IU
116	3	IC
116	2	IU



LOG/iC is a trademark of Isdata Corporation.
PLD Toolkit is a trademark of Cypress Semiconductor Corporation.

Using Hierarchical VHDL Design

Introduction

Hierarchical design methodology has been commonly used for quite some time by system designers and software developers. There are two primary advantages to using this methodology. First, it allows commonly-used building blocks to be created separately and saved for later use without having to redesign or reverify them. Second, it allows for more readable design files by keeping the top-level design file as a simple integration of smaller building blocks, either user-defined or from a vendor-supplied library. In system design, these building blocks normally take the form of schematic symbols instantiated into a schematic drawing, while in software they are functions or procedures that are called from the main program.

Warp2™ and *Warp3*™ VHDL includes a set of features specifically designed to make hierarchical design both simple and powerful. This note will first describe these features and then walk through a simple example of how they might be used. It assumes that the reader has read the *Warp2* User's Manual and is familiar with how to create a VHDL design unit consisting of an entity-architecture pair.

Key Concepts

In order to construct a hierarchical design in VHDL, the designer must understand the concepts of components, packages and libraries.

Component – A component is a VHDL design unit that may be instantiated in other VHDL design units. Before it can be instantiated, it must be declared using the COMPONENT declaration which specifies the name of the component and lists its local signal names.

Package – A package is a collection of VHDL declarations that can be used by other VHDL descriptions. For the purpose of creating hierarchical designs, a package consists of one or more components. However, a package may also include other types of declarations.

Library – A library is a logical storage facility for design units. Before a component can be instantiated in a higher-level design unit, its package must be compiled into a library that is visible to that design unit, usually the current work library.

Simple Example

Consider the following example. A designer discovers that for a specific type of circuit design he commonly needs an unusual type of counter. ("Commonly," in this reference, means that this counter is likely to be used either multiple times in a particular design or across multiple designs. Both are cases where hierarchical design simplifies things.) This counter is a simple four-bit counter, but it must output a terminal count indication (tc) and roll over to zero when it reaches 1110 rather than 1111.

A design file that would accomplish this is shown in Appendix A. (The reader should understand the contents of the entity-architecture pair—they will not be discussed further.) In order to use this counter in other VHDL design units, it is declared as a component within a package at the top of the file. The component declaration simply names the design unit and lists its signal names. When this file is compiled, the package is placed into the current library and the component it contains may then be instantiated into other designs compiled into that library. If this were a standalone design, the entire package declaration could be omitted.

Now, suppose this design consists of two of these counters with their outputs multiplexed as in *Figure 1*. We can then instantiate our counter twice as in the design in Appendix B. All that is necessary is the statement:

```
use work.cnt_pkg.all;
```

at the top of the file, which makes any components in the cnt_pkg visible within the current design unit, as long as the package was compiled into the work library. The counters are then instantiated by giving them unique labels and listing the signals connected to the port map in the same order as the component declaration.

We could also have created the mux as a separate component and instantiated it, but it is simpler to use the if-then-else structure.

Multiple Components

For further illustration, assume our complete design includes two types of counters, one that rolls over at 1110 and one that rolls over at 1011, as shown in *Figure 2*. We could simply repeat the above procedure and create another design file with another component and package and then use both of these packages in our top-level design file.

However, it may be easier to keep track of things if we keep similar counter designs together in a single package as in Appendix C. This file contains both entity-architecture pairs and two components in a single package. As before, when this file is compiled, the package is added to the current library and its components are made visible with a single-use clause as in Appendix D.

Configurable Components using Generics

When multiple components that have the same basic architecture but differ in one or more parameters are needed (such as the two counters in the previous example) VHDL generics allow a more compact approach. Generics are a means by which parameters may be passed to a component when it is instantiated allowing a configurable component.

In Appendix E a component is created that is the same basic counter, but allows the terminal count to be configured using a generic. Instead of hard-cod-

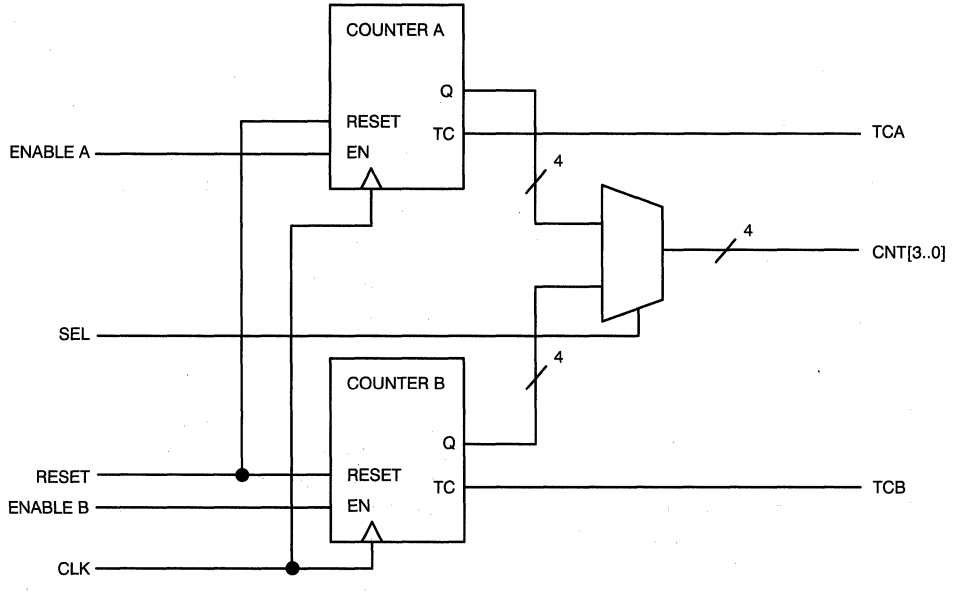


Figure 1. Multiplexed Dual Counter Design

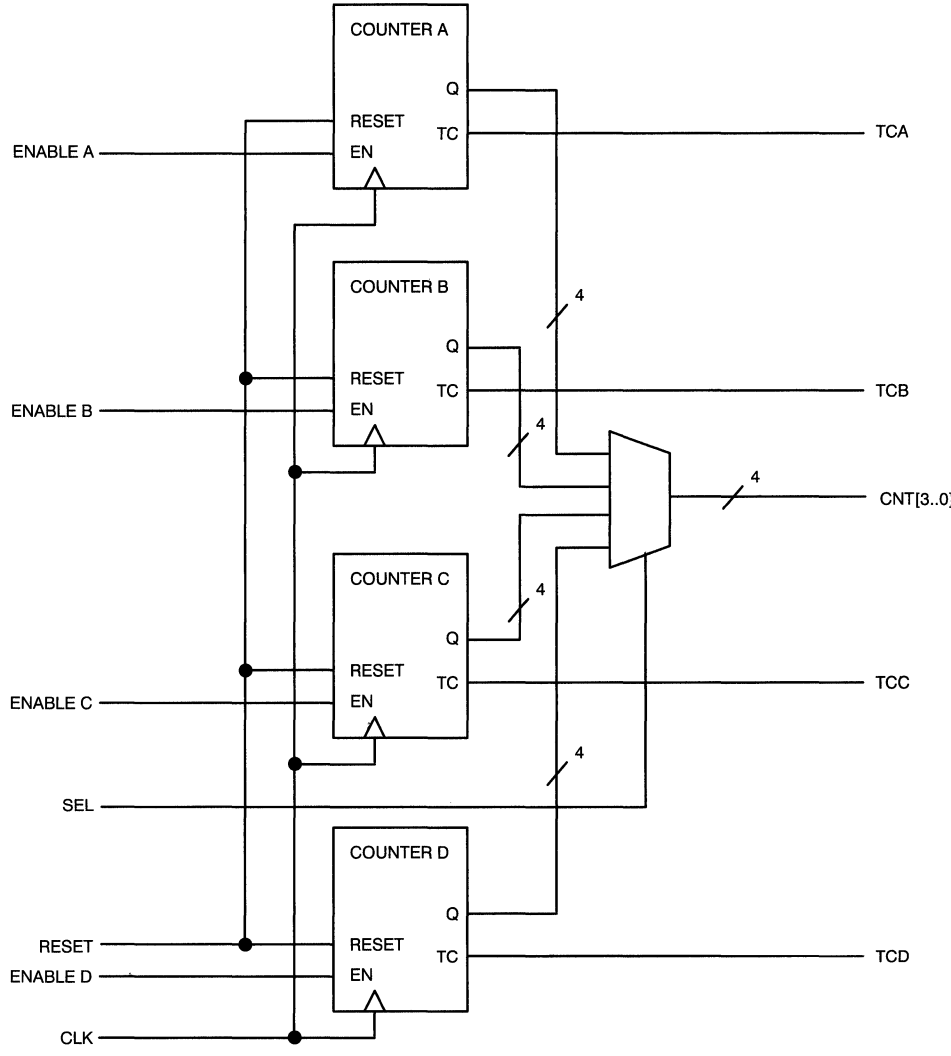


Figure 2. Multiplexed Quad Counter Design

ing this value, a bit_vector is used in the architecture. This bit_vector is then declared in the entity and component declarations. Generics may also be of other types such as integers and a component may contain multiple generics (although our example contains only one).

Appendix F is the top-level design unit of the same design from *Figure 2*, but this time it is using the component with the generic rather than two different components. When the component is instantiated, it is configured by passing it the specific bit_vector in the generic map.

Warp2 and *Warp3* are trademarks of Cypress Semiconductor Corporation.

Appendix A. Counter with Terminal Count and Rollover Selection

```
use work.cypress.all;
use work.rtlpkg.all;

package cnt_pkg is
  component count15 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bv_math.all;

entity count15 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count15;

architecture one of count15 is
begin

process begin
if cnt="1110" then
    tc<='1';
else
    tc<='0';
end if;

end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1110" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=inc_bv(cnt);
        else
            cnt<=cnt;
        end if;
    end if;

end process;

end one;
```

Appendix B. Instantiation of Counter from Appendix A

```
use work.cypress.all;
use work.rtlpkg.all;
use work.cnt_pkg.all;

entity muxcntr is port(
     clk, enablea, enableb, reset, sel:in bit;
     cnt:out bit_vector (3 downto 0);
     tca, tcb:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb:bit_vector(3 downto 0);

begin

cntra:count15 port map(clk, enablea, reset, muxina, tca);
cntrb:count15 port map(clk, enableb, reset, muxinb, tcb);

process begin
if sel='1' then
     cnt<=muxina;
else
     cnt<=muxinb;
end if;

end process;
end one;
```

Appendix C. Multiple Counters in a Single Package

```
use work.cypress.all;
use work.rtlpkg.all;

package cnt_pkg is
  component count15 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
  component count12 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bv_math.all;

entity count15 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count15;

architecture one of count15 is
begin

process begin
if cnt="1110" then
    tc<='1';
else
    tc<='0';
end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1110" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=inc_bv(cnt);
        else
            cnt<=cnt;
        end if;
    end if;
```

Appendix C. Multiple Counters in a Single Package (continued)

```
end process;
end one;

use work.bv_math.all;

entity count12 is port(
     clk, enable, reset:in bit;
     cnt:inout bit_vector (3 downto 0);
     tc:out bit);
end count12;

architecture one of count12 is
begin

process begin
if cnt="1011" then
     tc<='1';
else
     tc<='0';
end if;
end process;

process(clk,reset) begin
     if reset='1' then
          cnt<="0000";
     elsif (clk'event and clk='1') then
          if cnt="1011" and enable='1' then
               cnt<="0000";
          elsif enable='1' then
               cnt<=inc_bv(cnt);
          else
               cnt<=cnt;
          end if;
     end if;

end process;
end one;
```

Appendix D. Instantiation of Counters in Appendix C

```
use work.cypress.all;
use work.rtlpkg.all;
use work.cnt_pkg.all;

entity muxcntr is port(
     clk, enablea, enableb, enablec, enabled, reset:in bit;
     sel:in bit_vector (1 downto 0);
     cnt:out bit_vector (3 downto 0);
     tca, tcb, tcc, tcd:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb, muxinc, muxind:bit_vector(3 downto 0);

begin

cntra:count15 port map(clk, enablea, reset, muxina, tca);
cntrb:count15 port map(clk, enableb, reset, muxinb, tcb);
cntrc:count12 port map(clk, enablec, reset, muxinc, tcc);
cntrd:count12 port map(clk, enabled, reset, muxind, tcd);

process begin
if sel="11" then
     cnt<=muxina;
elsif sel="10" then
     cnt<=muxinb;
elsif sel="01" then
     cnt<=muxinc;
else
     cnt<=muxind;
end if;

end process;
end one;
```

Appendix E. Parametrizable Counters Using Generics

```
use work.cypress.all;
use work.rtlpkg.all;

package cnt_pkg is
  component countg
  generic (stop:bit_vector(3 downto 0):="1111");
    port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bv_math.all;

entity countg is
  generic (stop:bit_vector(3 downto 0):="1111");
    port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end countg;

architecture one of countg is
begin

process begin
if cnt=stop then
    tc<='1';
else
    tc<='0';
end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt=stop and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=inc_bv(cnt);
        else
            cnt<=cnt;
        end if;
    end if;

end process;
end one;
```

Appendix F. Multiplexed Quad Counter Design

```
use work.cypress.all;
use work.rtlpkg.all;
use work.cnt_pkg.all;

entity muxcntr is port(
     clk, enablea, enableb, enablec, enabled, reset:in bit;
     sel:in bit_vector (1 downto 0);
     cnt:out bit_vector (3 downto 0);
     tca, tcb, tcc, tcd:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb, muxinc, muxind:bit_vector(3 downto 0);

begin
cntra:countg generic map("1110") port map(clk, enablea, reset, muxina, tca);
cntrb:countg generic map("1110") port map(clk, enableb, reset, muxinb, tcb);
cntrc:countg generic map("1011") port map(clk, enablec, reset, muxinc, tcc);
cntrd:countg generic map("1011") port map(clk, enabled, reset, muxind, tcd);

process begin
if sel="11" then
     cnt<=muxina;
elsif sel="10" then
     cnt<=muxinb;
elsif sel="01" then
     cnt<=muxinc;
else
     cnt<=muxind;
end if;

end process;
end one;
```

Designing UltraLogic™ With Exemplar and Synopsys™

Introduction

Galileo™ from Exemplar Logic and the Design Compiler from Synopsys™ provide two pathways for programmer logic users to use Cypress's UltraLogic™ devices with third-party design environments. They provide behavioral Hardware Description Language (HDL) synthesis through the support of a wide variety of HDL design entry formats and powerful constraint-driven synthesis and optimization capabilities. Both of these tools integrate tightly with Cypress's *Warp*™ design tool to complete the design flow when targeting UltraLogic devices.

This application note is intended to familiarize the reader with these two third-party design tools, as well as the Cypress-specific design pathway by covering the following topics:

- Design entry formats
- UltraLogic device support
- Software Requirements
- Design flow and integration with *Warp*
- Design Synthesis and Optimization Capabilities

EXEMPLAR LOGIC – GALILEO

Galileo consists of three separate modules—the Logic Explorer (the synthesis engine), the Time Explorer (the timing analysis engine), and the V-System (the simulation engine). We will focus mainly on the capabilities of the Logic Explorer and its integration with *Warp*™.

Design Entry Formats

The Logic Explorer provides powerful behavioral synthesis by supporting a wide variety of design entry formats:

- VHDL (IEEE 1164 & 1076)
- Verilog™
- Palasm 2™
- OpenABEL™

Various formats of netlist are also supported for design retargeting and conversion:

- EDIF 2 0 0
- Berkeley PLA
- Actel ADL
- Xilinx XNF

The following design entry format is also provided to facilitate the integration of multiple designs in different formats:

- Exemplar Logic Integration Language (EIL)

UltraLogic Device Support

Logic Explorer currently supports the following family of programmable logic devices from Cypress:

- MAX340® EPLDs
- FLASH370™ CPLDs
- pASIC380™ FPGAs

Software Requirements

To design with the MAX340 and FLASH370 devices, *Warp2* alone is sufficient.

To design with the pASIC380 devices, *Warp2+* is required as a minimum.

Design Flow and Integration with *Warp*

The Logic Explorer–*Warp* design flow includes design entry, synthesis and optimization, fitting (for MAX340 and FLASH370) or place & route (for pASIC380), simulation, and programming (see *Figure 3*). Designs in design entry formats supported by Exemplar can be entered using any text editor, which then goes through the Logic Explorer for synthesis and optimization. The output from the Logic Explorer then goes into *Warp* for fitting or place & route, and programming files and/or timing models are generated by *Warp* for device simulation and programming.

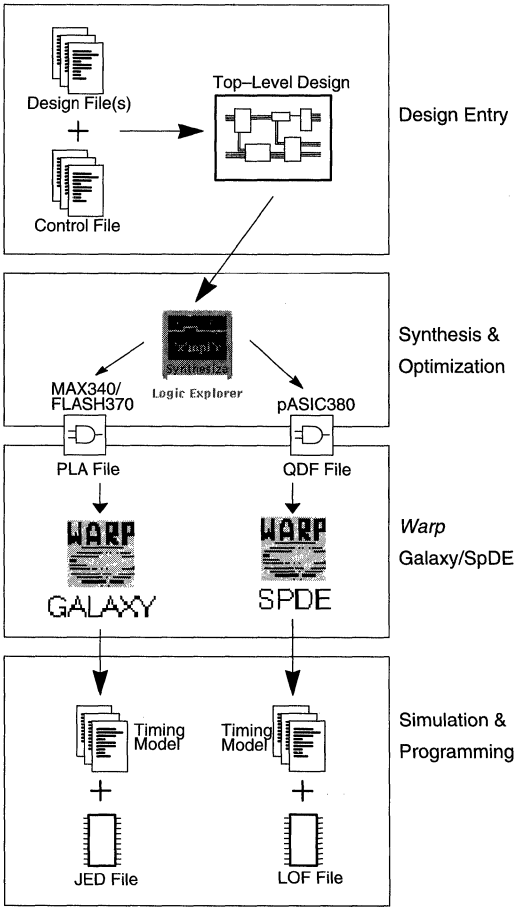


Figure 3. Logic Explorer–*Warp2* Design Flow

Details about each of the design stages are described below:

(A) Design Entry

Designs (in description languages, netlist, or EIL) are entered using any text editor and saved as ASCII text files. Hierarchical designs can be described across multiple design files. The Exemplar Logic Integration Language (EIL) can also be used to link multiple design files in different entry formats into one large design.

An optional control file can be included to specify design-specific parameters such as defining I/O pad mappings and timing requirements. The control file should have the same name as the design file with a .ctr extension.

(B) Design Optimization and Synthesis using the Logic Explorer

The next step is to synthesize and optimize the design(s) using the Logic Explorer.

The Logic Explorer main window allows the user to specify design-specific information like input and output filenames, source and target technology, and entry format, as well as synthesis and runtime options (for details refer to Design Environment below). Depending on the target technology and user-specified constraints, a number of optimization passes (ranging from one to eleven) will be run. The results of these passes will be plotted on an Area vs. Delay graph. The user can save the pass that best fulfills the user-specified constraints.

(C) Device Fitting or Place & Route using *Warp*

After synthesis and optimization, the results generated by the Logic Explorer will be used in *Warp* for fitting or place & route.

Depending on the target technology, results from the Logic Explorer will be saved in the following output formats for interfacing with *Warp:*

Table 1. Logic Explorer to *Warp* Output Formats

Target Technology	Output Format
MAX340	PLA
FLASH370	PLA
pASIC380	QDIF

For MAX340 and FLASH370 devices, a PLA file will be generated by the Logic Explorer which will be taken by the *Warp* Galaxy fitter as input to perform device fitting. A JEDEC file will be produced for device programming and timing models will be generated for device simulation.

For pASIC380 devices, a QDIF file will be generated by the Logic Explorer which will be taken by the *Warp* SpDE place & route tool as input to perform place & route and timing analysis. A LOF file will be generated for device programming and timing models will be generated for device simulation.

Design Synthesis and Optimization Capabilities

We will now highlight some of the features offered by the Logic Explorer. We will begin by summarizing how to access these features by describing its user interface and options in Design Environment. We will then move on to describe these features as categorized by Design Synthesis and Optimization capabilities, Design Integration, and Command and Control File creation.

(A) Design Environment

The Logic Explorer main window has a simple, user-friendly graphical user interface. It consists of a main menu where the user can specify Input and Output Filenames, Source and Target Technology, and Entry Format. In addition, four submenus are available to set Synthesis and Runtime options:

(1) Input Options

- Lower-level design filename(s) for hierarchical designs
- VHDL style (IEEE 1164, 1076, or View-Logic)
- State machine encoding style (binary, gray, random, and one-hot)
- Module generation library names

(2) Output Options

- Target device
- Target package

(3) Synthesis Options

- Optimization constraints (specific area and/or delay)
- Number of optimization passes to be performed
- Derating factors for delay calculations
- Retarget switches for remapping to different technologies
- Command and additional source library filename(s)
- Report file options

(4) Runtime Options

- Global optimization goal (area or delay)
- Technology mapping effort level

Most of these options are self-explanatory. Some of them will be explained in further detail below.

(B) Design Optimization

The Logic Explorer allows users to exert control over the synthesis of their designs by providing ample features in the following areas:

(1) Global and Local Optimization

Global optimization refers to the synthesis of a design that has multiple building blocks as a whole, while local optimization refers to the synthesis of a design's individual building block before combining them together.

Global optimization can be accessed through the Synthesis and Runtime Options menus in the Logic Explorer main window. Optimization is applied to the design overall, including all lower-level modules.

Local optimization trades off control over global optimization with overall design results. The flexibility of being able to optimize each of a design's lower-level building block locally before linking them together can be achieved by using the Exemplar Logic Integration Language (EIL).

EIL is a simple language that describes a netlist of instances of building blocks, each of which can be written in any input format, and can be optimized using different constraints. EIL allows the user to specify the I/O interface of the top-level design and the interconnection of instances which make up the design.

(2) Constraint-Driven Optimization

By setting optimization constraints, the user can specify different design requirements which will affect the synthesis outcome. The Logic Explorer will try its best to synthesis and optimize in such a way that all constraints are met. Design constraints can be applied to an overall design or to individual signals:

- Area
- Delay
- Max Fan-in (for MAX340 only)
- Max PT (for MAX340 only)
- Max Load (for pASIC380 only)

All of the above constraints can be specified as command line options or placed in the control file (see Command and Control File below). Area and delay constraints can be accessed through the Synthesis and Runtime Options menus in the Logic Explorer main window. The user can either let the Logic Explorer synthesize to the best area or delay that it can achieve (Runtime Options), or set specific constraints by specifying values for the maximum area and/or the delay allowed (Synthesis Options).

(3) Technology Mapping

For technology-independent design entry (e.g., VHDL), the Logic Explorer will first translate the design into their internal technology-independent Logic Data Structures. Architecture-specific logic optimization will then begin. When this is done, the design will be mapped into gates that are available from the target technology library. Multiple passes of this technology-mapping step can be run (using different strategies) to achieve results that will best fulfill the design constraints set by the user.

For technology-dependent design entry (e.g., netlist), the source technology library also needs to be specified. The technology mapper will then perform device-specific transformations to map gates from the source technology to the target technology. Some of these retargeting switches, like the mapping of internal three-states into combinatorial logic, can be accessed through the Synthesis Options menu.

(4) I/O Mapping

Automatic synthesis of I/O pads is part of the Logic Explorer's default mode. Howev-

er, the user can also assign pads manually to have better control over pad assignments. Manual pad assignments can be done either through component instantiations in the design input files (e.g., in VHDL or Verilog), or through the use of the control file (see Command and Control File below).

Pin assignments can also be done through the control file.

(C) Design Integration with EIL

The Exemplar Logic Integration Language (EIL) can also be used to link multiple design files in different entry formats (e.g., mixing HDLs and netlists) into one large design.

EIL is a simple language that essentially describes a netlist of instances of building blocks each of which can be written in any input format and can be optimized using different constraints. EIL allows the user to specify the I/O interface of the top-level design and the interconnection of instances which make up the design.

(D) Command and Control File

Command files can be used to store command line options that the user will want to reuse. Any Logic Explorer command line options (e.g., Input and Output filenames, target and source technology libraries, etc.) can be saved in a command file. The user can specify any command file to be reused in subsequent runs of the Logic Explorer. A Command File Editor is available from the File menu in the Logic Explorer main window.

Control files can be used to store design constraints, manual pad assignments, and pin assignments for a specific design. Any control file can be specified to be used with a specific design by specifying it's name in the Control File menu in the Logic Explorer main window.

Please refer to the Galileo Reference Manual for specific formats of the command and control file. However, here is a summary of some useful command file *(Table 2)* and control file options *(Table 3):*

Table 2. Useful Command File Options

Function	Command File Option
Optimize for Area	-area
Optimize for Delay	-delay
Design Constraint for Max Area	-maxarea=<n>
Design Constraint for Max Delay	-maxdelay=<n>
Design Constraint for Max Fan-in	-max_fanin=<n>
Design Constraint for Max PT	-max_pt=<n>
Design Constraint for Max Load	-maxload=<load>
Control File Name	-control=<name>
FSM Encoding Style	-encoding=<encoding style>
Package Type	-package=<name>
Part Name	-part=<part_number>
Source Library Name	-source=<library_name>
Target Library Name	-target=<library_name>

Table 3. Useful Control File Options

Function	Control File Option
Design Constraint for Max. Load	MAX LOAD
Signal Name Preservation	PRESERVE SIGNAL
Manual Pad Assignment	PAD or GATE
Pin Assignment	SET...PIN NUMBER

SYNOPSYS – DESIGN COMPILER

Like the Logic Explorer, the Design Compiler from Synopsys also aims to provide powerful synthesis through the support of a variety of behavioral HDLs, as well as some netlist support for design entry formats. It is also tightly integrated with the *Warp* design tool to provide a seamless design pathway for designing with UltraLogic devices.

Design Entry Formats

Hardware Description Language (HDL) support for the Design Compiler is as follows:

- VHDL (IEEE 1164 & 1076)
- Verilog

Netlist support is as follows:

- Berkeley PLA
- EDIF 2 0 0

UltraLogic Device Support

The Design Compiler currently supports pASIC FPGAs.

Software Requirements

To design with the pASIC380 devices, *Warp2+* is required as a minimum.

Design Flow and Integration with *Warp*

The Design Compiler-*Warp* design flow includes design entry, synthesis and optimization, place & route (for pASIC380), simulation, and programming (see *Figure 2*). Designs in HDL and netlists can be entered using any text editor, which then goes through the Design Compiler for synthesis and optimization. The output from the Design Compiler then goes into *Warp* for place & route, and programming files and/or timing models are generated by *Warp* for device simulation and programming.

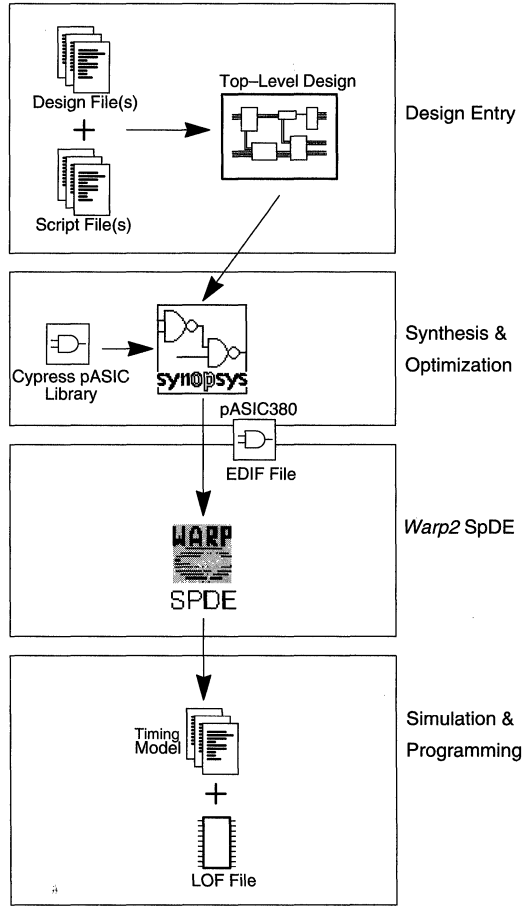


Figure 4. Design Compiler-*Warp* Design Flow

Details about each of the design stages are described below:

(A) Design Entry

Designs (in HDL or netlist) are entered using any text editor and saved as ASCII text files. Hierarchical designs can be described across multiple design files.

Optional Design Compiler Shell Script files can be included to specify synthesis commands as well as design-specific parameters such as input and output filenames, source and target libraries, design constraints, I/O pad mappings, and pin assignments.

(B) Design Optimization and Synthesis using the Design Compiler

The next step is to synthesize and optimize the design(s) using the Design Compiler.

The Design Compiler's graphical interface is called the Design Analyzer. Its main window allows the user to specify design-specific information like input and output filenames, source and target technology, and entry format, as well as synthesis and design constraint options (for details refer to Design Environment below). Upon completion of synthesis and optimization, the resulting netlist will be displayed in graphical form in the Design Analyzer. Users can then push in and out of design hierarchies, examine the timing of critical nets, and generate report files.

(C) Device Place & Route using *Warp*

After synthesis and optimization, the results are generated by *Warp* for place & route. The output format for interfacing with *Warp* is EDIF 2 0 0.

For pASIC380 devices, an EDIF file (containing pASIC primitives will be generated by the Design Compiler which will be taken by the *Warp* SpDE place & route tool as input to perform place & route and timing analysis. A LOF file will be generated for device programming and timing models will be generated for device simulation.

Design Synthesis and Optimization Capabilities

We will now highlight some of the features offered by the Design Compiler. We will begin by summarizing how to access these features by describing its user interface and options in Design Environment. We will then move on to describe these features as categorized by Design Synthesis and Optimization capabilities, Design Integration, and DC Shell Script creation.

(A) Design Environment

The Design Analyzer is a graphical user interface that consists of pull-down menus where the user can specify input and output filenames, source and target libraries, design constraints, I/O pad mappings, and pin assignments. It is also a hierarchical netlist viewer that allows users to view the design in terms of functional blocks before synthesis and in mapped gates after technology mapping. The user can also interactively examine the timing of the critical nets.

The user can open a Command Window to enter Design Compiler commands interactively in command line form. Options that are available from the pull-down menus have an equivalent command line format.

The user can also execute commands in batch form by using DC Shell Scripts. Please refer to the section on DC Shell Scripts for further details.

Some useful options that are available from the pull-down menus are summarized below:

(B) Design Synthesis and Optimization

Synthesis in the Design Compiler involves the translation of an HDL design into a Synopsys built-in generic logic representation and the op-

timization and mapping of that representation using the Cypress pASIC library elements.

As in the Logic Explorer, various synthesis and optimization features are available to the user to better control the results of synthesis.

(1) Constraint-Driven Optimization

Users can control the synthesis outcome by setting optimization constraints on individual signals, on modules under any level of the design hierarchy, or on the overall design. The Design Compiler will try its best to synthesize and optimize in such a way so that all constraints are met. Design constraints that are available to the user for pASIC380 devices are:

- Area
- Delay
- Fanout

All the above constraints can be specified graphically from the Design Analyzer or placed in the DC shell script (see DC Shell Script below and Appendix D). For example, an adder that is constrained by area will be synthesized using a ripple-carry algorithm, while one that is constrained by speed will be synthesized using a carry-lookahead algorithm.

(2) FSM Extraction

Designs that include descriptions of finite state machines (FSMs) can be extracted into a State Table format. Once extracted into this format, the Design Compiler can perform the following FSM optimization techniques on the extracted design(s):

- Automatic state assignments, or completion of partial assignments,
- Optimization of the FSM(s) for Area or Delay,
- Optimization of "don't care" sets,
- Removal of redundant states, and
- Allows users to explore alternative FSM implementations with different state-encoding schemes (e.g., sequential, one-hot, gray, or manual).

For details on how to extract and optimize FSMs refer to Design Examples and Appendix E.

(3) Synthetic Cells

Arithmetic or relational operators are inferred from HDL descriptions as individual logic blocks to allow for more specific and optimal synthesis for these modules. For example, in the following VHDL code fragment:

```
ADD8 <= A8 + B8;

SIX  <= '1' when (ADD8 >
        "00000110") else '0';
```

The '+' sign in the first statement and the '>' sign in the second one will be inferred as an adder and a comparator respectively. These modules are referred to as synthetic cells, and will be synthesized according to design constraints that are set on them by the user (if any).

(4) Resource Sharing

Resource sharing is the using of a single hardware resource for multiple operations. In the following VHDL code fragment:

```
Z <= A + B when X else C + D;
```

Instead of inferring two synthetic adder cells due to two occurrences of the '+' operator, a single synthetic adder cell will be inferred, with the inputs A and C passing through one

two-to-one multiplexer, and inputs B and D passing through another one. In this way, additional logic for generating an extra adder is avoided. This is made possible because depending on the condition of 'X', either A and B or C and D uses the adder exclusively. And hence the resource (adder) can be shared. Other arithmetic and relational operators can be shared in the same fashion.

Resources are automatically shared during design compilation (and can be overridden) and are constraint-driven.

(C) Design Hierarchy

Designs with multiple levels of hierarchy can be viewed, manipulated, and synthesized using the Design Analyzer. Users can select signal paths or logic modules and set constraints on them, or push into lower hierarchical levels to view their gate-level implementations.

In addition, users can manipulate hierarchical designs using the following commands:

(1) Uniquify:

Each instance of the same cell (e.g., an 8-bit adder) is set to be unique (not referenced) so that each instance can be optimized individually through different constraints.

(2) Set Don't Touch:

Lower level modules specified with the set_dont_touch attribute will not be optimized or recompiled.

(3) Ungroup:

Hierarchical designs can be ungrouped or flattened into one single level before compilation and synthesis.

(D) DC Shell Script

DC shell scripts can be specified when invoking the Design Analyzer to perform design compilation and synthesis in batch mode. Any command that are accessible from the Design Analyzer's graphical menus has a command line equivalent that can be used from with a DC shell script. Shell scripts allows users to re-use part or all of the commands that make up the compilation and synthesis procedures.

Specialty Memories – 5

Specialty Memories Section Contents and Abstracts

This application note reviews the history of multi-port memories and explains the operation of Cypress's Dual-Port RAMs. Features discussed range from basic dual-port fundamentals to more advanced issues (like the "deadly embrace," for example) and ends with a design example. This application note is intended for designers of all experience levels and addresses most of the common issues that arise when using dual-port RAMs.

This application note explains the operation, architecture, and design considerations of Cypress's CY7C42X, 7C43X, and 7C46X families of FIFOs. These FIFOs feature industry-standard operation and pinout and are available in depths up to 32K x 9. Basic logic and timing operations such as reading and writing to the FIFO memory array, are covered in detail. Timing waveforms are included to help illustrate these operations. Common FIFO configurations such as Standalone Mode, Depth Expansion and Width Expansion are explained in detail. These sections explain how to properly use the flags in these modes and cover the operation of the expansion-in/expansion-out (XI/XO) pins. The final sections of this application note cover common problems and solutions encountered when using large FIFOs. These common problems include corrupted or repetitive data, missing or disappearing data, and FIFO lock up. Boundary flag operation is also discussed in relation to these problems. The final section covers V_{CC} noise related failures and recommends specific power bypassing techniques.

This application note explains the basic operations and features of Cypress Clocked FIFOs (CY7C44X and CY7C45X Clocked FIFOs). The first few sections explain the Clocked FIFO architecture in detail. Reading and writing to the FIFO memory array are discussed and timing waveforms are included to illustrate these operations. A large portion of the application note is devoted to explaining the synchronous flag architecture. Gate-level logic diagrams are provided to help explain flag operation. This section also explains commonly misunderstood concepts such as flag encoding and flag latency cycles. A section on programming and resetting Clocked FIFOs explains how to properly perform these operations and covers the common design pitfalls to avoid. Two sections are devoted to configuring Clocked FIFOs for depth or width expansion modes. These sections include discussions on proper flag decoding and expansion-in/expansion-out (XI/XO) pin operation. The final section discusses how to use a Clocked FIFO like an industry standard asynchronous FIFO.

Programmable FIFO flags can often simplify the design of a digital system by generating status which will prevent overrun or underrun conditions for an elastic FIFO buffer. Although many FIFOs are available with programmable flag functions on-chip, these features are not available on industry-standard asynchronous FIFOs. Of those FIFOs that do have programmable flags, some do not allow the almost-empty and almost-full values to be programmed independently, or in some cases, for these values to be programmed at any specific word boundary. This application note will present a method by which FIFOs of any size may be monitored by an external Programmable Logic Device that will then generate all of the flags necessary for most FIFO applications. The FIFO Dipstick PLD behaves like a measuring device that can observe the level of data within a FIFO.

Understanding Dual-Port RAMs

This application note examines the evolution of multi-port memories and explains the operation and benefits of Cypress's dual-port RAMs.

A dual-port RAM is a random-access memory that can be accessed simultaneously by two independent entities. In digital ICs, this implies a dual-port memory cell that can be accessed at the same time using two independent sets of address, data, and control lines.

A Brief History of Multi-Port Memories

The first multi-port memories were probably used in the CPU of the first computers. Many two-operand instructions are efficiently implemented using dual-port registers for the operands and the result.

For example, consider *Equation 1*, which describes a typical two-operand operation in the ALU (arithmetic logic unit) of a CPU:

$$(C) = (A) [OPERATOR] (B) \qquad \text{Eq. 1}$$

A and B could be either the operands (i.e., the data) or the addresses of the operands, in which case the data could be either in memory or in registers. In any case, *Equation 1* describes two pieces of data, A and B, being operated upon by the OPERATOR and the results designated as C. C could also be the data, a register, or a memory location. OPERATOR could be arithmetic or logical.

The Combinatorial ALU

The 74181 was the first integrated circuit ALU. In this IC, the 4-bit operands, A and B, are operated upon according to a 4-bit command; the result, C, is output. The chip also provides a carry-in input, a carry-out output, and A = B outputs. A mode-control pin selects either logical or arithmetic operations. The 74181 is combinatorial; no storage is provided.

Early computers used the contents of a memory location as one operand and an accumulator in the CPU as the second operand. The results were usually stored in the accumulator.

Bringing the Registers On Chip

The 67901 was the first 4-bit slice that brought 16 4-bit registers onto the chip. The MMI 67901 was second-sourced by AMD and became the 2901. At one time, five vendors offered this industry-standard bipolar ALU. The Cypress CMOS CY7C901 is the highest-performance, TTL-compatible, 4-bit slice that is form, fit, and functionally equivalent to the original 901.

The 16-word-deep, 4-bit-wide register array is functionally equivalent to a 16 x 4 dual-port memory. Four A address lines and four B address lines select the contents of two of the 16 registers, whose outputs are applied to transparent latches. The latch outputs are then applied to 3:1 multiplexers, whose outputs drive the ALU inputs. The ALU outputs can be sent off chip, entered into a temporary register (Q), or written back into the register file, thus replacing one of the operands. This architecture is shown in the CY7C901 block diagram in Cypress's 1991 *Data Book*.

CY7C901 Dual-Port Memory Operation

A simplified CY7C901 block diagram appears in *Figure 1*. The device's A and B addresses select the contents of two registers, whose outputs are applied to two 4-bit latches. When the clock (CP) is HIGH, the latch outputs follow their data inputs (i.e., are

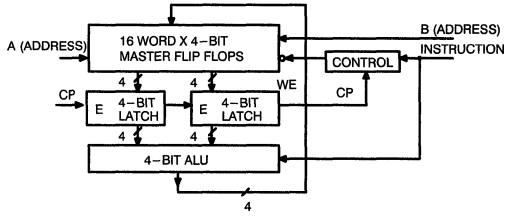


Figure 1. CY7C901 Dual-Port Memory (Simplified)

transparent). When the clock is LOW, the ALU outputs are written ($\overline{WE}$) into the register array at the location specified by the A or B addresses, depending upon the instruction being executed. A LOW on the clock causes the data in the latches not to change, so that the ALU outputs are stable when they are written back into the register array.

Note that the CY7C901 does not perform the three-port function described by *Equation 1*. In the CY7C901, the C operand equals either the A or B operand, depending upon the instruction being executed. In fact, the A and B addresses can be the same. An old programming trick is to Exclusive-OR the contents of a register with itself, which clears the register.

Additionally, the CY7C901's dual-port memory does not use a dual-port memory cell. This type of cell is not required because the CY7C901 does not need the ability to simultaneously write independently to two separate memory locations.

Dual-Port Memory Using Single-Port RAM

Before the dual-port memory cell existed, designers created dual-port RAMs from single-port RAMs by adding a multiplexer between the RAM and the two entities that shared the RAM. *Figure 2* illustrates a block diagram of such an arrangement. Two processors, MP1 and MP2, share the RAM. If each processor has access to the RAM half the time, the resource is shared equally and is said to be allocated according to a fairness doctrine.

This time division multiplexing assures that there is no contention for the RAM. However, performance

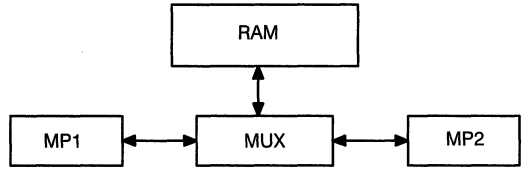


Figure 2. Dual-Port Memory Using Single-Port RAM

suffers if the RAM's access time does not equal 1/2 or less of the processors' clock period, assuming that the processors are clocked from the same source.

For example, consider two processors clocked from the same 25-MHz source, for a period of 40 ns. Because the processors are closely coupled, only one operating system is in memory. In this case, the maximum access time of the dual port has to be 20-ns or less. The highest-speed dual-port RAM available has a 25-ns access time. Therefore, each processor suffers a worst-case 20% performance degradation.

Dual-Port RAM Applications

The first applications for dual-port memories were for CPU register files. Dual-port RAMs can also serve as data or instruction cache memories. However, the largest usage of dual-port RAMs is in communications, which includes the exchange of data between processors, processes, and systems.

Virtual Dual-Port RAM

Communication between systems does not require physical dual-port RAMs. Instead, a conventional RAM memory is partitioned into virtual data-storage areas (buffers), usually to store at least two data packets. These buffers are shared between the communications controller and the intelligent element that assembles the packets and stores them (usually a microprocessor). The communications controller can also be a microprocessor. It reads the data from memory, converts the data from parallel to serial form, encodes the data, converts the data to analog form, and sends the data out over the communications channel on the transmit side. If the system contains only one processor, the data buffers are not shared, and the system needs neither a virtual nor a physical dual-port RAM.

Control information associated with each data buffer tells the communications controller the number of words in the buffer and the starting address of the data in the buffer. The control information resides in one or more memory locations whose addresses have been previously agreed upon by the two processors.

This simple software-based buffer example requires a second level of control—a mechanism or procedure that prevents the two microprocessors from getting in each other's way. In other words, the system needs a procedure control mechanism.

Another way of analyzing this requirement introduces the concept of data ownership. Say, for example, that processor A assembles and stores messages and thus owns the data while performing these tasks. Likewise, the communications processor B owns the data while performing its tasks. The procedure control mechanism amounts to a technique for transferring data ownership between processor A and B.

In large systems, where many processors perform many different operations, the processing of the information is called a job or a procedure. The procedure is divided into many tasks, which can be performed by different processors. The tasks can either be scheduled and assigned by a processor dedicated to that task or be performed by any available processor. These alternatives are referred to as autocratic and egalitarian systems, respectively. The term egalitarian implies that the processors are treated equally. In either case, the processors must have access to a shared-memory location used for message passing.

Synchronizing sequential processes is the cornerstone of concurrent programming, which applies to multi-tasking, single-processor systems; distributed-processor networks; and tightly coupled multiprocessor systems.

Message Passing

In the two-processor system under consideration, synchronization can be achieved by using a lockword or lockvariable. The lockvariable can apply either to data (as in this example) or to executable instructions.

The lockvariable is a location in shared memory that is operated upon using two synchronization primitives: LOCK (v) and UNLOCK (v), where (v) is the location operated upon. These are simple binary switch operations. If a processor wishes to lock or own a critical section of code or data, the processor indivisibly sets the lockvariable if testing shows the lockvariable to be zero. If the lockvariable is not zero, then the operation is repeated until the lockvariable is zero. To unlock the critical section, a processor sets the lockvariable to zero and continues.

Most modern processors have indivisible read/modify/write instructions, also called test and set (TAS) instructions. In Reference 1, however, E. W. Dijkstra shows that lockvariables can be implemented without using a read/modify/write instruction. And in Reference 2 he develops the semaphore, a technique for managing a queue of tasks waiting for a resource. Lockvariables surround or bracket semaphores and thus provide entry and exit control on a mutual-exclusion basis.

Typical TAS Instruction

The current example assumes that the processors have a TAS instruction. A typical TAS instruction operates as follows: read, test, and set to X. The addressed memory location is read, and if its contents are zero, the value X is written into that location. If the contents are not zero, the contents are returned to the processor, and the value in the memory location is not disturbed.

The usual convention is that a value of zero in the lockvariable means that the resource associated with it is available. A non-zero value means that another processor temporarily owns the resource and that the resource is not available. After performing the task associated with the lockvariable, the processor sets the lockvariable's value to zero. The system is initialized with all lockvariables set to zero.

In the current example, processor A performs a TAS operation on the lockvariable and, finding the lockvariable to be zero, sets the lockvariable to a one. This tells processor B that the message is in the pro-

cess of being assembled in the memory buffer area and is not ready to be transmitted. Processor A then assembles the message. After the message is assembled, processor A clears the lockvariable, sends a message to processor B saying that the message is ready to be transmitted, and gives the data's location and the number of bytes to be sent. Processor B reads the message from processor A and performs a TAS operation on the lockvariable; finding the lockvariable to be zero, processor B sets it to a two. This tells processor A that the message is in the process of being transmitted. Processor B then transmits the message and clears the lockvariable. Processor B sends processor A a message that the transmission task has been completed. After receiving the message from processor B, processor A performs a TAS operation on the lockvariable; finding the lockvariable to be zero, processor A concludes that the message has been successfully transmitted.

Note that this procedure does not require the use of a dual-port RAM. The procedure does require each processor to perform a TAS instruction, clear the lockvariable, and send a message to the other processor. Sending a message implies writing to a location in shared memory. To know that a message is waiting, the processor receiving the message must either read the memory location periodically (referred to as polling a mailbox) or the act of writing to the mailbox must generate an interrupt to the receiving processor. The interrupt-driven alternative is usually preferred because the receiving processor does not have to waste time in a polling sequence.

Dual-Port RAM Cell History

The first dual-port RAM ICs to use a dual-port RAM cell were the Synertek SY2130 and SY2131, introduced in 1983. These products are organized as 1024 words of 8 bits and use n-channel, double-polysilicon technology to achieve 100-ns access times. The SY2130 has an automatic power down feature controlled by the chip enables, and the SY2131 does not. The smaller (512 x 8) SY2132 and SY2133 were similar but unsuccessful.

The original dual-port RAMs include two mailboxes for message passing. When written to from one port, a mailbox generates an interrupt to the opposite port. Additionally, on-chip arbitration logic generates a busy signal to the loser when both left and right ports address the same memory location. If the loser was attempting to write, the write is suppressed.

Most of the dual-port RAMs on the market today are functionally equivalent to the original Synertek products. The "new features" added to several dual-port RAM products by Cypress, Motorola, and Integrated Device Technology (IDT) include dedicated semaphore registers. Hardware semaphores provide efficient means of allocating exclusive priority accesses to blocks of shared memory locations in dual-port RAMs.

The SY2130 was second-sourced by IDT in 1984 and Advanced Micro Devices (AMD) in 1985. IDT also doubled the density to 2K x 8 and called the new part the IDT7132. Due to pin limitations (48 pins), the interrupt functions were deleted.

In 1985 IDT added slave companion parts to the company's dual-port family. The IDT7140 (1024 x 8) is the slave to the IDT7130, and the IDT7142 (2K x 8) is the slave to the IDT7132. The slave device provides word-width expansion. BUSY is an input to the slave from the master, and the slave contains no arbitration logic. One master can drive many slaves. This arrangement avoids the classic deadly embrace problem described in the next section.

The Deadly Embrace

The deadly embrace can occur when two masters are connected in parallel to make a wider word. If the left and right port addresses match, and the left and right port chip enables then become active to both chips at approximately the same time, it is possible to have one port of one master lose and the opposite port of the other master also lose. In other words, if an address match occurs and both ports are enabled during a small time window or an aperture of uncertainty, the dual-port RAM cannot determined which port wins or loses.

Under these conditions, if the corresponding left and right port busy pins are connected together, both ports of both masters are active (LOW). This condition occurs because the busy outputs are open drain, and the loser pulls the node LOW.

This condition is the simplest example of the deadly embrace. As far as the external world is concerned, both ports are busy, and the system remains locked up indefinitely, with each port waiting to be released by the other. Each master's arbiter section thinks it has lost the arbitration and is waiting to be released by the other.

In general, the deadly embrace occurs under two conditions: a processor requires one or more resources to perform a task, and one or more of the required resources is temporarily owned by another processor, which requires one or more of the same resources to perform its task.

For example, if processor A owns resource X and processor B owns resource Y, and both resources are required to accomplish the task, a stalemate occurs in which each processor waits for the other to relinquish the required resource. This is the simplest example. The concept extends to *n* processors and *m* resources.

The solution to the deadly embrace depends upon whether the system is autocratic or egalitarian, the tasks' priorities, etc., and is beyond the scope of this discussion. In the case of dual-port RAMs, however, the solution is simple: Do not cascade two masters in width; use a master and a slave.

The Cypress Dual-Port RAM Family

Table 1 lists the members of the Cypress dual-port RAM family. The package designator D26 stands for 600-mil ceramic DIP, and P25 stands for 600-mil plastic DIP. The 48-pin ceramic leadless chip carrier (LCC) is designated as L68. The 52-pin packages are designated as L69 for ceramic LCC and J69 for plastic LCC (PLCC). The 68-pin packages are designated by L81 for ceramic LCC, J81 for plastic LCC (PLCC), and G68 for ceramic pin grid array.

Note that the interrupt function is not available at the 2048 x 8 level in a 48-pin package. This is due to pin limitations. At the 2-Kbyte level, each port requires an additional address pin for the address's most significant bit.

The M/S column in *Table 1* indicates whether the device is a master or slave. The difference between these devices is that the masters have arbitration logic and the slaves do not. The busy signals are outputs from the master and inputs to the slave. (The ramifications of this are examined later.)

Table 1. The Cypress Dual-Port RAM Family

Config.	Part #	Min. Access	M/S	Sem.	Int.	Busy	Package Options			
							DIP (P)	PLCC (J)	TQFP (A)	PQFP (N)
1Kx8	CY7C130	25	M	N	Y	Y	48			
	CY7C131	25	M	N	Y	Y		52		52
	CY7C140	25	S	N	Y	Y	48			
	CY7C141	25	S	N	Y	Y		52		52
2Kx8	CY7C132	25	M	N	N	Y	48			
	CY7C136	25	M	N	Y	Y		52		52
	CY7C142	25	S	N	N	Y	48			
	CY7C146	25	S	N	Y	Y		52		52
2Kx16	CY7C133	15	M	N	Y	Y		68		
	CY7C143	15	S	N	Y	Y		68		

Table 1. The Cypress Dual-Port RAM Family (continued)

Config.	Part #	Min. Access	M/S	Sem.	Int.	Busy	Package Options			
							DIP (P)	PLCC (J)	TQFP (A)	PQFP (N)
4Kx8	CY7B134	20	M/S	N	N	N	48			
	CY7B135	20	M/S	N	N	N		52		
	CY7B1342	20	M/S	Y	N	N		52		
	CY7B138	15	M/S	Y	Y	Y		68	64	
4Kx9	CY7B139	15	M/S	Y	Y	Y		68	80	
4Kx16	CY7C024	15	M/S	Y	Y	Y		85	100	
8Kx8	CY7B144	15	M/S	Y	Y	Y		68	64	
8Kx9	CY7B145	15	M/S	Y	Y	Y		68	80	
8Kx16	CY7C025	15	M/S	Y	Y	Y		84	100	
8Kx18	CY7C0251	15	M/S	Y	Y	Y		84	100	
16Kx8	CY7C006	15	M/S	Y	Y	Y		68	64	
16Kx9	CY7C106	15	M/S	Y	Y	Y		68	80	

Cypress Dual-Port RAM Operation

A simplified block diagram of the Cypress dual-port RAM appears in *Figure 1*. The device interface includes three types of signals: address, data, and control. There are two sets of these signals: those of the left port and those of the right port. Each signal has either the subscript L or R to designate left or right, respectively.

The address pins are designated A0 through A9 (1024 x 8) and A0 through A10 (2048 x 8), where A0 is the least significant bit (LSB) and A9 or A10 is the most significant bit (MSB). The address pins are unidirectional inputs to the device; their states specify the memory location to be read from or written into.

The data pins are designated I/O0 through I/O7, where I/O0 is the LSB and I/O7 is the MSB. The data pins are bidirectional; their states represent either the data to be written or the data to be read.

The control pins are chip enable ($\overline{CE}$), read/write (R/$\overline{W}$), and output enable ($\overline{OE}$). A semaphore en-

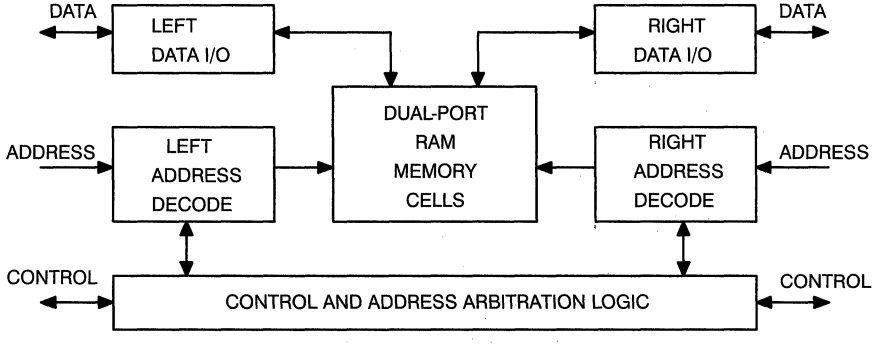


Figure 1. Dual-Port RAM Block Diagram

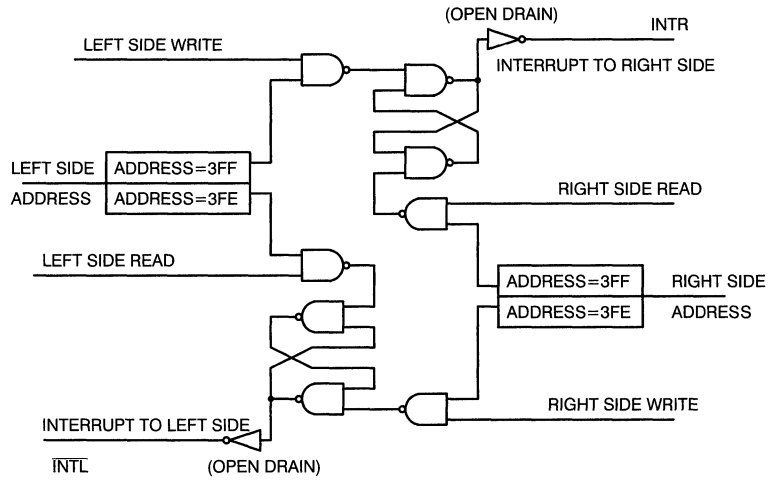


Figure 2. Interrupt Logic

able control pin ($\overline{SEM}$) is included on dual-port RAMs with semaphores. Two flags are also provided, $\overline{INT}$ and $\overline{BUSY}$; both have open-drain outputs and require external pull-up resistors. A LOW on the chip enable input allows that port to become functional. Data is either read from the internal dual-port RAM array or written into it, depending upon the state of the read/write signal; a LOW initiates a write operation. The three-state data output drivers are enabled by a LOW output enable.

When one port writes to a pre-determined mailbox, an interrupt to the other port is generated. When the interrupted port reads that memory location, the interrupt is reset.

When both ports address the same memory location and both chip enables are active (LOW), contention occurs for that address. An arbitration is then performed, and ownership of the memory location is assigned to the winner. An active (LOW) busy signal notifies the loser of the arbitration.

Dual-Port RAM Functional Description

An important aspect of the Cypress dual-port RAMs is their interrupt logic. A simplified logic diagram of this logic appears in *Figure 2*, with the chip enables deleted. A port's chip enable must be asserted for the port to either read from or write to any location, including the mailboxes. Note that you can use the mailbox locations as conventional memory by not connecting the interrupt line to the appropriate processor.

The upper two memory locations (7FF and 7FE for 2K x 8; 3FF and 3FE for 1K x 8) can be used for message passing. The highest memory location serves as the mailbox for the right processor. When the left processor writes to this mailbox, the interrupt (request) to the right processor, $\overline{INTR}$, goes LOW. When the right processor reads its mailbox, the flip-flop is reset, and $\overline{INTR}$ goes HIGH.

The second highest memory location serves as the mailbox for the left processor. When the right processor writes to this mailbox, the interrupt (request) to the left processor, $\overline{INTL}$, goes LOW. When the left processor reads its mailbox, the flip-flop is reset, and $\overline{INTL}$ goes HIGH.

Note that each port can read the other port's mailbox without resetting the associated flip-flop. If your application does not require message passing, leave the appropriate pin open. Do not connect a pull-up resistor to the pin, and do not connect the pin to the processor's interrupt request pin.

Table 2. Functional Operation of Dual-Port Masters

Operation			Result of Operation after Arbitration (Master)
Case	Left Port	Right Port	
1	Read	Read	Both ports read.
2	Read	Write	Loser is prevented from writing. If loser is reading and ports are asynchronous, data read might not be valid.
3	Write	Read	
4	Write	Write	

Note that the active state of the busy signal prevents a port from setting the interrupt to the winning port. Additionally, an active busy signal to a port prevents that port from reading its own mailbox and thus resetting the interrupt. These operations are ramifications of the data-ownership concept.

If both ports address the same memory location at the same time, the master performs an arbitration, so that one port wins and the other loses. Because each of the two ports can be in either the reading or writing state, there are four possible combinations of ports and states (*Table 2*).

Both Ports Reading

If both ports of a dual-port IC read the same location at the same time, you can assume that both ports read the same data. When arbitration occurs as a result of contention in a Cypress dual-port RAM, the port that wins the arbitration gets temporary ownership of the memory location. The losing port can read the memory location but the busy signal tells it that it lost the arbitration.

To guarantee data integrity in a multiprocessor system, it is standard practice to apply the concept of data ownership. This ownership can apply to executable code, data, or control locations in memory. The control locations in memory can be associated with a resource, such as a printer, tape drive, disk drive, or communications port.

One Port Reading, the Other Writing

The result of arbitration will allocate priority to either the reading or the writing port. In Cypress dual-port RAMs, if the losing port is attempting to write data, the write is inhibited so that the data in memory is not corrupted. The $\overline{\text{BUSY}}$ flag to the losing port signals that the write was not performed.

If the losing port is attempting to read data, it is possible for the data to be old data, new data, or some random combination of the two. The $\overline{\text{BUSY}}$ flag to the losing port signals that the old data is still being read on the losing port's data lines. The old data will remain undisturbed for an access time after either $\overline{\text{BUSY}}$ on the losing port goes HIGH, the losing port's address is toggled, $\overline{\text{CE}}$ for the losing port is toggled, or R/$\overline{\text{W}}$ for the losing port is toggled during a valid read.

If the new data is needed, the $\overline{\text{BUSY}}$ flag can be used to generate a delay until the new data is present or can signal a processor to attempt the read again after $\overline{\text{BUSY}}$ is cleared.

Both Ports Writing

The losing port is prevented from writing so that the data cannot be corrupted. $\overline{\text{BUSY}}$ is asserted to the losing port, indicating that the write operation was unsuccessful.

Arbitration Logic

Figure 3 shows the arbitration logic used in Cypress dual-port RAM masters. The arbitration logic has three functions: to decide which port wins and which loses if the addresses are equal simultaneously, to prevent the losing port from writing, and to provide a busy signal to the losing port.

The arbitration logic consists of left and right address equality comparators with their associated delay buffers; the arbitration latch formed by the cross-coupled, three-input NAND gates labeled L and R; and the gates that generate the busy signals.

Operation With Unequal Addresses

When the addresses of the right and left ports are not equal, the outputs of the address comparators (nodes A and B) are both LOW, and the outputs of the gates labeled L and R (nodes C and D) are both HIGH. This condition forces both $\overline{\text{BUSY}}$ signals HIGH and both $\overline{\text{Write Inhibit}}$ signals HIGH. The arbitration latch does not function as a latch.

Left Port Camped on an Address

Next, consider the condition where the left-port address and chip enable are quiescent, and the right-port address changes to an address equal to that of the left port. Nodes A and B are initially LOW.

Because the right-port address does not go through the delay buffer, the output of the right-address comparator (node B) goes HIGH before node A goes HIGH by a delay interval, d. The delay must be greater than the delay through the R gate, so that when node B goes HIGH, node D goes LOW, causing node C to remain HIGH. CE(R) and CE(L) are both HIGH; they are the inverse of the chip enable inputs. Node D going LOW causes the output of the BR gate to go LOW, which tells the right port that the memory location it just addressed belongs to the left port. A write inhibit signal is also generated that prevents the right port from writing into the addressed memory location.

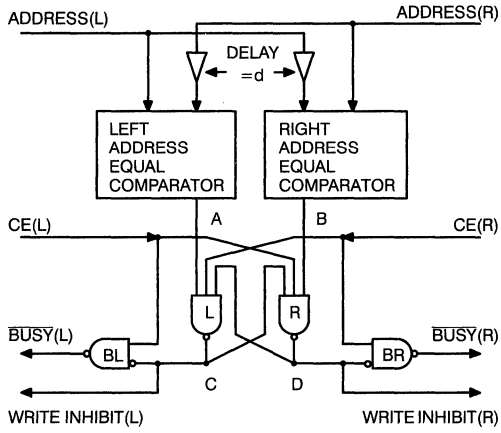


Figure 3. Arbitration Logic

In summary, when the right port addresses a memory location that is already being addressed by the left port, a delay occurs that equals the sum of the propagation delays of the right-address comparator, the R gate, the BR gate, and the output driver (not shown in the diagram). Then the busy signal to the right port is asserted. Nodes A, B, and C are now HIGH, and node D is LOW. $\overline{\text{BUSY}}$ is asserted to the right port.

Due to the symmetry of the arbitration logic, the device operates the same when either the right or left ports are camped on an address.

Right and Left Addresses Equal Simultaneously

In the general case, it is possible to have both ports access the same memory location simultaneously, unless this is guaranteed not to occur by the design of the system. When nodes A and B go from LOW to HIGH at exactly the same instant, the arbitration latch settles into one of two states and determines which port wins and which port loses. The latch is designed such that its two outputs are never LOW at the same time. It also has a very fast switching time.

The dual-port RAM imposes a minimum time difference between either of two events: the two chip enables going from inactive to active and the two sets of addresses going from mismatch to equal. If the events are close together in time, the probability of each port either winning or losing the arbitration is approximately equal. This parameter is called port set-up time for priority and is abbreviated as t_{PS} on the datasheets. The specified value is 5 ns. (Note, though, that Cypress product engineers have measured t_{PS} at room temperature and nominal V_{CC} (5V) and found a value of approximately 200 ps.) In other words, if one port addresses a memory location 5 ns before the other port, the first port is guaranteed to win. If not, the result of the subsequent arbitration is unpredictable.

Other Key $\overline{\text{BUSY}}$ Parameters

Several other key parameters are specified with respect to the busy signal. For example, $\overline{\text{BUSY}}$ LOW from address match, t_{BLA}, is the maximum time it takes busy to go LOW, as measured from the time the two port addresses are the same. This is the time from an address match until the losing port is notified that it has lost the arbitration. Obviously, the sooner this occurs the better. If the value of t_{BLA} is greater than the memory cycle time, another cycle must be added to detect the condition, which can severely reduce performance. This time is less than the minimum cycle time for all speed grades of all Cypress dual-port RAMs.

Another parameter, $\overline{\text{BUSY}}$ HIGH from address mismatch, t_{BHA}, is the maximum time it takes $\overline{\text{BUSY}}$ to go from LOW to HIGH, as measured from the time the two port addresses do not match until the $\overline{\text{BUSY}}$ signal goes HIGH. The comments of the preceding paragraph also apply here.

The next two parameters are similar to the preceding two. The difference is that the chip enable controls the busy signal. The parameters are $\overline{\text{BUSY}}$ LOW from $\overline{\text{CE}}$ LOW, t_{BLC}, and $\overline{\text{BUSY}}$ HIGH from $\overline{\text{CE}}$ HIGH, t_{BHC}. Both of these parameters are less than the minimum cycle time for all speed grades of all Cypress dual-port RAMs.

$\overline{\text{BUSY}}$ HIGH to valid data, t_{BDD}, is the maximum time it takes the data to become valid to the losing port after $\overline{\text{BUSY}}$ goes away. This parameter's value equals the address access time, t_{AA}, because a read cycle is initiated to the losing port when its $\overline{\text{BUSY}}$ signal transitions from LOW to HIGH. An action by either port can cause the busy transition. The winning port can either change its address or deassert its chip enable.

To illustrate the last two parameters, *Figure 4* shows the timing for the right port performing a write operation and the left port asynchronously moving to the same address and attempting to perform a read operation. The first parameter of interest is t_{DDD}, which is the maximum time between the stabilization of the data to be written by the winning port and that same data becoming valid at the outputs of the port that received the $\overline{\text{BUSY}}$. The second parameter of interest is t_{WDD}, which is the maximum time between the HIGH-to-LOW transition of the winning port's write strobe and the data becoming valid at the outputs of the port that received the $\overline{\text{BUSY}}$.

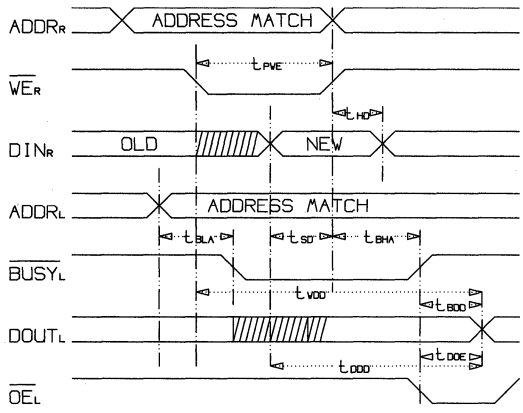


Figure 4. $\overline{\text{BUSY}}$ Timing

It is possible for the losing port to read either the old data, the new data, or some random combination of the two under these circumstances: the two ports are operating asynchronously (i.e., with independent clocks), and the conditions illustrated in *Figure 4* occur (winning port writing and losing port reading). If the read occurs early with respect to the write, old data is read. If the read occurs late with respect to the write, new data is read. And, if the read occurs at the same time the data is changing from old to new, the data read is not predictable. However, all is not lost. There are two general solutions. Both use the fact that the busy signal is asserted to the losing port, telling the port in this instance that the data it is reading might not be valid.

One solution is to use the HIGH-to-LOW transition of the busy signal to the losing port to generate an interrupt to the processor (or state machine) so that operation can be repeated. The drawback of this technique is that a snapshot of the states of the losing port's address lines and read/write line must be taken, so that the processor can tell what load/store operation caused the interrupt. Taking this snapshot requires latches or flip-flops for the data and control logic for doing the sampling, and the tech-

nique uses up an interrupt line. The processor must also be able to read the sampled data later.

A second solution is to use the LOW level of the $\overline{BUSY}$ signal to the losing port to prompt one of three types of delays: delay the reading of data until the data becomes valid, which occurs an access time after the LOW-to-HIGH transition of $\overline{BUSY}$; insert wait states until $\overline{BUSY}$ goes HIGH; or stretch the clock until $\overline{BUSY}$ goes HIGH. Any of these methods probably require less hardware and control logic than the preceding approach. Use of these methods does mean that the $\overline{BUSY}$ signal must eventually go from LOW to HIGH. This happens when the winning port either changes its address or deasserts its chip enable. For this reason, as well as for system noise immunity and power-saving considerations, it is recommended that blocks of addresses be decoded to generate chip enables for the dual-port RAMs.

Because the losing port has no control over the winning port in the general case, however, a question arises: What can the losing port do to successfully read the data just written, assuming the winning port does not change its address, write, or chip enable signals? There are two possible operations:

1. Change an address line to a different address, then change back to the original address. This toggles the $\overline{BUSY}$ signal to the losing port.
2. Change the state of the chip enable. This also toggles the $\overline{BUSY}$ signal to the losing port.

Hardware Semaphores

Cypress offers dual-port RAMs with eight on-chip hardware semaphore latches that are independent from RAM memory locations. Semaphore signaling is a popular method of allocating mutually exclusive accesses to blocks of memory that are shared among several processors. Exclusive processor control guarantees data integrity in sensitive applications such as shared I/O buffers. Semaphore signaling can also improve the efficiency of block memory accesses by preventing delays and processor stalls due to a memory location being busy from another processor access.

Traditional semaphore signaling has been implemented in software using dedicated memory locations to hold the semaphore signals. A processor could attempt to gain control of a semaphore by using an indivisible test and set instruction to test if the semaphore was set by another processor. If the semaphore is free, the processor sets the semaphore and gains exclusive control of a block of memory.

Cypress dual-port RAMs have on-chip hardware semaphores that are independent from RAM memory locations. Hardware semaphores eliminate the need to use a processor with an indivisible test and set instruction. Semaphore control requests are handled using a standard write to the semaphore latch followed by a read instruction. There is no requirement to lockout other processor accesses to the semaphore between the write and read.

The hardware semaphores provide flexible software configuration of shared memory. The semaphores operate independent of any memory in the RAM allowing software to allocate block addresses and block sizes.

Cypress hardware semaphores implement a "token passing" scheme allowing the port in possession of the token to have exclusive access to a block of shared memory. Possession of the token can only be relinquished by the port with possession. A port's request for possession of the token will be denied if the token is held by the other port.

Possession of a token is indicated by the state of a semaphore latch formed from two cross-coupled NOR gates (see *Figure 5*). The latch can be set so that only one port controls the semaphore at a time. Additional input latches on the semaphore ports are used to hold requests to set or clear the latch. An output latch on each port is used to prevent the output from changing during a read from the port.

The semaphore latches are accessed through the data and address ports the same way as a RAM cell access. The semaphore enable line ($\overline{SEM}$ = LOW) initiates a semaphore access cycle. The A0–2 lines select which semaphore latch is accessed. Only the data on D_0 is latched into the semaphore during a write. The other data lines are ignored. During a

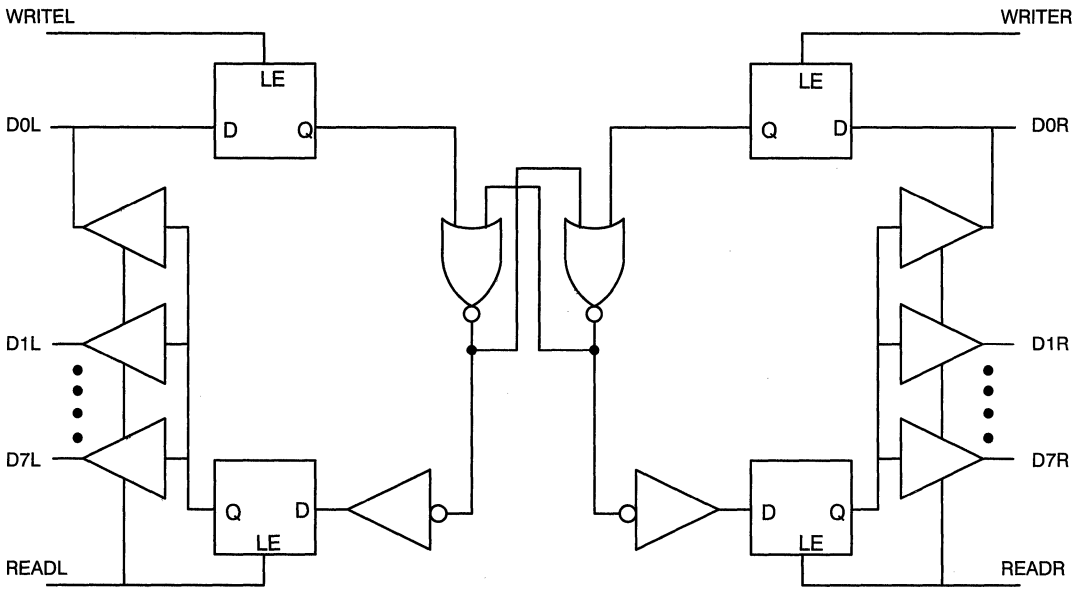


Figure 5. Semaphore Latch Cell

read, the semaphore drives all the data lines (D0 through D7, D8) with the semaphore signal.

A processor requests control of a semaphore by writing a 0 to the D0 port of the semaphore addressed by A0–2. The 0 is latched into the port's input register and held until another write attempts to set it to 1. If the semaphore is free at the time of the request, the port will immediately be granted control of the semaphore. If the semaphore is controlled by the other port, the request for control will be denied. If control of the semaphore is relinquished by the other port while the 0 is still pending, then the requesting port will gain control of the semaphore. Control of the semaphore can only be relinquished by the controlling port by writing a 1 to the semaphore.

To see if a request for control of the semaphore was successful, a read of the semaphore is performed. A port controls the semaphore if 0 is read out on D0. The port does not control the semaphore if a 1 is read. The semaphore outputs drive all of the data lines with the state of the semaphore, so D0–7 will be "00000000" when control is granted and will be "11111111" when control is denied. The state of the internal semaphore latches may change during a read, but the output latch prevents the changes from propagating to the data lines. A new read cycle must be performed in order to update the port's output lines.

If both ports attempt to write a 0 within t_{SPS} of each other while the semaphore is free, semaphore arbitration logic will guarantee that only one side gains control of the semaphore.

Address Transition Detection

Why does changing the address or chip enable allow a losing port to read data successfully? All Cypress dual-port RAMs, both masters and slaves, use a circuit design technique called Address Transition Detection (ATD) to improve performance and reduce power dissipation.

ATD improves performance by equilibrating differential paths, pre-charging critical nodes, and forcing the outputs to a high-impedance state. Equilibration and pre-charging will bias critical nodes to voltage levels approximately in the mid-point of the small-signal operating range; when the data is sensed, it takes a shorter amount of time to transition to the 0 or 1 level. Forcing the outputs to their high-impedance states improves speed slightly, but more importantly, the technique reduces output switching noise by eliminating crowbar current and separating the output current into two pulses instead of one.

ATD minimizes power consumption because it turns on power-hungry circuits only when they are required. Slightly over 50 percent of a RAM's circuits are linear, and approximately 70 percent of the power is dissipated in the sense amplifiers during a read operation. When the RAM is operating at its maximum frequency, the ATD circuits are constantly triggered, so the power savings are minimal. At lower speeds or smaller duty cycles, however, the power savings are significant.

A diagram representing a typical ATD sequence is illustrated in *Figure 6*. The event that triggers the ATD sequence for either port is the transition of any address, chip-enable, or read/write signal. Equilibration and pre-charging are performed next, followed by either turning on the sense amplifiers and latching the data (read operation) or pulling the BIT and $\overline{BIT}$ lines to the required levels (write operation) at the addressed location. The master clock pulse lasts from 7 to 11 ns, depending upon temperature, supply voltage, and the distribution of IC processing parameters. At the end of the pulse, the data is latched and the appropriate circuits are turned off.

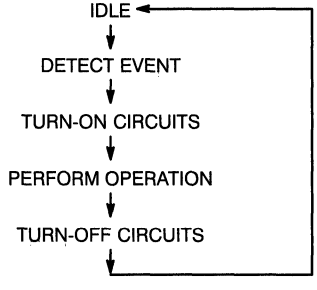


Figure 6. Simplified ATD Sequence

Master Standalone Operation

Figure 7 presents a block diagram of a system using two 8-bit microprocessors, the Cypress CY7C132 dual-port RAM, static RAM, and EPROM. The address lines of each microprocessor are decoded to generate the chip enables to the dual-port RAM, the SRAM, and the EPROM. Note that pull-up resistors are required on the interrupt requests to the microprocessors and the busy signals, which go to the microprocessors' wait inputs.

Slave Word-Width Expansion

The block diagram in *Figure 8* shows how to interconnect a CY7C132 (2K x 8) master and a CY7C142 (2K x 8) slave to form a 16-bit-wide word. The diagram does not show the interfaces to the processors or the connections for the interrupt signals. As previously explained, the interrupt outputs are not available at the 2K x 8 level in the 48-pin DIP due to pin limitations. In the LCC and PLCC packages, the interrupt outputs are available from both the master and the slave devices. You can use either one. You do not have to tie the corresponding interrupt pins of the master and the slave together.

Delaying the Write Strobe

In width expansion, the write signals to the slave devices must be delayed by an interval at least equal to t_{BLA}, which is the time required for the master to assert the busy signal to the slave after an address match. The delay prevents the slave data at the address in contention from being overwritten. Both the write and read cycle times must be increased by this amount of time. In equation form:

$$t_{WC} = t_{PWE} + t_{BLA} \qquad \text{Eq. 2}$$

where the delay must be at least equal to t_{BLA}.

Note that if you add more slaves to make a wider word, (e.g., 24 or 32 bits) the delay elements' outputs can connect directly to the write strobe inputs. Additional delay elements are not required.

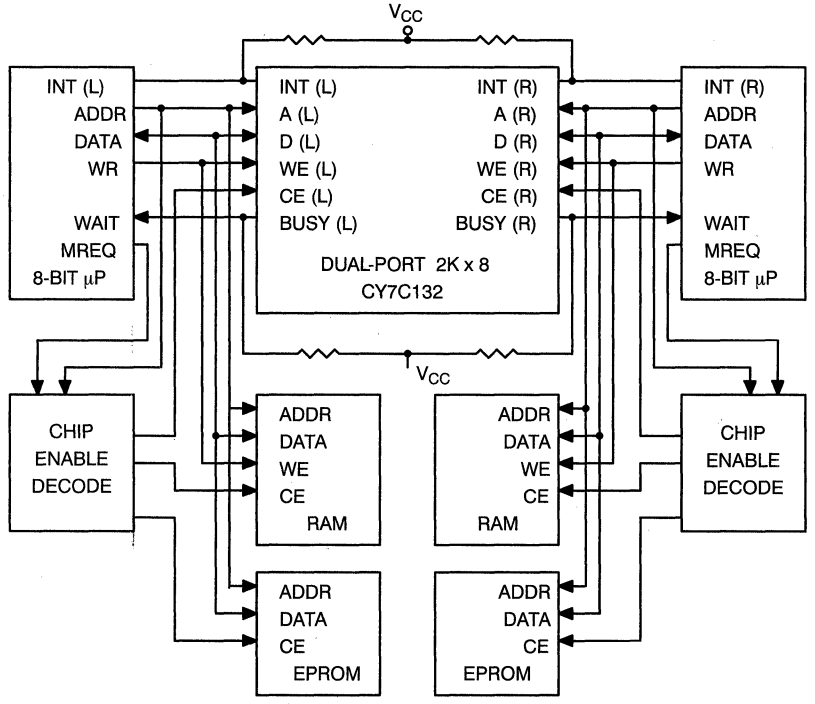


Figure 7. Typical 8-Bit Microprocessor

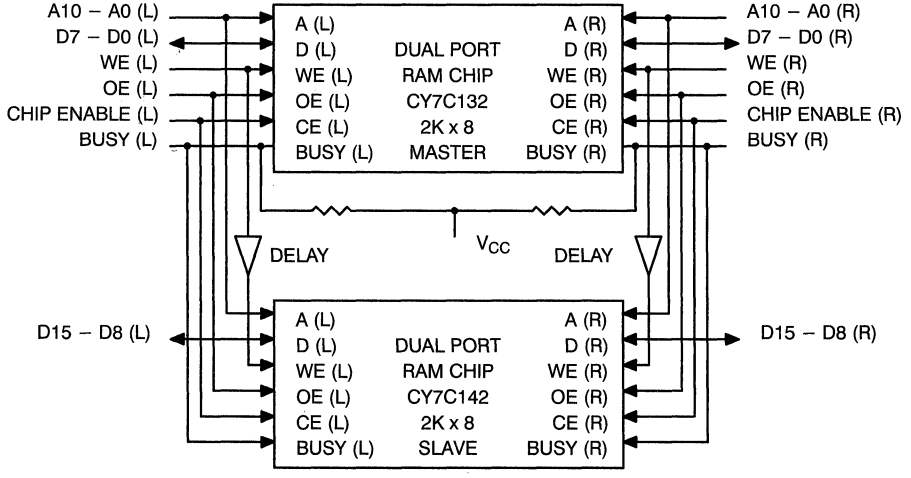


Figure 8. Expansion (2K x 16) with Slave

Slave Standalone Operation

Some applications might require that you give one port permanent and absolute priority over the other. You can easily do this by implementing the memory using only slave dual-port RAMs. The $\overline{BUSY}$ input to the priority port must be tied HIGH by either connecting it directly to V_{CC} or to V_{CC} through a 10-KΩ pull-up resistor. You can connect the low-priority port's $\overline{BUSY}$ input to the high-priority port's read/write input.

In this configuration, the busy (read/write) signal to the lower-priority port always prevents the port from writing when the high-priority port is writing to any location. The data of the lower-priority port is overwritten when the two ports operate asynchronously, the lower-priority port is writing, and the higher-priority port simultaneously writes. This is not a very elegant solution because the $\overline{BUSY}$ input to the low-priority port is not qualified by comparing the addresses of the two ports or their chip enables. However, this approach suggests how the slave dual-port RAMs can be used with external arbitration logic. The busy inputs can be used by control logic or under program control to dynamically change the port priorities.

If the lower-priority port is read only, you can tie its $\overline{BUSY}$ input HIGH by either connecting it directly to V_{CC} or to V_{CC} through a pull-up resistor.

Dual-Port Design Example

The following design example illustrates the methodology to follow when designing with Cypress dual-port RAMs. In this example, a dual-port memory is used for message passing and bus snooping for many bus masters on a 32-bit-wide system bus. The dual-port RAMs interface to a 32-bit system bus on the right side and a 16-bit processor on the left side. From the right port, the memory appears as 8K 32-bit words, and from the left port the memory appears as 16K 16-bit words.

The memory has the following characteristics:

1. The memory location corresponding to address 0 for both ports is the same.
2. The data read from and written to the memory from both ports is in the same order. Thus, D0 of the right port corresponds to D0 of the left port. Additionally, D16 of the right port appears as D0 of the left port in address location 2048.
3. The minimum cycle time is 35 ns.
4. To conserve power, blocks of addresses are decoded to generate the required chip selects.
5. The CY7C132 and CY7C142 dual-port RAMs are used. Part of the design task is to specify the number of masters and slaves required and the way they must be interconnected.
6. The appropriate $\overline{BUSY}$ signals must be generated to the correct port when contention occurs.
7. All possible mailbox locations that can be used for message passing are used.
8. The right port signals are AR0...AR12, DR0...DR31, $\overline{CER}$, and $\overline{BUSYR}$. The left port signals are AL0...AL13, DL0...DL15, $\overline{CEL}$, and $\overline{BUSYL}$.

A simplified logic diagram of the memory appears in *Figure 9*. A total of 16 2K x 8 dual-port RAMs are required. The devices labeled MA (master, bank A) through MD (master, bank D) are CY7C132 masters. The devices labeled SU (slave, upper half-word) and SL (slave, lower half-word) are CY7C142 slaves. The memory consists of four masters and twelve slaves, along with the required control logic.

From the right port the memory is configured as 8K 32-bit words, with a master controlling three slaves. The one-of-four decoder labeled RB (right bank) generates chip enable signals for each bank of 2K 32-bit words. Data is written (sampled) on the bus side, and the only reads performed are from the mailbox locations.

A general-purpose, right-port, control-logic block generates control signals that conform to the timing diagram shown in *Figure 10*. The diagram does not show the generation of the output enable control signals, but they are similar to the RB decoder signals. If your application does not require message passing to the right port, you can tie the right-port output enable pins of all of the dual-port RAMs directly to V_{CC}.

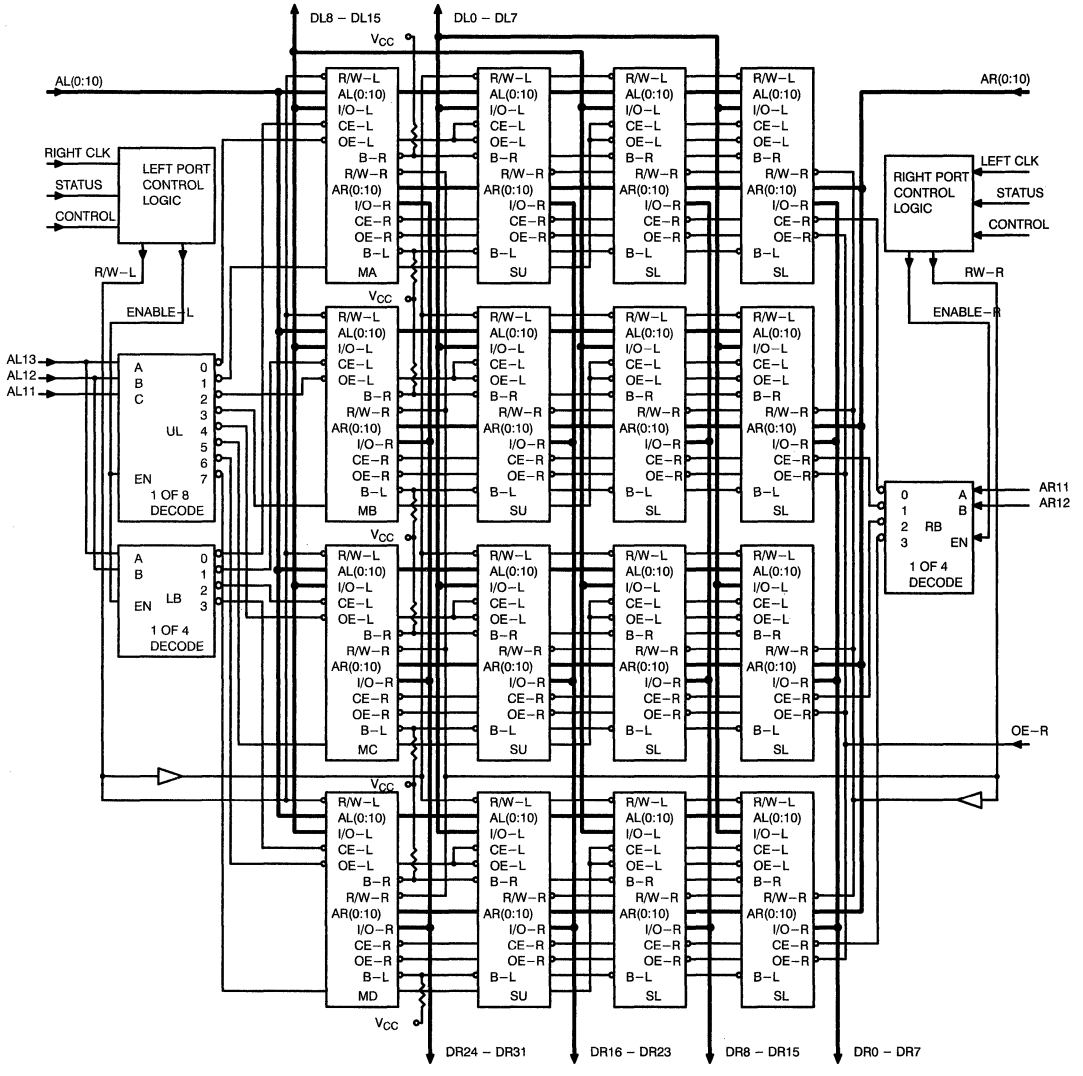


Figure 9. Logic Diagram for Dual-Port Example

From the left port, the memory is configured as 16K 16-bit words. For this organization, you might think that the slave dual-port RAMs in the second column from the right in *Figure 9* should be masters. If this were the case, however, you would have to defeat the arbitration logic in them when the right port addressed the same address; this would add logic, reduce the speed, and complicate the design. Therefore, this design uses a combination of left-bank decoding (LB, 1-of-4 decoder) and upper-lower 16-bit word decoding (UL, 1-of-8 decoder) to cause the bank master to arbitrate when the right port is

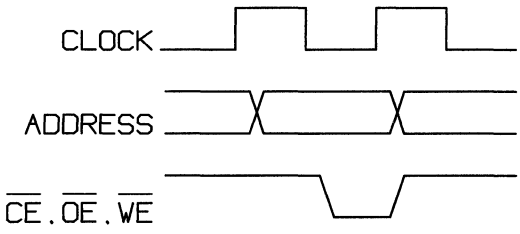


Figure 10. Timing for Dual-Port Example

addressing the same bank as the left port (more on this later).

Right-Port Operation

For purposes of this discussion, "word" refers to the 32-bit word at the right-port system-bus interface. At the 16-bit processor interface, the 32-bit word is referred to as either the lower half word (right-port bits 0 through 15) or the upper half-word (right-port bits 16 through 31).

The bank-selection process employs the chip enables. Specifically, the 1-of-4 RB decoder decodes the four combinations of the upper two right-port address-bus signals and generates four active-LOW chip enables to each bank of four dual-port RAMs. Bank A contains addresses 0 through 2047, bank B contains addresses 2048 through 4095, bank C contains addresses 4096 through 6143, and bank D contains addresses 6144 through 8191. In other words, bank A addresses 0 to 2K, bank B 2K to 4K, bank C 4K to 6K, and bank D 6K to 8K.

The lower 11 right-port address lines, AR(0:10), are connected to the A0 through A10 right-port address pins of all the dual-port RAMs.

Figure 10 does not show the generation of the write strobe, but does show the signal's timing. The write enable is applied directly to all the masters in parallel, then buffered, and then applied to all the slaves. The minimum propagation delay of the buffer must be at least as large as t_{BLA}, which is the time required for the master to assert the busy signal to the slaves after an address match occurs.

Note that all the right-port output-enable pins are connected together. These pins should be driven if reading is required; otherwise connect them to V_{CC}.

The open-drain busy outputs of the right port masters must be pulled up to V_{CC} using resistors. A value of 330Ω is recommended. The master busy outputs connect to all the right-port slave busy inputs for each bank.

For the data bus interface, the I/O pins of each RAM column connect to their respective I/O pins on each bank. This OR-tie connection is allowed because the bank-selection chip enable causes the output buffers of the unselected banks to go to the high-impedance state.

Left-Port Operation

The 1-of-4 decoder labeled LB performs bank selection for the left port. The upper two left-port address lines, AL13 and AL12, decode bank-select chip enable signals for the four masters only. Bank A corresponds to addresses 0 through 4095, bank B corresponds to addresses 4095 through 8191, bank C corresponds to addresses 8192 through 12,287, and bank D corresponds to addresses 12,288 through 16,383.

To perform upper and lower half-word selection, the 1-of-8 decoder labeled UL decodes the upper three right-port address signals. The decoder then generates eight chip enable signals with a resolution of 2048. The chip enables connect to the slaves' chip-enable and output enable pins (2048 resolution) and to the masters' output enable. Because the master chip enable resolution is 4096, the master arbitrates for two blocks of 2048 16-bit half words.

The lower eleven left-port address lines, AL(0:10), connect to left-port address pins A0 through A10 of all the dual-port RAMs.

At the 16-bit interface, writing is only required if the left port wishes to send a message to the right port. Otherwise, you can connect the left-port write pins of all the dual-port RAMs to V_{CC}.

To implement the left-port data bus interface, the left port's data I/O pins are connected together in the same manner as those of the right port for all RAMs in the same column. In addition, to multiplex

a 32-bit data word to a 16-bit half word, the least-significant bytes and the most-significant bytes of each 2048-word group are connected together. The UL decoder that controls the left-port output enable performs the selection.

If you use the masters' interrupt pins, pull them up to V_{CC} through a 330Ω resistor and connect them to the processor interrupt-request input. You can leave the slaves' interrupt pins unconnected.

If the control signal connections from their source to the dual-port memory constitute electrically long lines, they might require proper termination to avoid voltage reflections due to impedance mismatches. Refer to Cypress's application note titled "Systems Design Considerations When Using Cypress CMOS Circuits."

References

1. Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control." *CACM*, Vol 8, no.9, Sept. 1965, p 569.
2. Dijkstra, E.W., "Co-operating Sequential Processes." *Programming Languages*, F. Genyus (Ed.) Academic Press, New York, 1968, pp 43 – 112.

Notes

1. The Interrupt function is not available at the 2K x 8 level in a 48-pin package.

Understanding Large FIFOs

Introduction

This application note explains the internal operation of the large FIFOs manufactured by Cypress and shows how to use the devices to accomplish depth and width expansion. Other topics covered here include FIFO interfacing, the writing and reading process, failure modes, and typical problem symptoms and solutions. This information applies to the following Cypress FIFOs: CY7C419, CY7C420, CY7C421, CY7C424, CY7C425, CY7C428, CY7C429, CY7C432, CY7C433, CY7C439, CY7C460, CY7C462, CY7C464, CY7C470, CY7C472, and CY7C474.

Timing parameters given in this application note are taken from Cypress Semiconductor's *High Performance Data Book*.

Large FIFO Overview

The Cypress product line of large FIFOs include densities from 256 x 9 up to 32, 768 (32K) x 9,with the depth doubling (256, 512, 1K, 2K, 4K, 8K, 16K, 32K) between densities. These monolithic devices are available in a wide variety of packages with the industry standard pinout and with access times as fast as ten nanoseconds and cycle times as fast as twenty nanoseconds. Not all speed grades are available in all densities or all packages, so consult the Cypress databook to determine valid speed, density, package combinations. The smallest package available is the 32-lead 7mm x 7mm TQFP, which occupies less than one-third the area of a 300-mil-wide 28-pin DIP.

Although the first FIFOs utilized a shift-register type of architecture, today's large FIFOs employ an SRAM type of interface. Data is written into and read out of the devices, as with SRAM write and read operations. These operations can occur independently of one another and are made possible by a specially designed six-transistor, dual-ported SRAM cell. This cell makes use of separate read and write transistors to allow independent R/W operation.

Operating these FIFOs at their maximum throughput rates demands the generation of narrow write and read pulses. To facilitate significantly higher throughput rates, Cypress has developed the CY7C440 and CY7C450 families of clocked, or self-timed FIFOs.

These FIFOs feature 70-MHz operation and are characterized by self-timed interfaces. You generate the read and write enables, which are combined internally with the appropriate clocks. Thus, you do not need to generate narrow read and write pulses. These FIFOs also feature totally independent, asynchronous, read and write operations.

Each FIFO is organized such that data is read out in the same sequential order in which it was written. Full, half-full and empty flags facilitate writing and reading. Additional pins are provided to facilitate unlimited expansion in width and depth, with no performance penalty.

Writing to and Reading from the FIFO

Figure 1 shows the large FIFOs' read and write timing. Reads and writes are asynchronous to each other. The read process begins with $\overline{R}$'s falling edge. The output data bus, Q0 – Q8, leaves the high-impedance state t_{LZR} ns after $\overline{R}$'s falling edge. The output data becomes valid t_A ns after that same falling edge. This t_A period is referred to as the FIFO's read access time. $\overline{R}$'s rising edge ends the read process.

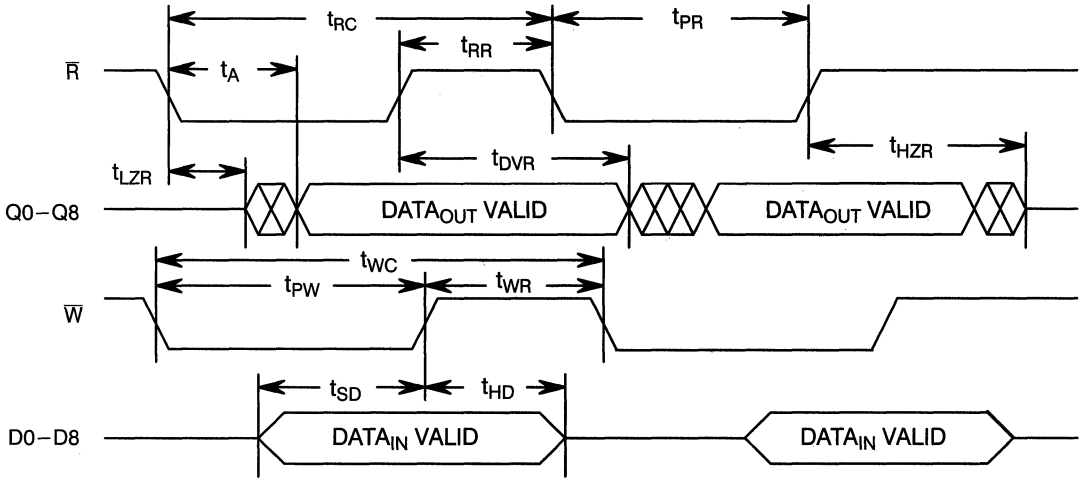


Figure 1. Asynchronous Read and Write Timing

The data on the Q0 – Q8 bus remains valid for t_{DVR} ns following the $\overline{R}$ rising edge. This is the output data hold time at the end of the read cycle. The internal circuitry then readies itself for the next read operation. This period is referred to as the t_{RR}, or read recovery time, and must be observed between consecutive read operations. The read signal's minimum pulse width is denoted by t_{PR} and is identical to the read access time, t_A.

You can determine the read cycle time (t_{RC}) by adding the access time (t_A) and the read recovery time (t_{RR}), which you can find in the FIFO data sheet. The maximum read frequency is the reciprocal of t_A + t_{RR}. For example, a Cypress FIFO with a 20-ns access time and a 10-ns read recovery time results in a 30-ns read cycle time, or 33.3-MHz maximum read cycle frequency.

The write process is similar to the read process. A write begins with the falling edge of the write line, $\overline{W}$, and terminates with $\overline{W}$'s rising edge. For a valid write to occur, the input data bus, D0 – D8, must be stable for t_{SD} ns prior to $\overline{W}$'s rising edge and for t_{HD} ns after this edge. These specifications are referred to as the data set-up and hold times, respectively. The write strobe also has a minimum negative pulse width, denoted as t_{PW}. A minimum recovery time, t_{WR}, is required between write cycles.

The maximum write frequency is the reciprocal of t_{PW} + t_{WR}. As an example, a device with a 20-ns write strobe width and a 10-ns write recovery time yields a 30-ns write cycle time, or a 33.3-MHz maximum write cycle frequency.

The FIFOs include separate write and read counters (pointers). Each write or read operation increments the appropriate counter one position. When the FIFO is empty, both counters point to the same location. The relative position of these counters determines the device's status, which is indicated externally via empty, half-full, and full flags.

Applications

FIFOs are asynchronous devices that are ideal for interfacing between two asynchronous processes. A FIFO allows two systems running at different data rates to communicate by providing a temporary data or control buffer.

Typical FIFO applications include

- Interprocessor communications, in which bidirectional devices are especially useful

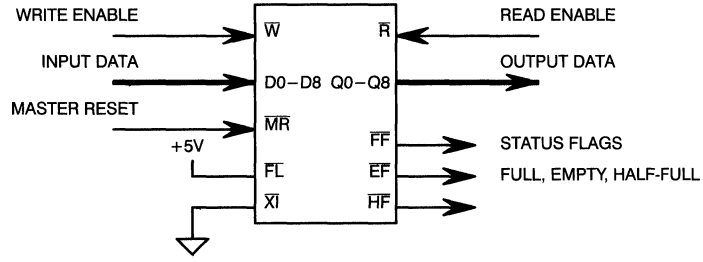


Figure 2. Standalone Operation

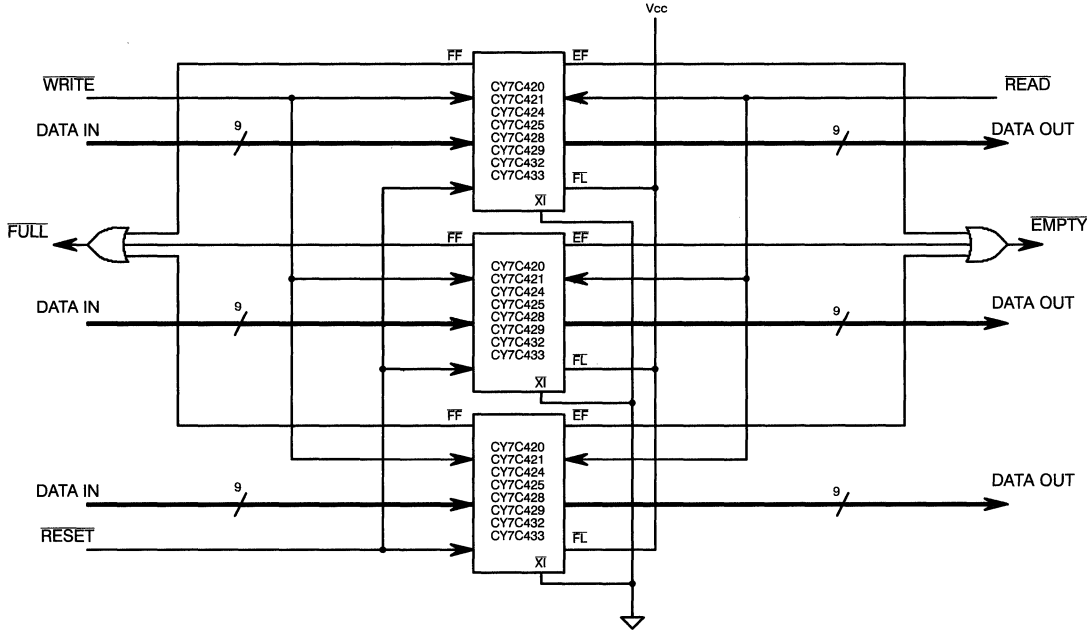


Figure 3. Width Expansion

- Communications systems, including local area networks
- Digital-signal-processing-based systems for buffering real-time data
- Electronic data processing, CPU, and peripheral equipment, including high-performance disk controllers

Common FIFO Configurations

All large FIFOs can be interconnected, without external logic, to create either wider FIFOs, deeper FIFOs, or both. Standalone operation, width expansion, depth expansion, and design considerations are described next.

Figure 2 illustrates standalone mode, and *Figure 3* shows width expansion mode. In both these modes,

the $\overline{XI}$ (expansion in) pin is grounded and the FL (first load) pin is tied HIGH.

The OR gates in the width-expansion design generate composite full, half-full, and empty flags (F, HF, E). Composite flags are necessary because variations in propagation delays might prevent the individual FIFOs in the design from entering the F, HF, or E states simultaneously. A composite flag properly reflects the instantaneous status of the entire word.

Figure 4 illustrates depth expansion. The $\overline{FL}$ (first load) pin on one device must be grounded to define that FIFO as the first FIFO to be written to. The FIFOs are then daisy-chained together by connecting one device's $\overline{XO}$ (expansion out) output pin to the next device's $\overline{XI}$ (expansion in) input. The $\overline{XO}$ of the last device in the chain is connected to the $\overline{XI}$ of the first device, thus forming a token-passing ring.

Token passing allows the writing and reading processes to stay consistent. That is, the passing and holding of a read or write token tells an individual FIFO whether it is actively being read from or written to. In the token-passing procedure for write operations, the first FIFO is written to until it is filled. An internal write pointer determines the location written to, and after every write, the pointer is incremented. When the pointer reaches the last physical location, no more writes can occur to that device. At that point, the first FIFO passes the write token to the next FIFO in the chain via the $\overline{XO}-\overline{XI}$ interface. The second device, now in possession of the write token, receives all future written data until this device also fills up and passes the write token onto the next device in the chain.

If enough writes occur to fill up the FIFO chain, the last device fails in its attempt to pass the write token back to the first device. This is because the full FIFO cannot accept a write token. No further writes to the FIFO chain are allowed until a read operation occurs, which frees up an internal location. The relative positions of the internal write and read count-

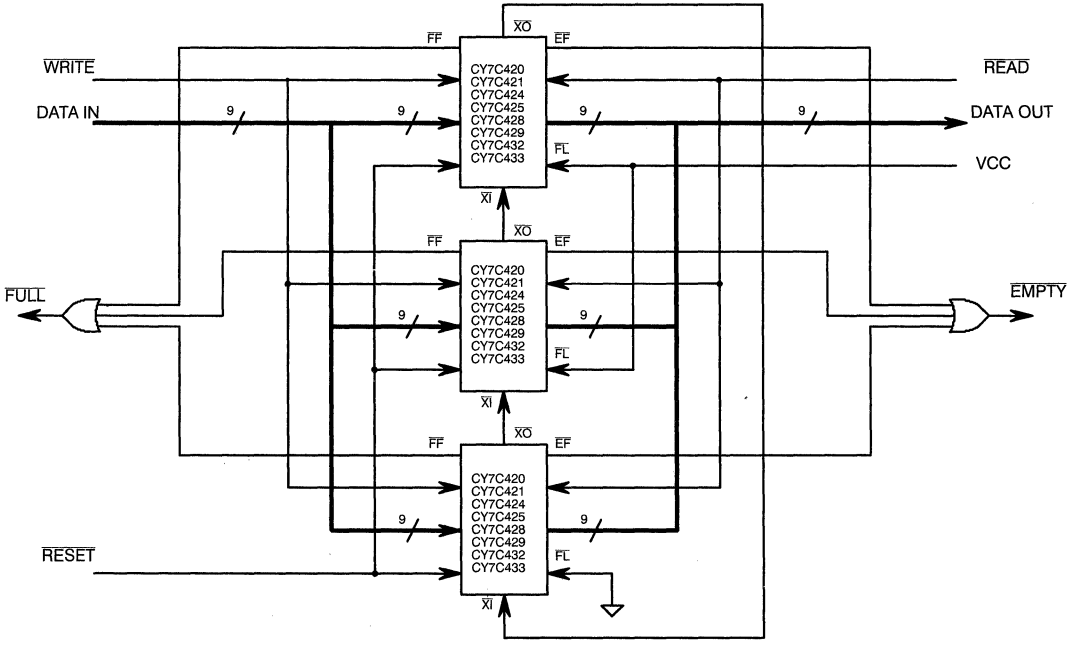


Figure 4. Depth Expansion

ers determine a device's status and whether it can accept data though a write operation. *Figure 5* shows the timing for write operations.

As with the procedure for writes, the first FIFO in the chain holds the read token. When the FIFO chain is read from, the device holding the read token supplies the data from the address specified by the device's read pointer. The read pointer is then incremented. The incrementing continues until the FIFO is empty, and the read token is passed to the next device in the chain. The passing of the read token is done via the $\overline{XO}-\overline{XI}$ interface. *Figure 6* shows the timing for read operations.

A depth-expansion design must generate composite status flags to adequately reflect the instantaneous state of the FIFO chain, as is done for width expansion.

Retransmit

The retransmit feature is useful in communications for retransmitting packets of data and in disk drives for rewriting sectors. It is especially useful in ap-

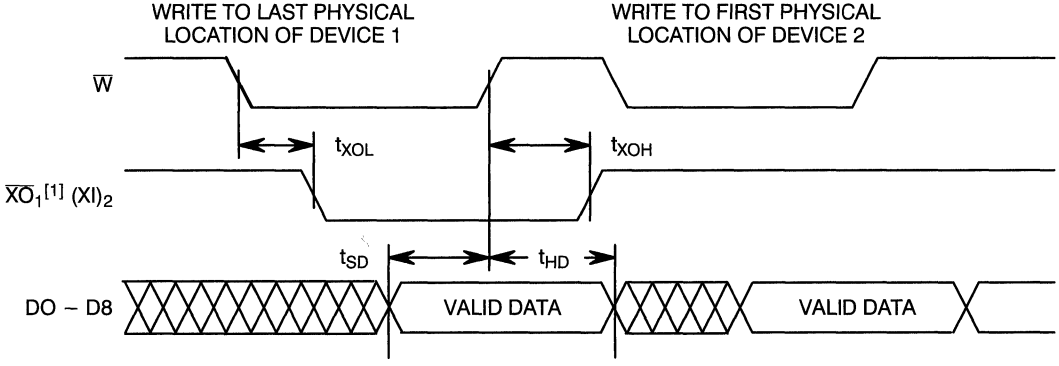


Figure 5. Write Expansion Timing

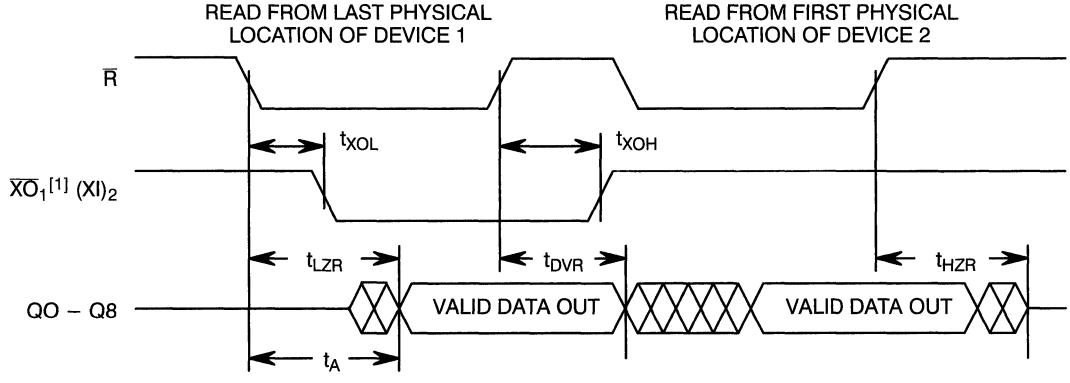


Figure 6. Read Expansion Timing

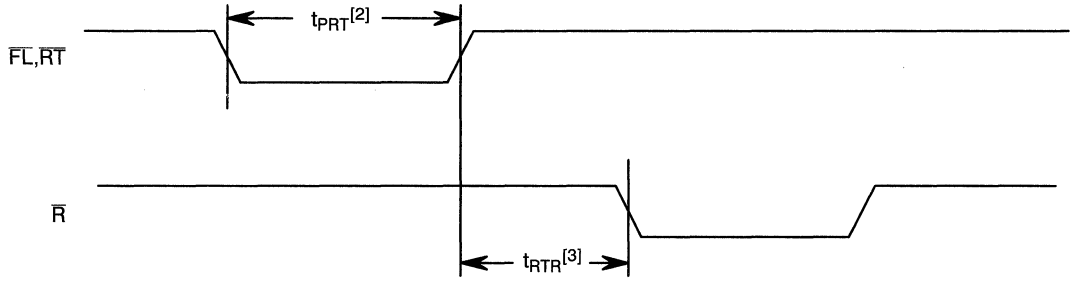


Figure 7. Retransmit Timing

plications where a single block of data in the FIFO must be sent out multiple times, as in a word or pattern generator.

Data can be retransmitted any number of times, and with Cypress FIFOs, the retransmit feature can be used at any time, no matter how much data the FIFO contains. This is in contrast to some competing FIFOs, such those from IDT, which do not allow use of the retransmit function when the FIFO is full.

In the retransmit operation, the read pointer is reset to its initial location and the $\overline{R}$ pin is pulsed until the read pointer advances to the same memory location addressed by the write pointer. The retransmit ($\overline{RT}$) pin is available in the single-device and width-expansion modes, but not in depth expansion because this pin designates the FIFO to be loaded first.

The retransmit function is initiated by asserting an active-LOW pulse to the retransmit input, which resets the internal read counter to zero. Keep the $\overline{R}$ input inactive during this time; otherwise, the conflicting requirements on the read counter might cause it to become corrupted. The retransmit process does not affect the state of the write counter or the write process, though the retransmit timing constraints shown in *Figure 7* must not be violated.

Note that the architectural description in the 1990 and previous Cypress data books incorrectly stated that the $\overline{W}$ input must be inactive during a retransmit cycle. No design or usage rules are violated if retransmit and write cycles overlap or occur simultaneously; the device does not lock up, and data is neither lost nor corrupted.

The reasons for the data book's retransmit/write restriction are more historical and application-oriented than functional. Specifically, the first large FIFOs did not permit writes during a retransmit cycle. This set a documentation precedent that all future devices had to match.

Additionally, keeping track of what data is currently in the FIFO and what data is being read out can become complicated. For example, if a FIFO is half full and the retransmit function is activated and writes continue, filling the FIFO to three quarters full before the read pointer catches up with the write pointer, the FIFO outputs all of the data.

Common Problems and Solutions

To help prevent problems and correct them when they occur, this section describes the causes and solutions to some common FIFO problems. The first problem to consider is corrupted or repetitive data in a FIFO.

Corrupted or Repetitive Data

The most common cause of corrupted and repetitive data being present in a FIFO is a spurious active signal (glitch) on the FIFO's $\overline{W}$ input. Because Cypress devices are extremely fast, a write pulse as short as 3 ns initiates a write. Write glitches cause whatever logic levels are present at the data inputs to be written into the FIFO, which can put false data into the device. If valid data is present at the data

inputs, a write glitch causes this data to be written a second time, resulting in duplicated data.

Write glitches are often the result of voltage reflections due to impedance mismatches, which you can eliminate using impedance-matching termination networks. Termination networks are recommended on the $\overline{W}$ and $\overline{R}$ traces on printed circuit boards (PCBs) when the lines exceed approximately 4 inches from source to a single load. This line length assumes a 2-ns rise/fall time for the read and write strobes. For $\overline{R}$ and $\overline{W}$ signals with sub-2-ns rise/fall times, line lengths as short as 1 inch might require termination.

A termination network matches the load impedance to the PCB trace's characteristic impedance, which is typically 50Ω or less for microstrip or stripline construction on G−10 glass epoxy material. To minimize voltage reflections, a slightly overdamped termination is preferred. Cypress recommends a 47-pF (max.) series capacitor and a 47-ohm resistor be connected from the read or write pin to ground (*Figure 8*). This termination network acts as a high-pass filter to short, high-frequency pulses and dissipates no DC power. Read or write lines that drive more than one FIFO require only one termination network. Put the network at the input that is electrically farthest from the source. For multiple loads, see the "Systems Design Considerations When Using Cypress CMOS Circuits" application note for help in determining the maximum line length.

FIFO data corruption can also be caused by violation of master-reset timing constraints. As shown in

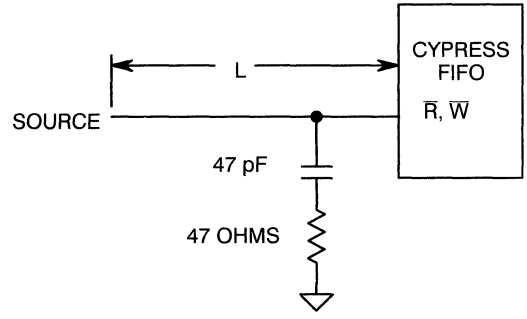


Figure 8. Recommended Termination Network

the timing diagram in *Figure 9*, the read and write signals must be inactive around the rising edge of $\overline{MR}$ (master reset) to satisfy the t_{RMR}, or master-reset recovery-time specification. This constraint is necessary because the FIFO goes through an internal initialization process during reset and requires a settling period after the reset terminates.

FIFO Locks Up

Short noise pulses on the FIFO's master reset pin can cause the FIFO to not respond because it is "partially reset." If this problem occurs, you need to terminate the master reset line.

Missing or Disappearing Data

Glitches on the $\overline{R}$ input can cause data to disappear because of an unintended read operation. The read increments the internal read counter, resulting in

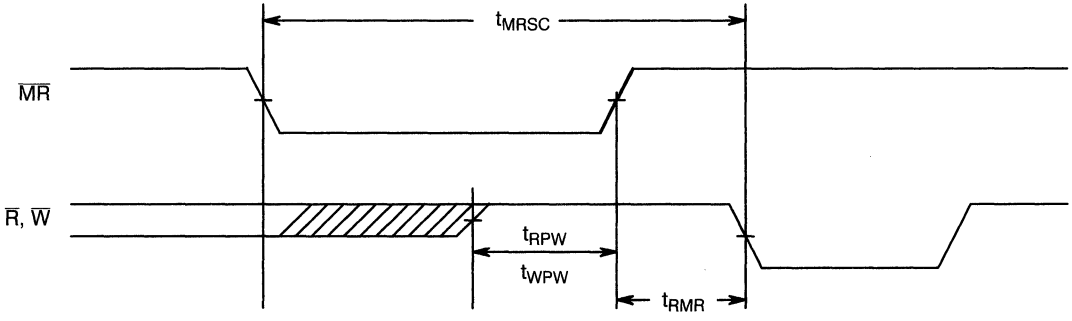


Figure 9. Master Reset Timing

the loss of the current data word. Here again, a termination network eliminates the unwanted glitches.

Repetitive or Out-of-Sequence Data, False Full or Empty

A misaligned internal read or write pointer can cause a variety of symptoms, including repetitive or out-of-sequence data and false full and/or empty conditions. The two most common causes of misaligned pointers are master-reset violations and boundary-condition violations.

Boundary conditions are defined as the FIFO being either full or empty. When high-density FIFOs are connected in parallel to make a wider word, certain conditions can cause the FIFOs to choose individually to either ignore or act upon a read or write request. The system-level symptom of individual FIFOs making different decisions is word misalignment. The problem occurs in the empty condition when a read immediately follows a write and in the full condition when a write immediately follows a read.

Operation at the Empty Boundary

Consider a FIFO that has been reset and is empty. The empty flag is active (LOW), and internal logic inhibits read operations. In the general case, the read and write signals are asynchronous. Upon completion of the write operation the internal state of the FIFO goes from empty to empty + 1. During this interval, a read operation might or might not be recognized. A read preceding the write is ignored; a read following the write is not. In between these conditions, the FIFO decides whether to recognize the read. During this aperture of uncertainty, it cannot be determined whether the read will be ignored or not. With one FIFO, this uncertainty is acceptable. However, if two or more FIFOs are connected in parallel to make a wider word, some might ignore the read, and others might not.

Operation at the Full Boundary

A similar condition occurs when a single FIFO becomes full. The full flag is active (LOW), and internal logic inhibits write operations. A read operation immediately followed by a write operation causes the FIFO to go from full to full − 1 and back to full. During the time the FIFO is going from full to full − 1, a write operation might or might not be recognized. The aperture of uncertainty applies here because the FIFO takes a finite amount of time to change states, and a write command arriving at this instant might be ignored.

Waiting at the Empty Boundary

Figure 10 shows the timing that prevents problems with reads at the empty boundary. Any device reading from the FIFO must wait an amount of time, t_{RAE}, after the termination of the write operation before causing a HIGH-to-LOW transition of the $\overline{R}$ signal. The $\overline{W}$ signal's rising edge indicates the termination of the write operation.

One way to satisfy this timing is to gate read operations with the composite empty flag ($\overline{EF}$) such that the read operation is prevented when the empty flag is active. Note, however, that the $\overline{R}$ signal can be LOW either before or during the first write to the empty FIFO and the data still propagates to the outputs correctly.

Waiting at the Full Boundary

Figure 11 shows the timing that prevents problems with writes at the full boundary. Any device writing to the FIFO must wait an amount of time, t_{WAF}, after the termination of the read operation before causing a HIGH-to-LOW transition of the $\overline{W}$ signal. The $\overline{R}$ signal's rising edge indicates the end of the read operation.

You can meet this timing by gating write operations with the composite full flag ($\overline{FF}$) such that the write operation is prevented when the full flag is active. However, the $\overline{W}$ signal can be LOW either before or during the first read from a full FIFO and the data is still properly written.

Empty Reads and Full Writes

When Cypress FIFOs are empty, their data outputs go to the high-impedance state. Therefore, attempting to read from an empty FIFO yields unpredictable data. Internal logic inhibits the read, and the read pointer is not incremented.

Internal logic also inhibits attempts to write to a full FIFO, and the write pointer is not incremented.

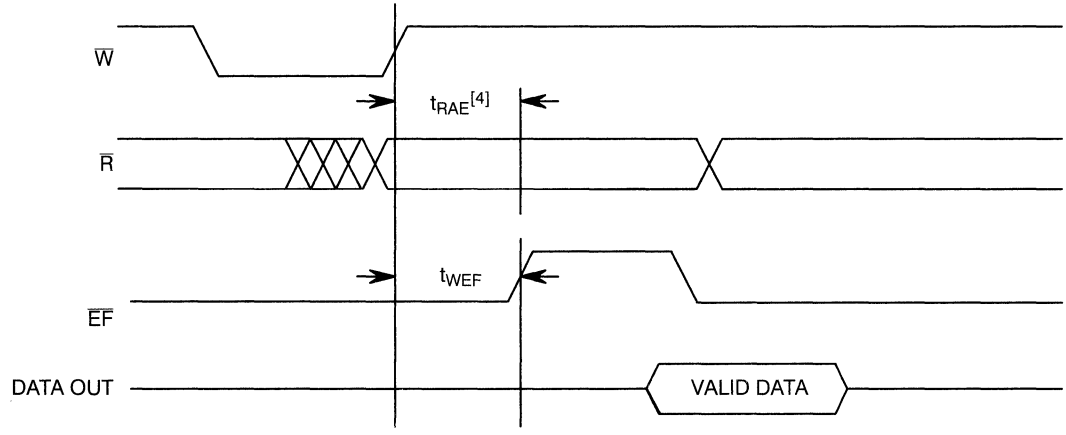


Figure 10. Read Fall-Through Timing Violation

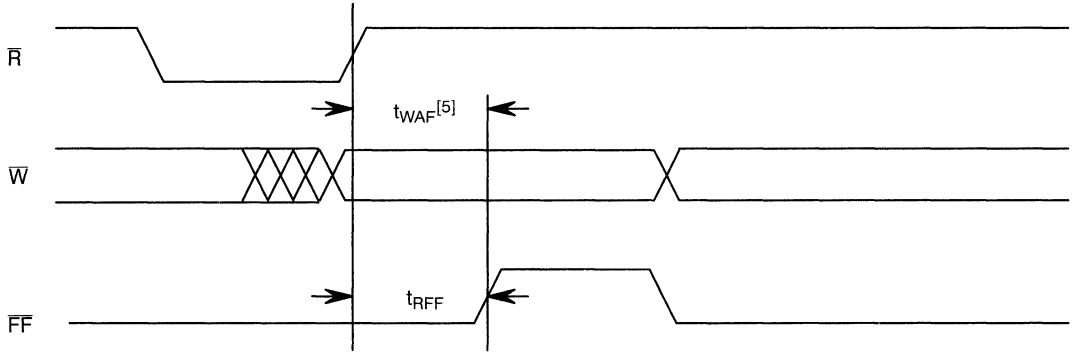


Figure 11. Write Bubble-Through Timing Violation

Effective Pulse Width Violation

This phenomenon can occur at either the empty or the full boundary if the flags are not properly used. The empty flag must be used to prevent reading from an empty FIFO and the full flag must be used to prevent writing into a full FIFO. Otherwise, the effective pulse width of the read or the write strobe will be violated, even though the actual signals meet the data sheet specifications.

Consider an empty FIFO that is receiving read pulses. Because the FIFO is empty, the read pulses are ignored, and nothing happens. Next, a single word is written into the FIFO, with a signal that is asynchronous to the read pulses, while the read pulses continue. The internal state machine in the FIFO goes from empty to empty + 1 shortly after the rising edge of the write pulse. However, it does this asynchronously with respect to the read pulse, and it does not look at the read signal until it enters the empty + 1 state. If the rising edge of the write signal occurs slightly before the rising edge of the read signal an effective minimum LOW read pulse width violation will occur.

In a similar manner, the minimum write pulse width width may be violated by attempting to write into a

full FIFO and asynchronously performing a read. The empty and full flags must be used to avoid these effective pulse width violations.

Intermittent Malfunctions

If all the timing requirements appear to be met and data in the FIFO is still corrupted, the cause is likely to be noise on the power supply. Random spikes on either the V_{CC} or ground pins of the FIFO are likely culprits when non-repeatable failures occur.

The cure for this problem is to add a high-pass filter capacitor between the device's power and ground pins. This practice is recommended whenever the read or write frequency exceeds 5 MHz. Use a very small (100 – 500 pF) ceramic or mica capacitor. Surface-mounted capacitors are recommended because they have at least an order of magnitude less lead inductance than radial or axial leaded capacitors.

The filter capacitor is in addition to the 0.1- or 0.01-μF decoupling capacitor that should always be present with any high-speed digital chip. Although decoupling capacitors are often referred to as bypass capacitors—implying filtering properties—their true function is to supply the instantaneous current required when many or all device outputs simultaneously switch from LOW to HIGH. This larger capacitor thus decouples or isolates the IC from the power distribution system.

Notes

1. Expansion out of device 1 (XO_1) is connected to expansion in of device 2 (XI_2).
2. t_{PRT} is the minimum retransmit pulse width.
3. t_{RTR} is the retransmit recovery time. It is a timing window that must not be violated.
4. t_{RAE} is an invalid read window. A read operation should never be initiated inside this window.
5. t_{WAF} is an invalid write window. A write operation should never be initiated inside this window.

Understanding Clocked FIFOs

Introduction

This application note explains the basic operations and features of Cypress clocked FIFO memories. Cypress clocked FIFOs are ideally suited for applications requiring high data throughput and asynchronous data buffering. The clocked FIFO interface simplifies high-speed design and provides greater noise immunity over industry-standard asynchronous FIFOs.

Design considerations of the clocked FIFO architecture are examined, including proper flag operation and decoding, FIFO boundary operation, and resetting and programming the FIFO. FIFO depth and width expansion are also covered.

The Cypress family of clocked FIFOs are available in several densities with a variety of features. *Table 1* outlines the features of Cypress's clocked FIFOs. The entire clocked FIFO family feature fully asynchronous operation at clock rates of up to 70 MHz in non-depth expansion mode. Clocked FIFOs cascaded for depth expansion can operate at frequencies of up to 50 MHz.

The CY7C441 and CY7C443 feature 512 and 2K word by 9 bit memory arrays, respectively. These FIFOs feature high-speed operation and $\overline{\text{Empty}}$, $\overline{\text{AlmostEmpty}}$, and $\overline{\text{AlmostFull}}$ flags, center power and ground pins, and width expandability. Both FIFOs are available in either a 32-pin PLCC/LCC package or a 28-pin DIP package.

The CY7C451 and CY7C453 clocked FIFOs have all of the features of the 7C44X FIFOs plus $\overline{\text{Full}}$ and $\overline{\text{HalfFull}}$ flags, programmable $\overline{\text{AlmostEmpty}}$ and $\overline{\text{AlmostFull}}$ flags, parity generation and parity checking, output enable ($\overline{\text{OE}}$), and depth expandability. The 7C451 features a 512 word by 9 bit memory array and the 7C453 features a 2K word by 9 bit memory array. Both FIFOs are available in either a 32-pin PLCC/LCC package or a 32-pin DIP package.

Table 1. Features of Cypress Clocked FIFOs

FIFO	Density	Speed	Flag Architecture	Parity	Output Enable	Depth Expandable	Width Expandable
7C441	512 x 9	71.4 MHz	Synchronous	No	No	No	Yes
7C443	2048 x 9	71.4 MHz	Synchronous	No	No	No	Yes
7C451	512 x 9	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes
7C453	2048 x 9	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes
7C455	512 x 18	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes
7C456	1024 x 18	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes
7C457	2048 x 18	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes

* 50 MHz in this mode

Clocked Architecture

The clocked FIFO architecture is designed to achieve maximum performance from FIFO memories while simplifying their use in a system. Timing pulses for the memory array are generated internally from the read and write clocks thus eliminating the need for generating very narrow external read and write pulses.

The read and write ports have separate clock inputs (CKR, CKW), and read and write operations are enabled through separate clock-enable pins ($\overline{ENR}$, $\overline{ENW}$). The read and write clocks can be fully asynchronous. *Figure 1* demonstrates asynchronous reading and writing to a clocked FIFO.

The clocked FIFO interface is ideally suited for state machine control. A state machine can perform reads or writes by simply asserting the respective enable lines LOW. It is not necessary to toggle the enable lines to perform consecutive operations.

FIFO Writes

Figure 2 shows a simplified block diagram of the clocked FIFO data path. The internal write control logic circuitry controls the input register, the write pointer, and the write port of the dual-ported memory array.

This write operation is similar to writing to a standard 377 register. The FIFO input register is clocked by CKW and enabled by $\overline{ENW}$. Data is clocked into the FIFO on the enabled rising edge of CKW. The data is then written into the memory location pointed to by the write pointer, provided the FIFO is not full ($\overline{Full}$ = 1). The write pointer is then incremented. A full FIFO will ignore any attempted write without upsetting the memory array or the flags. The 70-MHz clocked FIFOs have a data and enable set-up time (t_{SD} and t_{SEN}) of 7 ns.

FIFO Reads

The internal read control logic circuitry controls the output register, the read pointer, and the read port of the dual-ported memory array. The output register holds the word that was last read from the FIFO memory array. This register is loaded from the memory array in a manner similar to loading a standard 377 register. The output register is clocked by CKR and enabled by $\overline{ENR}$. Note that the CY7C45X family of clocked FIFOs feature a three-state output register controlled by $\overline{OE}$.

The read pointer points to a word in the memory array. That word is loaded into the output register

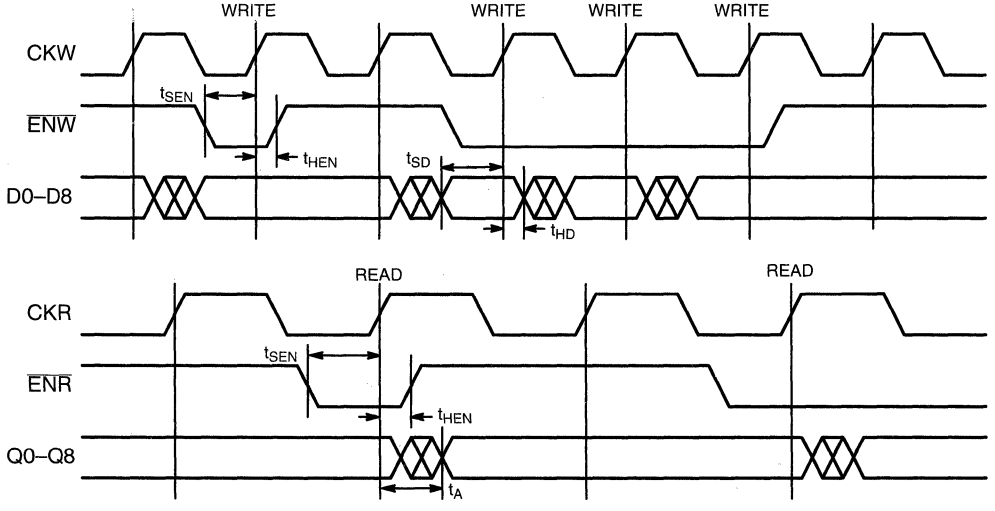


Figure 1. Asynchronous Writing and Reading to a Clocked FIFO

on the enabled rising edge of CKR, provided the FIFO is not empty ($\overline{Empty}$ = 1), and the read pointer is then incremented. The word is available at the output pins t_A after the clock edge. An empty FIFO will ignore the attempted read and continue to hold the last word in its output register. The set-up time for $\overline{ENR}$ (t_{SEN}) is 7 ns and the data access time (t_A) is 10 ns for a 70-MHz clocked FIFO.

Flag Architecture

Cypress clocked FIFOs feature a synchronous encoded flag architecture that simplifies FIFO integration into a synchronous system. Synchronous flags guarantee that a flag update is only triggered by a rising clock edge. The state of a flag is guaranteed to be valid t_{FD} after the rising clock edge.

Unclocked asynchronous FIFOs can generate narrow flag pulses with indeterminate timing based on the timing relationship of read and write pulses. External flag synchronization logic is required in synchronous designs using unclocked FIFOs. The Clocked FIFO architecture eliminates these short flag pulses and avoids the need for external flag synchronization logic.

A small package footprint is maintained by encoding the state of the flags. Pin count and package size are reduced and fewer PCB board signals require routing. Only two signals are needed to encode four states of the 7C44X FIFOs and three signals encode six states of the 7C45X FIFOs.

The FIFO flags are easily decoded inside a programmable control unit or a state machine controller. Decoding the signals properly produces flags synchronized to a single clock. *Figures 3* and *4* show a block diagram of the flag architecture for both the 7C44X and 7C45X FIFOs. The diagrams also show the external logic needed to decode and synchronize the flags.

The decoded Empty-type flags are synchronized to the read clock (CKR) and decoded Full-type flags are synchronized to the write clock (CKW). The CY7C45X family of Clocked FIFOs features a Programmable Almost Full/Empty flag ($\overline{PAFE}$) that is synchronized to the read and write clocks.

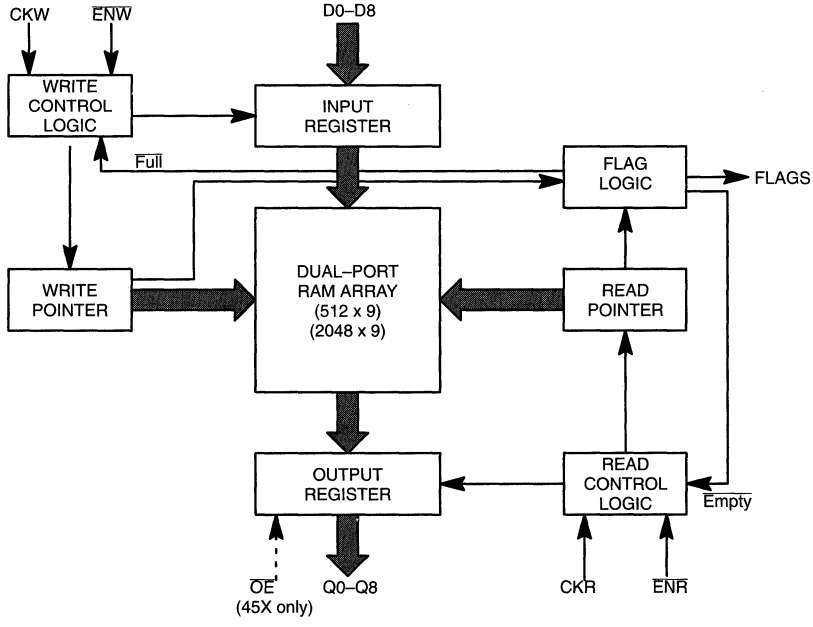


Figure 2. Clocked FIFO Data Path

Reads and Writes with Boundary Flags

The Empty and Full flags are considered Boundary flags because they indicate that the FIFO has reached its boundary of operation. Attention must be paid to the status of these flags when operating the FIFO at or near the boundaries. The internal FIFO write and read control logic uses the Boundary flags to determine if an access to the memory array is possible. The internal write control logic will not attempt to write to the memory array or increment the write pointer if the FIFO is Full, as indicated by the registered $\overline{\text{Full}}$ flag. Similarly, the read control logic will not load the output register or increment the Read Pointer if the FIFO is empty, as indicated by the registered $\overline{\text{Empty}}$ flag (see *Figures 2, 3,* and *4*).

The boundary flags determine the state of the read and write logic control circuits inside the Clocked FIFO. Design considerations with boundary flags are explored in the next two sections.

Boundary Latency Cycles

A write or a read can cause the FIFO memory array to exit from an empty- or full-boundary condition. At the empty boundary, the FIFO write control logic will allow an enabled write clock to store a word in the memory array. However, the $\overline{\text{Empty}}$ flag synchronization register will not reflect the current state of the FIFO memory array until it is clocked by the read clock. Similarly, at the full boundary, the FIFO read control logic will allow an enabled read clock to remove a word from the memory array, but the $\overline{\text{Full}}$ flag synchronization register will not reflect the current state of the FIFO until it is clocked by write clock.

A FIFO latency cycle (update cycle) refers to the clock cycle that causes a boundary flag register to be updated with the current status of the memory array. During this cycle, only a boundary flag regis-

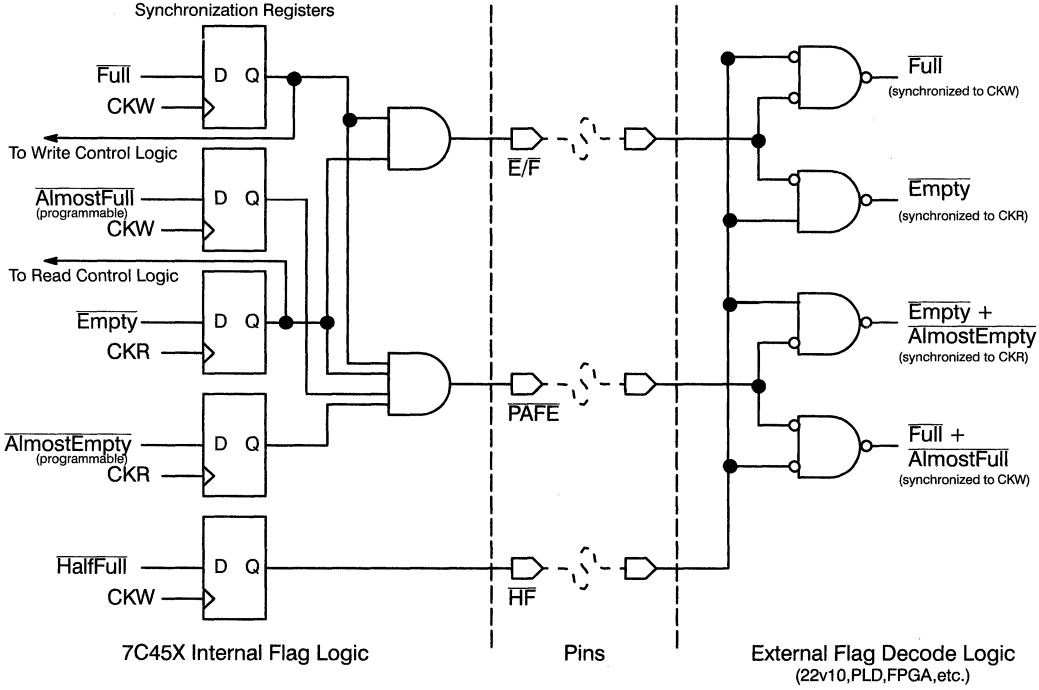


Figure 3. 7C45X Flag Architecture

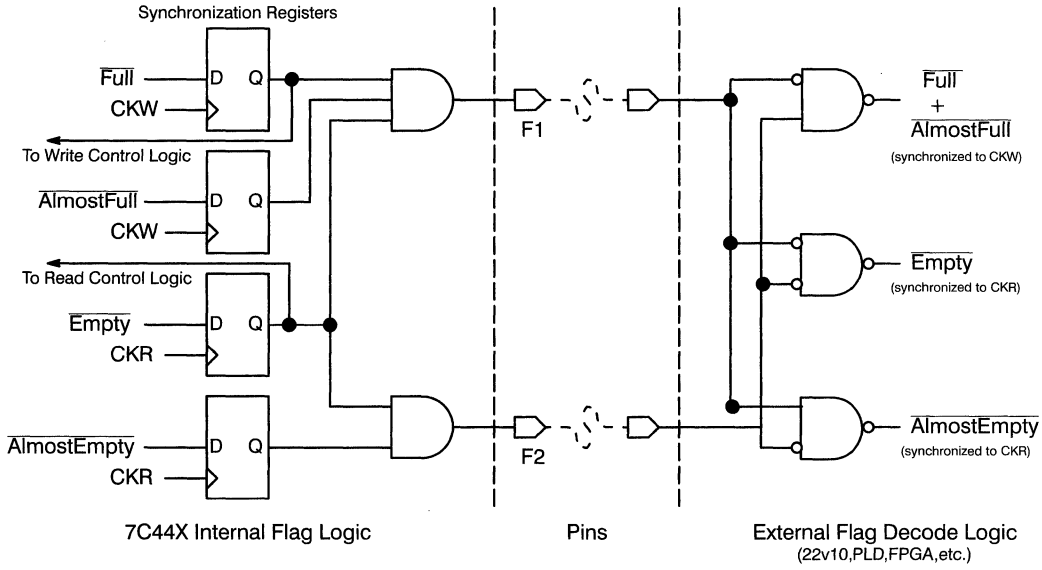


Figure 4. 7C44X Flag Architecture

ter is updated regardless of the state of $\overline{ENR}$ or $\overline{ENW}$. A read-clock latency cycle updates the $\overline{Empty}$ flag register from LOW to HIGH regardless of the state of $\overline{ENR}$. When the $\overline{Empty}$ flag register is in the HIGH state, an enabled read clock can retrieve data from the memory array. The overall effect is that after the FIFO memory becomes non-empty, it takes two read cycles to get the first word from the FIFO—one to update the flag and one to read the data.

Similarly, a write clock latency cycle updates the $\overline{Full}$ flag register from LOW to HIGH regardless of the state of $\overline{ENW}$. When the $\overline{Full}$ flag register is in the HIGH state, an enabled write clock can store data in the memory array. The overall result is that after the FIFO memory becomes non-full, it takes two write cycles to put the first word in the FIFO.

This type of flag operation is desirable because it guarantees that flags in the inactive (HIGH) state will be valid and usable for at least one clock cycle. This architecture eliminates indeterminate short flag pulses characteristic of asynchronous flag architectures.

Free-Running CKR and CKW Clocks

Boundary-operation timing and latency cycles should pose no problem in designs that employ free-running read and write clocks. Free-running clocks insure that flag update cycles will be performed automatically. The flag registers will be constantly updated with the current FIFO status.

Designs that do not use free-running clocks must explicitly issue a clock cycle near the FIFO boundaries in order to update the flag registers. Absence of free-running clocks may decrease system performance by causing the external control circuitry to wait for one clock cycle during the flag update cycle before performing an operation.

Resetting and Programming Clocked FIFOs

Master Reset

Clocked FIFOs are reset by pulsing the $\overline{MR}$ (Master Reset) pin LOW. Resetting the FIFO clears the read and write pointers so that they both point to location zero of the memory array, causing the FIFO to be Empty. The data output register will contain all 0s after the reset pulse occurs. Master Reset also resets the internal read and write control

logic circuits. The 7C45X family of clocked FIFOs can also be programmed during Master Reset. Programming the FIFO causes the program word to be stored in the FIFO program register.

Clocked FIFOs generate internal timing pulses off of the falling edge of $\overline{MR}$ in order to reset and program the internal FIFO control logic. For this reason, it is very important that the assertion of $\overline{MR}$ be glitch free. A narrow glitch of only a few nanoseconds while $\overline{MR}$ is LOW can be interpreted as a false edge and interrupt the reset timing sequence. As a result, the FIFO will not be fully reset or programmed.

To insure that Master Reset is glitch free, it is recommended that $\overline{MR}$ be driven by a flip-flop. In applications requiring a single Master Reset signal to reset or program multiple FIFOs, the FIFO pin farthest way from the flip-flop may need to be terminated in order to reduce glitches caused by voltage reflections. The need for terminations is a function of trace length, rise time, and PCB characteristics (see "System Design Considerations When Using Cypress CMOS Circuits," in the *Cypress Semiconductor Applications Handbook*).

The probability of improperly resetting a clocked FIFO due to glitches induced by ground bounce or other sources of noise can be reduced by using a Master Reset pulse that is as short as possible but is greater than t_{PMR}. Long reset pulses increase the chance that noise from somewhere in the system will be coupled to the $\overline{MR}$ pin through the ground plane. *Figure 5* shows a circuit for creating a short $\overline{MR}$ pulse from a long reset pulse. The duration of the $\overline{MR}$ pulse can be increased by adding more delay registers before the AND gate.

The proper reset sequence requires that enabled read and write cycles not be performed during or near the Master Reset pulse. Clock cycles that are not enabled by $\overline{ENR}$ or $\overline{ENW}$ are allowed during Master Reset. To insure that the clocks are disabled, $\overline{ENR}$ and $\overline{ENW}$ should not glitch LOW. Exact timing parameters are given in the data sheet. An easy way to insure that timing restrictions are met with a state machine is to insert pad states (clock enables HIGH) between the last read and write before Master Reset and between the first read and write after Master Reset.

Programming the 7C45X

The 7C45X family of clocked FIFOs can be programmed during the Master Reset cycle. Programming affects the $\overline{AlmostEmpty}$ and $\overline{AlmostFull}$ flags and sets the Parity. Programming is accomplished by writing data to the FIFO while asserting $\overline{MR}$ LOW. The program word is stored in the program register. The programming information may be ver-

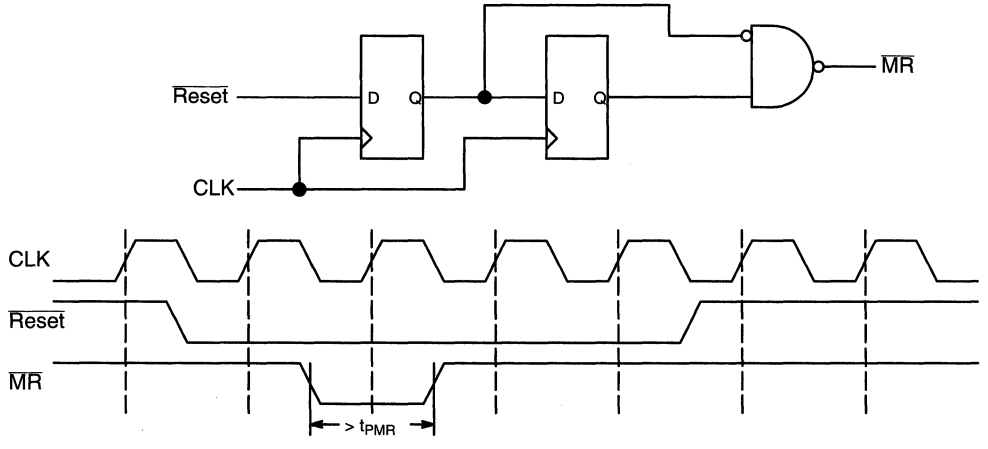


Figure 5. $\overline{MR}$ Pulse Generation

ified my reading the FIFO while $\overline{MR}$ is still asserted LOW. The FIFO program register is programmed to its default value if no write is performed during a Master Reset.

Data lines D0–D5 are are used to program the $\overline{AlmostEmpty}$ and $\overline{AlmostFull}$ flags. The value of D0–D5, which is written into the program register, determines the distance from the FIFO boundary flags ($\overline{Empty}$ and $\overline{Full}$) that these flags become active. The distance is programmable in 16-word increments and is determined by 16•P where P is the value of D0–D5. The $\overline{PAFE}$ pin encodes the programmable flag states.

Data lines D6–D8 program the FIFO parity option. D8 enables the Parity feature when set HIGH. D7 selects between Parity Generation and Parity Checking. Parity Generation is selected when D7 is LOW. D6 selects even parity when set LOW and odd parity when set HIGH.

Parity generation provides a simple means for systems to detect data bit errors. When enabled, the FIFO parity checker will examine bits D0–D7 being written into the FIFO before writing them into the memory array. The ninth bit (D8) will be set according to the parity mode set in the program register. Even-parity mode will set D8 such that the sum off all the bits including D8 is even. Odd-parity mode will set D8 such that the sum is odd. D8 is available on output line Q8/PG/$\overline{PE}$ during a read from the FIFO. Parity checkers down stream in the system can use D8 to determine when data has been corrupted.

The 7C45X can be configured as a parity error checker. During a write, data bits D0–D8 are examined before being stored in the memory array. D8 is set LOW if a parity error is detected. When set for for even parity checking, a parity error occurs if bits D0–D7 add to an odd number. Odd-parity checking will detect an error if D0–D8 add to an even number. D8 is written into the memory array with the rest of bits D0–D8. The parity-error bit (D8) is then available on Q8/PG/$\overline{PE}$ during a read from the FIFO.

Depth Expansion

The 7C45X Family of Clocked FIFOs feature depth expandability. Two or more 7C45Xs may be cascaded to achieve a single, large FIFO memory array. Depth expansion may be used in applications requiring buffering of large data packets, using extremely disparate read and write rates, or having long read latencies.

Depth expansion is achieved by cascading several FIFOs using the expansion pins. Data is automatically multiplexed from the FIFOs onto a single output bus using the FIFO's three-state output drivers. The flags must be combined to form composite flags. *Figure 6* shows two FIFOs cascaded for depth expansion.

The cascaded devices act as a single FIFO memory array. Read and write control is passed from one FIFO to another using the expansion pins. When a single FIFO has had all of its memory locations written to, it asserts the Expansion Out pin ($\overline{XO}$) signaling the next FIFO to begin writing to its array. Similarly, when the FIFO has had all of its memory locations read from, it deasserts the Expansion Out pin to signal the next FIFO to read data from its array. The FIFOs' expansion pins form a simple token ring.

The token-passing architecture necessitates the use of composite flags in order to detect when composite FIFO is in a boundary state (Full or Empty). In a long series of reads and writes, it is difficult to track which of the individual FIFOs possess the read and write tokens. The state of the composite FIFO could be determined by looking at the flags of the FIFOs in possession of these tokens, but this is difficult and unnecessary. Composite flags, shown in *Figure 6*, bypass this problem by looking at all the flags in parallel.

The First Load pin ($\overline{FL}$) indicates which device possesses the read and write tokens following a Master Reset. Only one device should have its $\overline{FL}$ pin tied to V_{SS}. All other devices should tie $\overline{FL}$ to V_{CC}.

The Almost Empty and Almost Full flags are not usable in depth expansion. The cascaded devices, however, can be programmed for parity. All cascaded devices will be programmed the same since

all control and data pins are common. Program read occurs automatically on the First Load device only to avoid bus contention.

Width Expansion

Both the 7C44X and 7C45X family of Clocked FIFOs can be width expanded for applications requiring data wider than 9 bits.

Width expansion is achieved by wiring the FIFOs in parallel. *Figure 7* shows two FIFOs wired for width expansion. Composite flags should be used to provide proper read and write signaling near the FIFO Empty and Full boundaries. Process variations between FIFOs can result in differences in t_{SKEW1} and t_{SKEW2}. This can cause the update cycles to occur on staggered clock cycles in different FIFOs. Data misalignment can occur at the boundary condition if an operation is performed before all FIFO flag registers are in the same state. Composite flag signaling insures that all FIFOs are in the same state so that an operation at the boundary is performed concurrently by all FIFOs.

The $\overline{PAFE}$ flag from either FIFO may be monitored and will give the correct status, or each FIFO may be programmed differently to give different $\overline{PAFE}$ flags. Parity generation/checking is performed in each device independently according to how they are individually programmed.

Using a Clocked FIFO Like a Standard FIFO

Applications that require high-speed unclocked asynchronous FIFOs memory may use clocked FIFOs. Unclocked asynchronous FIFOs operate at much lower frequencies than clocked FIFOs but feature read and write interfaces driven by single read and write strobes.

Applications can use clocked FIFOs to emulate this operation at high speeds by tying the clock to the appropriate enable line. The enable lines should not be tied straight to ground. Grounding the enable lines directly increases the probability of violating enable set-up times in a noisy environment. Tying the enables to the clocks closes the timing window (when noise can affect the enable pins) and filters out unwanted ground noise. The zero hold-time

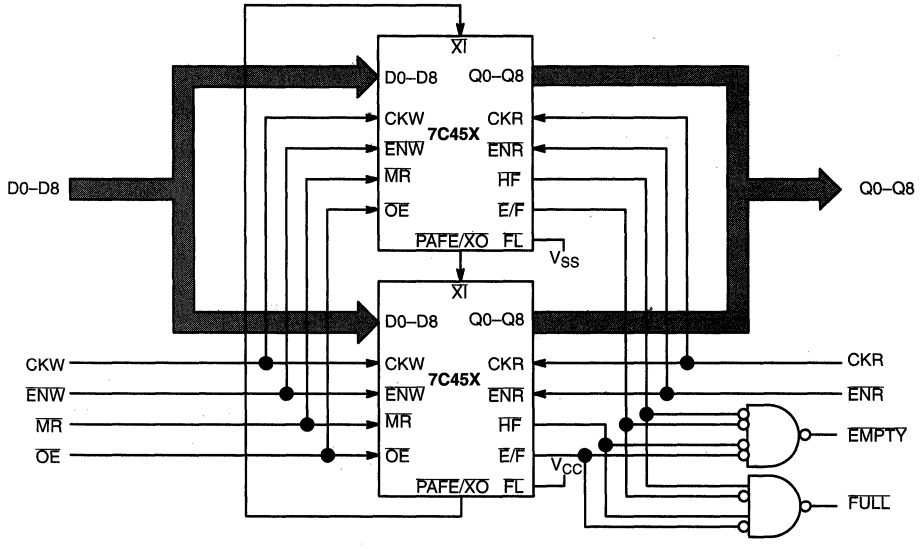


Figure 6. Depth Expansion with CY7C45X

feature of the enable makes this configuration possible. *Figure 8* shows a 7C45X configured as a standard FIFO.

A caveat occurs at the boundary condition flag timing. Absence of a free running clock will prevent the flags from being updated. As a consequence, the internal FIFO control logic will inhibit read or write operations if the respective flag is not updated. To avoid this problem, the FIFO in a boundary state must be strobed in order to force a flag update cycle. Data is not destroyed during the update cycle. The desired operation may proceed once the the flag is updated correctly.

For example, an empty FIFO with its empty flag asserted is written to by strobing the write port. The empty flag, however, is only updated by the rising

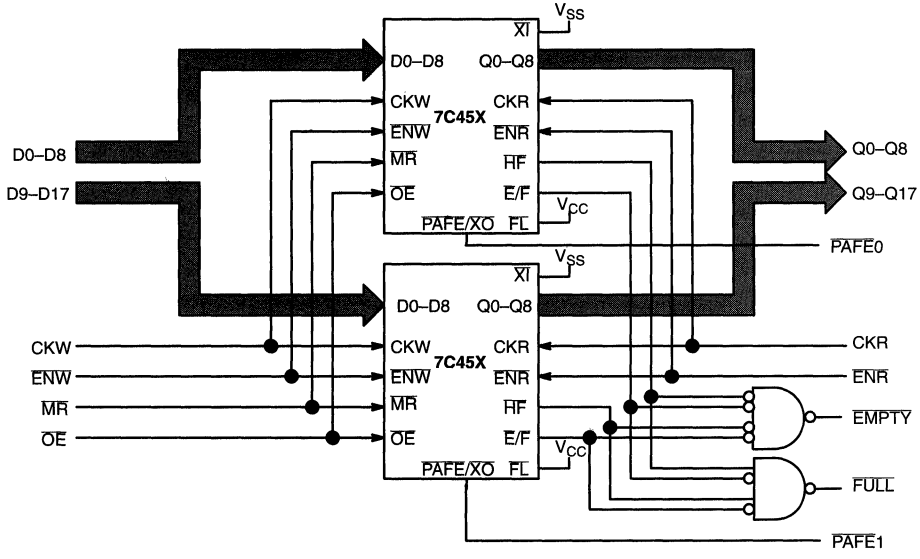


Figure 7. Width Expansion with CY7C45X

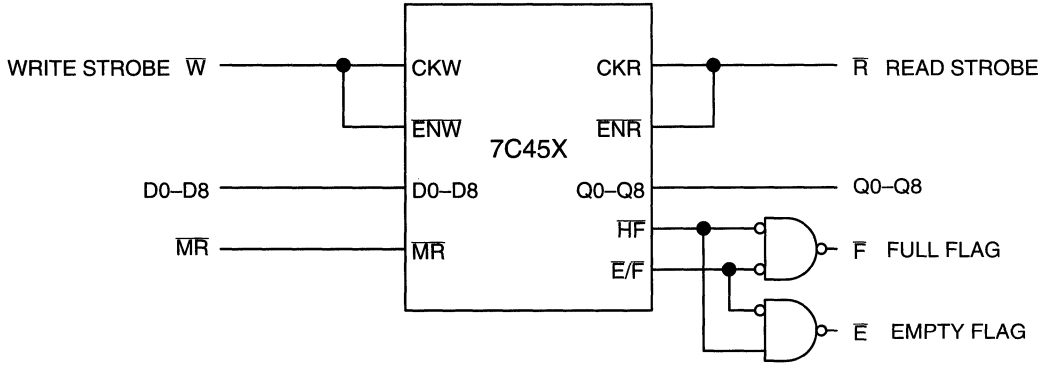


Figure 8. Using a Clocked FIFO Like a Standard FIFO

edge of CKR. Consequently, the read port must be strobed in order to force the flag to be updated. While the empty flag is asserted, the attempted reads are ignored (data remains in the FIFO) and only serve to update the empty flag. Once the empty flag is deasserted, the data can be read from the FIFO in the normal manner.

It also possible to build a controller that forces an update cycle at the FIFO boundary without checking the state of the flags. When a read or a write strobe occurs affecting the state of the memory array, the controller forces an update automatically by toggle the other strobe line.

Conclusion

Cypress 7C44X and 7C45X Clocked FIFOs solve a wide variety of data buffering and storage needs for telecommunications, interprocessor, and data gathering applications. The clocked FIFO architecture offers 70-MHz performance and avoids the timing and noise problems inherent in unclocked asynchronous FIFOs.

FIFO Dipstick Using *Warp2*™ VHDL and the CY7C371

Introduction

Programmable FIFO flags can often simplify the design of a digital system by automatically indicating a status that can prevent overrun or underrun in an elastic FIFO buffer. Although many FIFOs are available with on-chip programmable flag functions, these features are not available on industry-standard asynchronous FIFOs. Of those FIFOs that do have programmable flags, some do not allow the almost-empty and almost-full values to be programmed independently, or in some cases, for these values to be programmed to any specific word boundary. This application note presents a method by which FIFOs of any size may be monitored by an external Programmable Logic Device which will then generate all of the flags necessary for most FIFO applications. The FIFO Dipstick PLD behaves like a measuring device that can observe the level of data within a FIFO.

Application Description

A variable-length up-down counter is implemented with VHDL to measure the exact level of data within a FIFO. The number of bits required for the dipstick counter is dependent on the size of the FIFO and must satisfy the following equation:

2^n = FIFO Depth; Where: n = number of counter bits required

For example, a 2K FIFO would require an 11-bit counter. The n^{th} bit is necessary to prevent the dipstick counter from rolling over to zero when the last byte is written into the FIFO. In other words, the n^{th} bit will only be set when the FIFO is completely full.

Due to the truly asynchronous nature of the read and write ports of a FIFO, a state machine must be implemented to control the operation of the dipstick counter. This state machine must resolve the overlapping and nesting conditions that may occur with the FIFO_READ_L and FIFO_WRITE_L signals to the FIFO. For instance, multiple read pulses may occur within a single write pulse, read and write pulses may occur simultaneously, or read and write pulses may overlap by any amount of time.

The status of the almost-full and almost-empty flags is determined by simply comparing the dipstick counter value to pre-programmed levels and generating the appropriate combinatorial outputs. This method allows for the generation of any flag outputs required for a given application. The almost-full and almost-empty flags are the most typical levels required and are used to determine greater-than-or-equal-to and less-than-or-equal-to specified levels, respectively. Many possibilities exist, however, such as an approx-half-full flag, which could be used to add hysteresis to the half-full value of a FIFO.

Synchronous FIFO Ports

The VHDL/FLASH370™ implementation in this application note is based upon the following assumption. Both the read and write ports of the FIFO are controlled by clocked circuitry and the clocks for each port are synchronous to each other. This assumption allows a single clock to be used for the state machine and the counter. It also provides for the read, write, and reset inputs to be used without any chance of a metastable event occurring. As a result of this synchronous implementation, the almost-flags will change state combinatorially within

three clock cycles after the clock cycle that initiated the read or write. For instance, if a FIFO read is held active for two clock cycles followed by one clock cycle for read-recovery time, the updated almost-empty flag will be available during the read-recovery cycle.

Asynchronous FIFO Ports

The read and write ports of a FIFO may be controlled by clocked circuitry with clocks that are asynchronous to each other. In this case, the state machine and counter should be controlled by the one clock that best suits the application. If it is imperative that the write port of the FIFO receives the almost-full flag immediately, the write port clock should be used. If it is imperative that the read port of the FIFO receives the almost-empty flag immediately, the read port clock should be used. In either case, the read or write input from the opposite port needs to be synchronized to the dipstick's clock before it is used as a state machine input. The CY7C371 is ideally suited for this because of its dedicated inputs, which can be configured as single- or double-registered which will achieve a guaranteed 10-year MTBF. In addition, the port that is asynchronous to the dipstick's clock must also synchronize the almost-flags before use to prevent metastability problems. A negative aspect of using the FIFO dipstick in this application is that additional delays are introduced between a FIFO access and the almost-flags status change. These additional delays may or may not be tolerable, depending on the application.

State Machine Design

The finite state machine observes the FIFO_READ_L and FIFO_WRITE_L inputs in order to control the operation of the dipstick counter (see *Figure 1*). There are eight states required: an idle state, four counter-enabled states, and three counter-disabled states. The counter-enabled states are further categorized into count-up states (write and rd_hold_wr) and count-down states (read and wr_hold_rd). The counter-disabled states (rd_hold, wr_hold, rd_hold_wr_hold) are required for the FIFO_READ_L and FIFO_WRITE_L pulses that are active for greater than one clock cycle.

Within each state, all four permutations of FIFO_READ_L and FIFO_WRITE_L are evaluated to determine the next state. If neither signal is active, the state machine always returns to the idle state. If a single signal goes active or stays active, the state machine will progress to the appropriate state such that the counter-enabled states are active for a single clock cycle only, during each FIFO_READ_L and FIFO_WRITE_L pulse to the FIFO. If both FIFO_READ_L and FIFO_WRITE_L are observed going active on the same clock cycle, the counter-enabled states are avoided completely, allowing the dipstick counter to remain constant. The FIFO_RESET_L signal is not required as an input to the state machine because the dipstick counter will remain cleared if FIFO_RESET_L is active.

Warp2™ VHDL Implementation

The VHDL design used for the FIFO Dipstick is completely behavioral. This high-level design methodology eliminates any need to describe device specific implementations and it also allows for the most readability. The *Warp2* VHDL Compiler will synthesize the design into low-level components necessary for a CY7C371 automatically.

The design entity defines all the inputs and outputs of the design and assigns a type to these signals. The architecture describes the behavior of the circuit. See Appendix A for a listing of the code.

The entity declaration is comprised of a port map, a generic statement, and an attribute statement. The port map defines the FIFO Dipstick inputs, outputs, and the bidirectional counter bits for a variable-length up-down counter. The FIFO_READ_L, FIFO_WRITE_L, and FIFO_RESET_L inputs are written with a _l suffix to indicate that they are active LOW signals, all others are active HIGH. The generic statement is used as a convenient way to define the actual size of the dipstick counter. Simply defining the counter size once in this statement allows the entire design to be modified accordingly, including the number of bidirectional pins defined for the dipstick counter. The attribute statement has been included to define the CY7C371 as the PLD for which

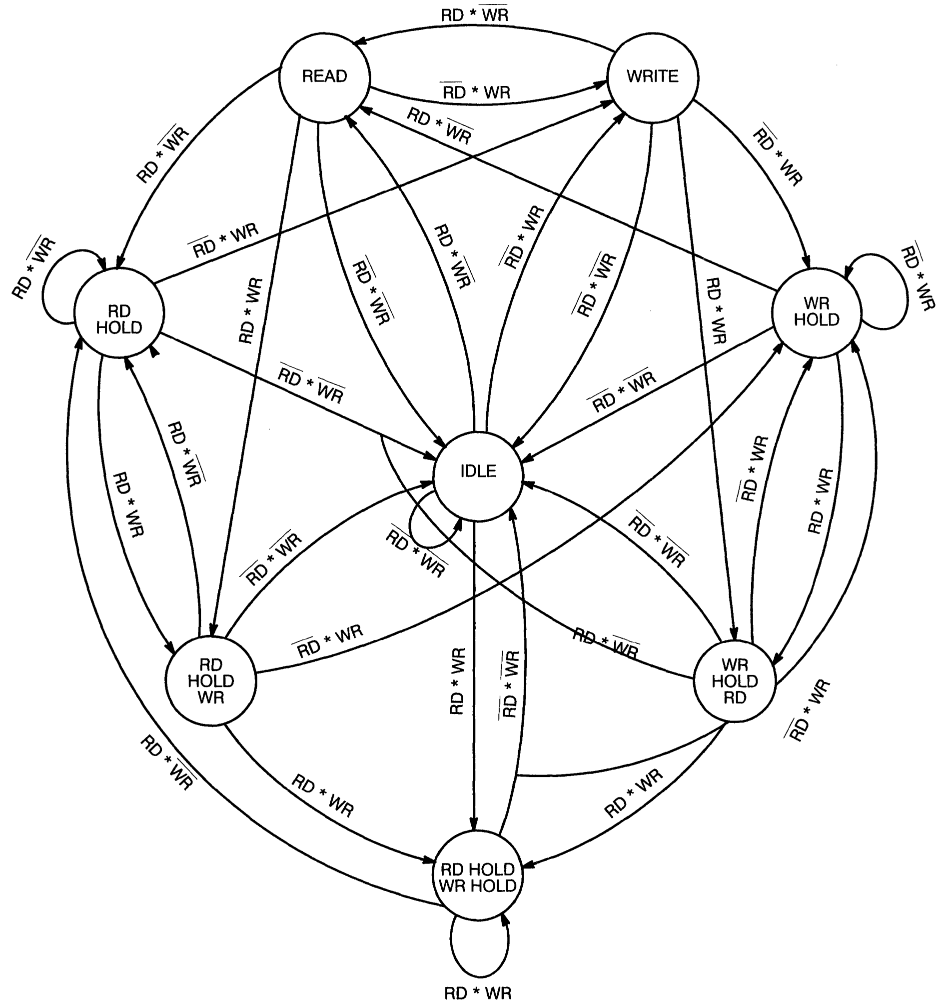


Figure 1. FIFO Dipstick State Machine Bubble Chart

Warp2 will generate a JEDEC file. This statement could be deleted if the "C371" is chosen from the device options menu of *Warp2* instead of the default device.

The architecture body of the design is comprised of three separate processes which will execute in parallel. The process titled *outputs* defines the output flags as a function of the dipstick counter level. Relational operators are used to compare the counter bit_vector to the integer constants defined at the beginning of the architecture. Comparing a bit_vector to an integer is typically not allowed in VHDL, however, *Warp2*'s int_math package provides this capability. Since there is no wait statement included, the afull and aempty signals are combinatorial outputs from this process.

The process titled *counter* controls the operation of the FIFO dipstick counter. If the FIFO_RESET_L input is asserted, then the counter is cleared on the next clock edge. This is accomplished by using a *Warp2* function-call titled i2bv, which converts the integer constant "zero_count" to a bit_vector of the appropriate length. The result is then assigned as the new counter value which is a bit_vector of all zeros. If the FIFO_RESET_L input is not active, the counter operation is then determined by the state of the dipstick state machine and the current counter value. The count-up states will increment the counter unless the counter's MSB is set indicating that the maximum count has been reached. The count-down states will decrement the counter unless it is currently equal to zero. If none of the conditions described above exist, then the counter will maintain its current value as indicated in the else statement. The inc_bv and dec_bv functions, used to increment and decrement the counter bit_vector respectively, are provided by *Warp2* as part of the bv_math package in the work library.

The process titled *state_machine* implements the finite state machine, as represented in the bubble chart of *Figure 1*. This behavioral description makes use of the enumerated-type form. The major advantage of this form is that the state encoding can be easily changed by the user. The current encoding options available are sequential, gray, one-hot, and user-defined and are determined by the attribute state_encoding. The enumerated type is defined at the beginning of the architecture body and is comprised of the eight state names. The case statement within the process defines the state machine transitions based on the current state and the FIFO_READ_L and FIFO_WRITE_L inputs.

Differences From Programmable FIFOs

The following two differences between the FIFO Dipstick design and the use of a FIFO with programmable flags must be understood. First, the latency incurred between a FIFO access and the update of flag status may be prohibitive; refer to the synchronous and asynchronous FIFO ports sections above. Second, the flag outputs of a FIFO will always go inactive based on a FIFO strobe going inactive, whereas the FIFO Dipstick solution will always change flag states based on the strobes going active.

Summary

This application note provides the information required to implement programmable flags for any size FIFO by simply changing the values in the VHDL statements of Appendix A, which are noted as application specific in the source code. For applications that require dynamically alterable flags, a microprocessor port is easily adaptable. The design in Appendix A is also easily adaptable to different FIFO applications, i.e., clocked FIFOs, BiFIFOs, FIFOs with asynchronously clocked ports, etc.

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code

```
USE work.bv_math.all;
USE work.int_math.all;

ENTITY dipstick IS
GENERIC (counter_size: INTEGER := 16);-- APPLICATION SPECIFIC
   PORT (clock, fifo_reset_l, fifo_rd_l, fifo_wr_l: IN BIT;
      afull, aempty: OUT BIT := '0';
      counter: INOUT BIT_VECTOR(counter_size DOWNTO 0));
   ATTRIBUTE part_name OF dipstick: ENTITY IS "C371";
END dipstick;

ARCHITECTURE behavior OF dipstick IS
   CONSTANT afull_value: INTEGER := 32000;-- APPLICATION SPECIFIC
   CONSTANT aempty_value: INTEGER := 07;-- APPLICATION SPECIFIC

   TYPE fifostate IS
      (idle, read, rd_hold, rd_hold_wr, rd_hold_wr_hold,
       write, wr_hold, wr_hold_rd);
   SIGNAL nextstate: fifostate;
   ATTRIBUTE state_encoding OF fifostate: TYPE IS SEQUENTIAL;

BEGIN
   outputs: PROCESS BEGIN
      IF (counter >= afull_value) THEN
         afull <= '1';
         aempty <= '0';
      ELSIF (counter <= aempty_value) THEN
         afull <= '0';
         aempty <= '1';
      ELSE
         afull <= '0';
         aempty <= '0';
      END IF;
   END PROCESS;

   counter: PROCESS
      CONSTANT zero_count: INTEGER := 0;
   BEGIN
   WAIT UNTIL (clock = '1');
      IF (fifo_reset_l = '0') THEN
                counter <= I2BV(zero_count, counter_size);
      ELSIF ((nextstate = write) OR (nextstate = rd_hold_wr)) AND
            (counter(counter_size-1) = '0') THEN
                counter <= inc_bv(counter);
      ELSIF ((nextstate = read) OR (nextstate = wr_hold_rd)) AND
            (counter /= zero_count) THEN
                counter <= dec_bv(counter);
      ELSE      counter <= counter;
      END IF;
   END PROCESS;
```

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code (continued)

```
state_machine: PROCESS
BEGIN
WAIT UNTIL (clock = '1');

CASE nextstate IS
   WHEN idle =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= read;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr_hold;
      END IF;
   WHEN read =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr;
      END IF;
   WHEN rd_hold =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr;
      END IF;
   WHEN rd_hold_wr =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= wr_hold;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr_hold;
      END IF;
   WHEN rd_hold_wr_hold =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
```

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code (continued)

```
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= rd_hold;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= rd_hold_wr_hold;
        END IF;
      WHEN write =>
        IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
           nextstate <= idle;
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= read;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold_rd;
        END IF;
      WHEN wr_hold =>
        IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
           nextstate <= idle;
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= read;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold_rd;
        END IF;
      WHEN wr_hold_rd =>
        IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
           nextstate <= idle;
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= rd_hold;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= rd_hold_wr_hold;
        END IF;
      WHEN OTHERS =>
          nextstate <= idle;
   END CASE;
   END PROCESS;
END behavior;
```

Data Communications – 6

Data Communications Section Contents and Abstracts

This application note covers the design of a dual-speed 100BASE-T4/10BASE-T Network Adapter Card for PCI buses. The CY7C971 100BASE-T4/10BASE-T Transceiver chip is used for the physical layer. The Digital Equipment Corporation 21140 is used as the Media Access Controller (MAC) and PCI interface chip. This application note covers how to interface the CY7C971 to twisted pair RJ–45 connector and how to interface the CY7C971 to the DEC 21140. Printed circuit board layout recommendations are included along with complete schematics and a Bill of Materials.

This application note describes the design of a 100BASE-T4 Ethernet Repeater. This repeater has eight ports, is unmanaged, Class I and is stackable. The physical layer is comprised of eight CY7C971 100BASE-T4 Ethernet Transceivers and the repeater core, which was written in Verilog, is implemented using a CY7C388A 8K FPGA.

This application note describes how to interface the CY7B951 SONET/SDH Serial Transceiver (SST™) with other physical-layer devices. The SST performs clock and data recovery from a SONET/SDH (Synchronous Optical NETwork/Synchronous Digital Hierarchy) 51.84 Mb/s or 155.52 Mb/s interface and can be used in a variety of SONET and ATM applications. The application note begins with a brief introduction to the SST. Next, interface examples will be given that illustrate how to connect the SST to three different ATM controller devices; the first from PMC-Sierra called the PM5345 SUNI, the second, also from PMC-Sierra, called the S/UNI-LITE, and the third from Integrated Telecom Technologies (IgT) called the WAC-013.

This document lists twenty common questions and answers about HOTLink operation and usage. The list of questions was based on customer requests for information on HOTLink. This document is also available in section two of the *HOTLink User's Guide*.

This application note describes how to implement and characterize high-speed serial links made using the CY7B923 and CY7B933 HOTLink parts. Primary topics are an overview of how both HOTLink parts operate internally, how to work with ECL signals, and how to interface to optical fiber and electrical (copper) cables.

Operating high speed parallel buses over significant distances can be problematic due to signal distortion, skew, and crosstalk. These effects can lead to loss of data and failure of the bus. This application note describes how to operate a parallel bus over a serial communication link. Using a high-speed serial link, the distortion, skew, and crosstalk problems are eliminated. In addition, serializing a parallel bus allows for operation of the bus over and extended distance.

This application note considers the interface issues between the Cypress CY7B923/933 (HOTLink) transmitter/receiver and Cypress Clocked FIFOs. This note is divided into two sections: HOTLink Transmitter-Clocked FIFO interfaces, and HOTLink Receiver-Clocked FIFO interfaces. The transmitter interface section provides a simple design example that uses a state machine to control the HOTLink-FIFO interface. A state transition diagram for the controller is provided. Critical path timing analysis is then discussed for this design example. The derived critical path equations and their critical datasheet parameters are provided and explained. A timing diagram is shown to help illustrate these critical timing relationships.

The HOTLink Receiver-FIFO interface section also includes a simple design example. A simple state machine controls this interface. The state machine addresses design issues such as reframing the serial data, BIST (Built-In Self-Test), and programming clocked FIFOs. These issues are discussed in detail. A state transition diagram is included. Critical path timing equations are derived and the advantages of pipelining the interface are discussed. Timing waveforms are shown to help illustrate the critical timing relationships.

This application note considers the interface issues between the Cypress CY7B923/933 (HOTLink) transmitter/receiver and Cypress Clocked FIFOs. The focus of this application note is on applications that use wide data, e.g., 32 bits. This note is divided into two sections: HOTLink Transmitter-Clocked FIFO interfaces, and HOTLink Receiver-Clocked FIFO interfaces. The transmitter interface section provides a simple design example that uses a state machine to control the HOTLink-FIFO interface. The data word size is chosen to be 32 bits. A simple 4:1 mux is used to funnel the data out of the FIFOs and into the HOTLink Transmitter. The state machine controls the sequencing of the data through the muxes. A state transition diagram for the controller is provided. Critical path timing analysis is then discussed for this design example. The derived critical path equations and their critical datasheet parameters are provided and explained. A timing diagram is shown to help illustrate these critical timing relationships.

This document lists twelve common questions and answers about usage and modifications to the CY9266 HOTLink evaluation cards. The list of questions was based on customer requests for information on the CY9266 HOTLink evaluation cards. This document is also available in section thirteen of the *HOTLink User's Guide*.

This document describes the construction, interfaces, and operation of the CY9266–F (optical), CY9266–T (shielded twisted-pair/twinax), and CY9266–C (coaxial cable) HOTLink Evaluation Boards. These boards implement a bidirectional parallel-to-serial and serial-to-parallel communications link, capable of operation at serial rates of 160 to 330 Mbits/second (16–33 Mbytes/second). Complete schematics, parts lists, and artwork are included.

100BASE-T4 / 10BASE-T Ethernet PCI Network Adapter

Background

This application note describes the design of a dual speed 100BASE-T4/10BASE-T Ethernet Network Adapter card for PCI systems using the Cypress CY7C971 PHY and the Digital Semiconductor 21140 MAC (Media Access Controller). The adapter card has the following features:

- Dual Speed 100BASE-T4/10BASE-T
- Full Duplex 10BASE-T
- IEEE Compliant Auto-Negotiation
- High Performance PCI Interface

The network adapter card's function is to interface the host computer to the network cabling. The adapter card plugs into the host computer's PCI bus. The twisted-pair network cable plugs into the end of the network adapter card via an 8-pin modular RJ−45 jack. *Figure 1* illustrates a PCI Network Adapter with a host motherboard.

The network interface card contains all of the circuitry for the Ethernet physical layer, MAC layer, and PCI interface. The Cypress CY7C971 contains all of the physical layer circuitry for 100BASE-T4, 10BASE-T, and Auto-Negotiation. The DEC 21140 contains all of the logic for Ethernet MAC and the PCI bus interface. The CY7C971 and the DEC

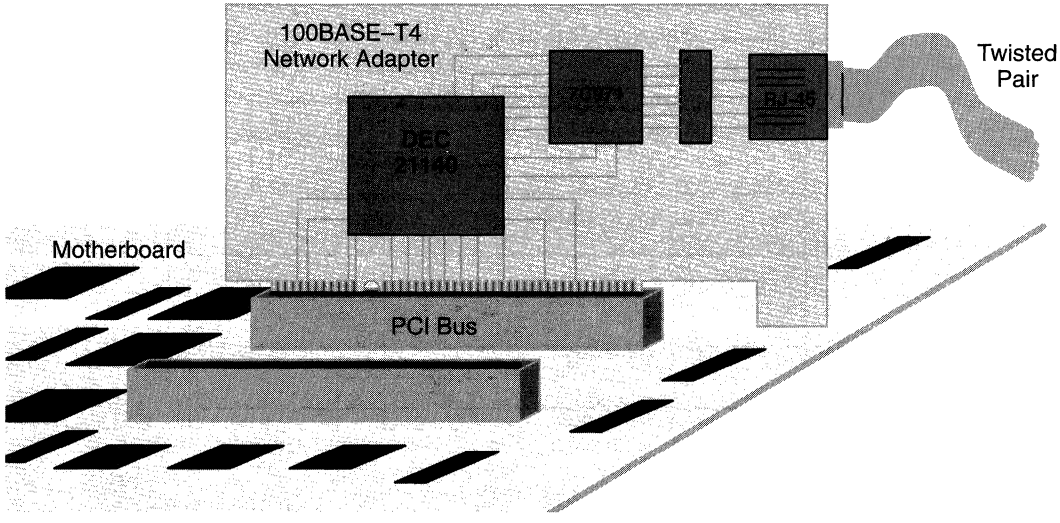


Figure 1. PCI Network Adapter Card

21140 interface to each other through the Media Independent Interface (MII). The MII is an IEEE standard interface between the Ethernet physical layer and the MAC layer.

CY7C971

Media Dependent Interface (MDI)

The CY7C971 provides a simple interface to the 8-pin modular RJ−45 jack. No expensive external filters or components are necessary because all transmit filtering and equalization are performed on-chip. All CY7C971 media interface pins are dual speed, allowing shared magnetics to be used. A quad 1:2 transformer for electrical isolation and termination resistors to match the cable impedance are all that is required.

The output buffer design uses a feedback voltage driver that minimizes power consumption and controls the common mode output voltage. The transformer provides sufficient common-mode rejection over the frequencies of interest so that an external common mode choke is not needed. *Figure 2* shows a schematic of the media interface with the CY7C971.

The characteristic impedance of the twisted pair medium is a nominal 100Ω The 1:2 transformer reduces (by the square of the turns ratio) medium load impedance to 25Ω on the primary (971) side. The termination resistors and the output buffer impedance together form a matching 25Ω load. The matching load insures that maximum signal is transferred to the medium and minimizes reflections due to impedance mismatch.

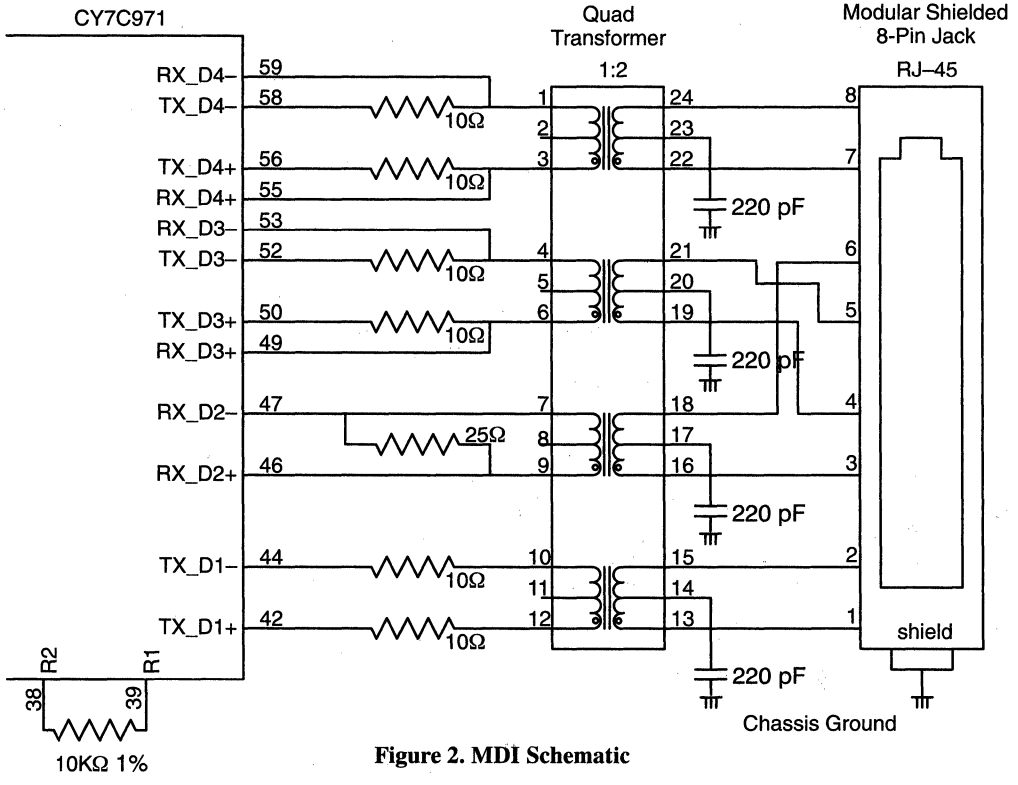


Figure 2. MDI Schematic

The center taps on the media side of the transformer are connected to the chassis ground through 220-pF (minimum) high-voltage (2 KV) capacitors. These capacitors help absorb common-mode noise that is picked up or generated on the twisted-pair medium. The capacitors must be capable of withstanding the isolation requirements specified in the 100BASE-T4 standard. High-voltage ceramic disc capacitors are economical and work well in this application.

The high precision currents needed for the transmit DAC and equalizer are derived from the external 10KΩ 1% resistor on pins R1 and R2. An internally generated band-gap voltage reference is used by the CY7C971 for all internal reference voltages.

Media Independent Interface (MII)

The Media Independent Interface (MII) is the IEEE Ethernet standard interface for communication between the MAC and PHY devices. The MII supports both 100 Mb/s and 10 Mb/s data transfer modes. In 100 Mb/s mode, the MII transfers nibble wide data groups at 25 MHz transfer rate yielding 100 Mb/s throughput. In 10 Mb/s mode, the transfer rate is reduced to 2.5 MHz for a 10 M/s throughput. During all transfers, the receive and transmit reference clock are continuously sourced from the CY7C971 PHY to the 21140 MAC. *Figure 3* shows the MII connections between the CY7C971 and the DEC 21140.

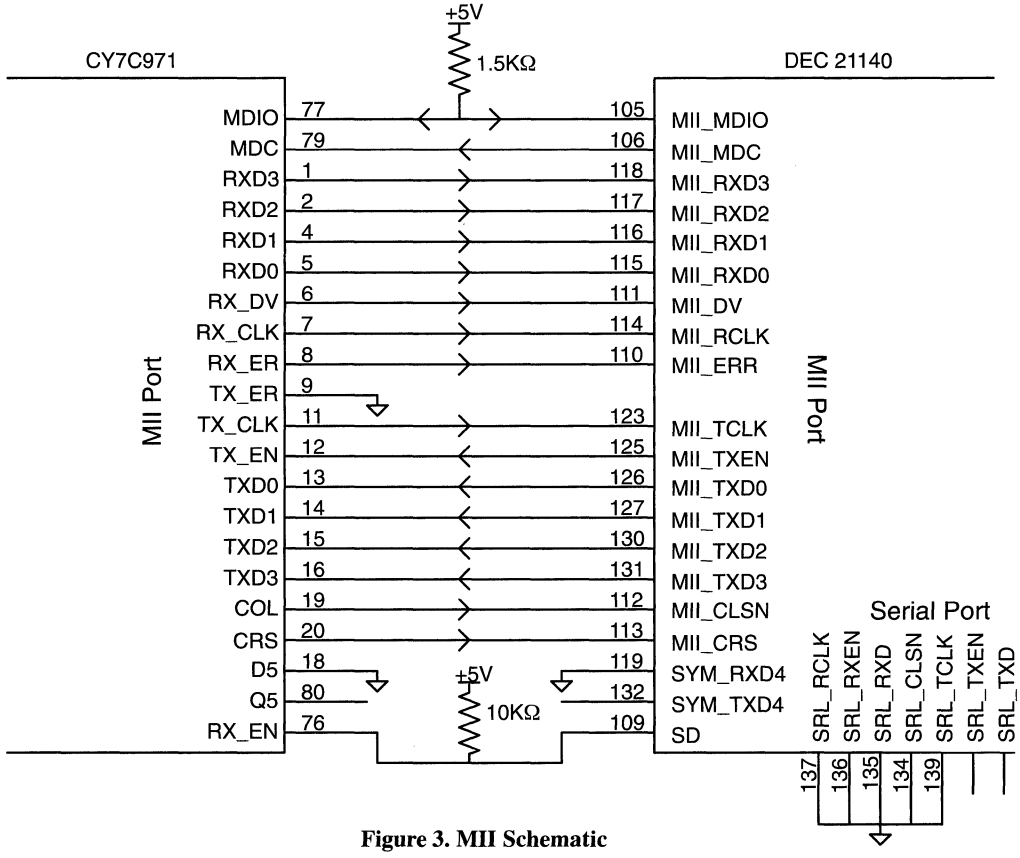


Figure 3. MII Schematic

All data transfers between the CY7C971 and the DEC 21140 are over the MII interface. The DEC 21140 has an additional 7-wire serial interface for an external 10 Mb/s transceiver. This port is not used in conjunction with the CY7C971 and these port pins are tied inactive as shown in the schematic (Appendix A).

The CY7C971 has a buffer enable input signal, RX_EN, that is not part of the MII standard. This pin is used to place the MII output buffers in high impedance. In this application, RX_EN should be tied HIGH to permanently enable the MII output buffers. The Q5 and D5 pins on the CY7C971 are not used in MII mode. D5 can be tied either HIGH or LOW. Since the DEC 21140 does not support explicit transmit error generation over the MII interface, the 971 TX_ER pin should be tied LOW to prevent inadvertent transmit error generation.

The MDC and MDIO pins form a simple two-wire serial management interface between the 7C971 and 21140. MDC is a clock signal sourced from the 21140. The MDIO line is a bidirectional data line used to transfer management data frames. The MDIO signal requires a 1.5 Kohm pull-up resistor to VCC. This interface is used to transfer standard management frames that control and monitor the behavior of the CY7C971. Management frames contain a PHY address, register number, op code, and a 16-bit data field.

Clock Pins

The CY7C971 generates all internal and external clock signals from its on-board oscillator circuit. The oscillator circuit requires an external 25 MHz parallel resonant crystal connected between the CLKO and CLKI pins. The external load capacitors (C_{load}) should be chosen so that the total load capacitance matches the parallel resonant capacitance of the crystal. The load capacitors form a series capacitance network. The required load capacitance is derived from the following equation:

$$C_{xtal} = (C_{pin} + C_{load} + C_{trace}) / 2$$

$$C_{load} = 2 \bullet C_{xtal} - C_{pin} - C_{trace}$$

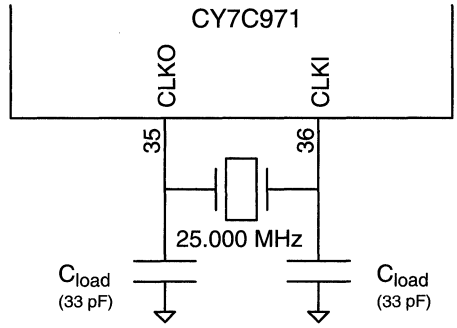


Figure 4. Clock Pins

The package pins contribute approximately 1.5 pF to the parallel load capacitance. Board trace and pads contribute between 1–2 pF of parasitic capacitance depending on trace length, width and dielectric thickness. According to this formula, an 18-pF parallel resonant crystal would require 33-pF load capacitors.

The crystal should have frequency stability of 100 ppm or less in order to comply with the Ethernet standards *Figure 4* shows the CY7C971 clock pin connections. The load capacitors are connected between the Clock pins and ground.

LED Pins

The CY7C971 can drive LEDs directly. The LED pins use an open drain output buffer that can sink up to 12 mA. The buffers have a weak internal pull-up resistor. *Figure 5* shows how the LED pins connect to the LEDs.

The $\overline{LTX}$ and $\overline{LRX}$ pins indicate when the CY7C971 is actively transmitting or receiving Ethernet frames. $\overline{LTX}$ indicates that the transmitter is active, and $\overline{LRX}$ indicates that the receiver is active. These signals are time stretched to at least 25 ms so that light pulses emitted from the LED can be detected by the human eye. These pins may be tied together in a wire-or fashion to form a generic activity indicator.

The $\overline{LINKT4}$, $\overline{LINKT}$, and $\overline{LINKFD}$ pins indicate when the CY7C971 is in the link pass state for

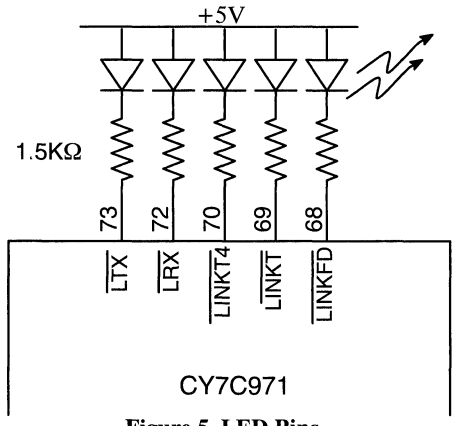


Figure 5. LED Pins

100BASE-T4, 10BASE-T, or 10BASE-T Full Duplex. The operating mode is determined either through the Auto-Negotiation process or by manual configuration with the control register (see section on MDC/MDIO Management Interface). The CY7C971 will enter a link pass state when an operating mode has been selected (either through Auto-Negotiation or manually) and properly formed technology dependent link integrity pulses are received from the medium. If only a single link indication is needed, the link indicator pins may be tied together in a wire-or fashion to form a generic link pass signal. These signals may also be individually connected to the 21140's General Purpose pins in order to quickly inform the MAC of any changes in the link status.

Configuration Pins

The configuration pins are wired for the adapter card application as shown in *Figure 6*. The ENT4, ENT, ENFD, AUTONEG are wired HIGH to enable all of the 7C971 operating modes. At power-up or during a hard reset, the logic values on these pins are loaded into their corresponding ability bits in the MII Status Register. The ability bits in the Status Register dictate whether an operating mode can be become active. After the power-up or reset cycle completes, the Auto-Negotiation process will advertise all operating modes that the Status Register reports as enabled. Management can alter the advertised abilities by changing the code word in the Auto-Negotiation Advertisement Register (Reg. 4).

The ISODEF (Isolate Default) pin is tied LOW in order to force the CY7C971 to power up with the MII ready for normal operation (not isolated). The Isolate Bit (0.10) will indicate normal operation as the default setting. The address pins (A0–A4) are wired for PHY address 01H. Address 00H is reserved for external transceivers and should not be used. The CY7C971 will respond to PHY management frames that use the assigned address. The values on the ISODEF and A0–A4 pins are latched into the 7C971 during a hard reset or power-on reset.

The MODE pin is tied HIGH to force the 7C971 into MII mode. MII mode enables the MII, PCS (Physical Coding Sublayer), and PLS (Physical Layer Signaling) logic. The PCS performs the 8B6T encoding/decoding and serial/parallel conversion for 100BASE-T4. The PLS performs Manchester encoding/decoding and serial/parallel conversion for 10BASE-T. When the MODE pin is LOW (PMA Mode), the MII, PCS, and PLS are disabled and the 100BASE-T4 PMA (Physical Medium Attachment) interface is exposed on the MII I/O pins. PMA Mode is used only in repeater applications.

The Test pin is tied LOW to permanently disable the CY7C971 test mode. Test mode is used for factory ATE testing only.

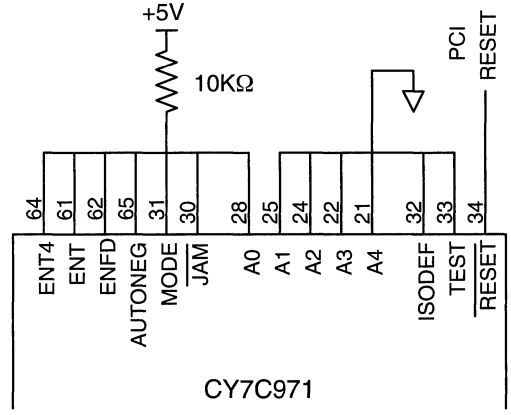


Figure 6. Configuration Pins

The $\overline{\text{RESET}}$ pin should be connected to the PCI reset pin on the card edge. Power-on reset is taken care of by an internally generated reset signal. During a hard or power-on reset, the values on the ENT4, ENT, ENFD, AUTONEG, ISODEF, and A0–A4 are loaded into the CY7C971 and all of the logic and analog circuits are forced to their default states. During a soft reset all of the logic and analog circuits are reset but the values on the configuration pins are ignored. The software drivers can issue a soft reset by setting the Reset Bit (0.15) in the Control Register. This bit is self clearing.

Layout Considerations

The adapter card design is simple enough to fit on a standard PCI short card (3.5" x 5") or smaller PCB. A 4 layer PCB construction with dedicated power and ground planes is recommended. The DEC 21140 requires a 3.3V power supply. The CY7C971 requires a 5V supply. Separate 5V and 3.3V power planes can be partitioned on a single power layer. *Figure 7* shows an example of partitioned power planes with component placement. The ground plane runs under both the 5V and 3.3V planes. There is a cutout in both the power and ground planes under the RJ–45 and transformer.

The media interface components can be neatly placed behind the RJ–45 connector. *Figure 8* illustrates the physical layout of the media interface with a 4-layer board. 0.027 µF decoupling capacitors are used on each of the CY7C971 power pins. These 0805 SMT capacitors are placed in a row as close to the pins as possible. The termination resistors fit neatly in a row behind the decoupling capacitors. Tantalum 10 µF capacitors are placed on opposite corners of the CY7C971. The CY7C971 media interface and power pins were placed in such a way to minimize the use of vias and simplify board layout.

Software Considerations

Software drivers are responsible for configuring registers within the DEC 21140 for proper operation with the CY7C971. The software drivers are also responsible for transferring Ethernet packets between the host computer's local memory and the

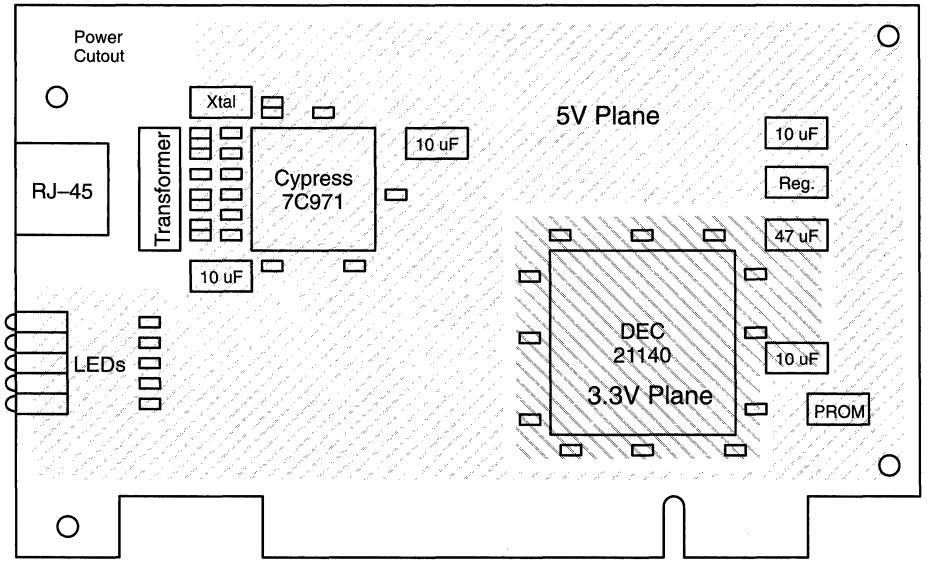


Figure 7. Power Plane and Component Placement

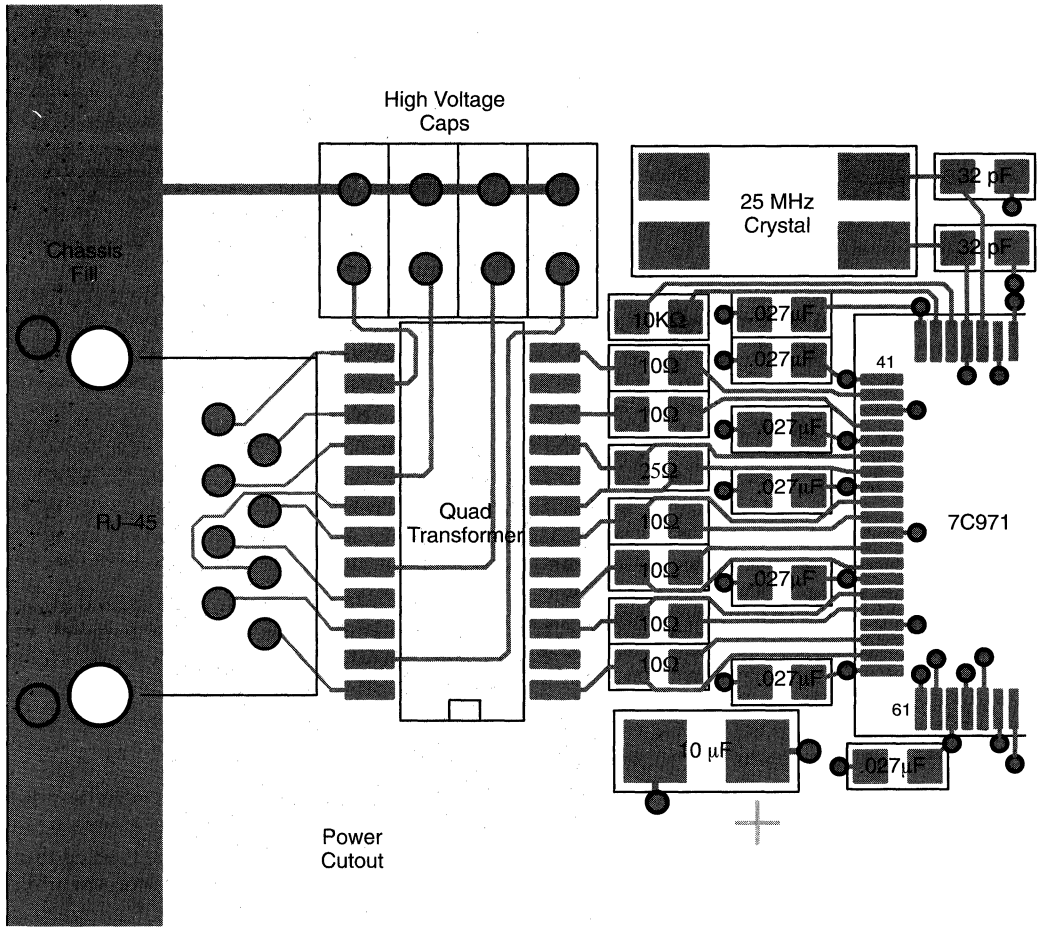


Figure 8. Media Interface Layout

21140's data buffers, and for managing the 21140 and CY7C971 resources during normal operation.

The CY7C971 contains an on-chip management facility that is accessed through its serial management port on the MII. The management facility consists of registers that report and control basic activities of the PHY such as Auto-Negotiation and link status.

The CY7C971 management facility acts as a slave device to management accesses from the MAC. Management data is transferred between CY7C971 and the DEC 21140 MAC with the MDC and MDIO pins on the MII. This connection is shown in *Figure 3*.

The DEC MAC emulates the management agent with its software drivers. During power-up, reset, or a down link, the drivers should poll the management registers to determine the result of Auto-Negotiation and the state of the link. While the link is up, the drivers should poll the CY7C971 Status Register on a timely basis to make sure the link is active. The CY7C971 was designed so that standard MII compliant software drivers can support the management facility.

DEC Register Set-Up

The 21140 Command and Status Registers (CSR) must be configured so that the 21140 communicates with the CY7C971 through the MII port. Register CSR6 in the 21140 controls the MAC-PHY interface configuration. The 21140 parallel MII port is enabled with the Port Select bit in CSR6 (CSR6, bit 18). When set, the MII port is enabled and the serial 10-Mb/s port is disabled.

The PCS Function and Scrambler Mode inside the 21140 must be disabled for proper operation with MII based transceivers such as the CY7C971. PCS and scrambler modes are used with 100BASE-X physical layer devices only. The PCS Function is disabled by clearing the PCS bit in CSR6 (CSR6, bit 23). The scrambler is disabled by clearing SCR bit in CSR6 (CRS6, bit 24).

The Transmit Threshold Mode (TTM) must be adjusted according to the operating speed of the link. This bit determines the number of bytes in a frame that must be stored in the transmit FIFO before the transmission process is initiated. In 10-Mb/s mode, the TTM bit (CSR6, bit 22) should be set. In 100-Mb/s mode, the TTM bit should be cleared. The link operating speed can be determined by polling the CY7C971 management Auto-Negotiation and Control registers or by monitoring the LED Link pins through the General Purpose Register.

MDC/MDIO Management Interface

The CY7C971 contains all of the standard and extended registers defined in the MII standard (Registers 0–7). There is also an additional CY7C971 specific register (Reg.16).The MAC can perform write and read operations to the CY7C971 management registers by transferring management frames over the MDIO serial interface. The MDC signal serves as the management data clock and is sourced from the MAC. The MDIO signal is bidirectional. The frame structure is shown in *Figure 9*.

The management frame is comprised of several fields. The start sequence 01 is used to identify the start of a frame. The op-code field determines whether a read, write, or no-op will be performed. The address field determines the target PHY. The CY7C971 will only respond to management frames whose address matches the address assigned to the CY7C971 by the address pins A0–4. In this application, the CY7C971 address has been permanently wired to 01H. All management accesses to the CY7C971 should use this address.

The register field determines the target register for the operation. The turn around field provides time to switch the direction of the bus during a read operation. The next 16 bits are the data field. During a read operation, the PHY will drive the MDIO line with the target register contents. During a write operation, 16 bits are transferred to the PHY from the MAC and written in the target register.

The CY7C971 can accept management frames that are not preceded by a 32-bit preamble. A sequence of 32 ones will force a reset on the CY7C971 management facility. It is recommended that the MAC issue this 32-bit sequence after power-up and periodically during normal operation.

The CY7C971 supports the standard and expanded MII register set. The Expanded Register set includes the OUI (Organizationally Unique Identifier) and Auto-Negotiation registers (registers 2–7). *Figure 10* shows the CY7C971 register map.

Control Register (Reg. 0)

The Control Register is used to manually set the operating modes and enable/disable certain features. Auto-Negotiation can be enabled/disabled through this register with bit 0.12. When Auto-Negotiation is enabled, the speed of the link is determined automatically, and the speed selection bit (0.13) has no effect. When Auto-Negotiation is disabled, the speed selection bit determines the speed of the link.

The loop back bit (0.14) is used to internally loop the transmit signal path to the receive signal path. Placing the CY7C971 in loopback mode will cause the

	Start	Op Code	PHY Address	Register Number	Turn Around	Data
Read	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD
Write	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD

Figure 9. Management Frame Structure

link to be broken and the transmit drivers will be forced to idle. The power-down bit (0.11) places the CY7C971 in low power stand-by mode. All of the analog circuits are placed in low power mode and the clock is stopped to all of the CMOS digital logic. Only the MDC/MDIO port is active. When power-down mode is exited, the CY7C971 will reset all of the registers to their default values. Any register setting other than the default value must be restored by the driver.

Status Register (Reg. 1)

The Status Register is a read-only register that reports the capabilities and status of the CY7C971. The status of the Auto-Negotiation process can be monitored through bit 1.5. This bit reports when Auto-Negotiation has completed. The Remote Fault bit (1.4) will indicate if Auto-Negotiation has detected a remote fault at the other end of the link. The Link Status bit indicates whenever any technology (i.e., the 10BASE-T or the 100BASE-T4 circuits of the CY7C971) has entered the Link Pass State. This means that the link is available for data transmission and reception.

OUI Registers (Reg. 2–3)

Registers 2 and 3 contain the Cypress Semiconductor Organizationally Unique Identifier and the CY7C971 part and revision number. The OUI is a 24-bit sequence that is uniquely assigned to organizations for identification purposes by the IEEE.

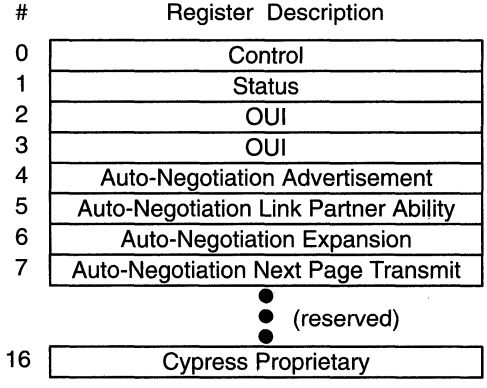


Figure 10. Register Map

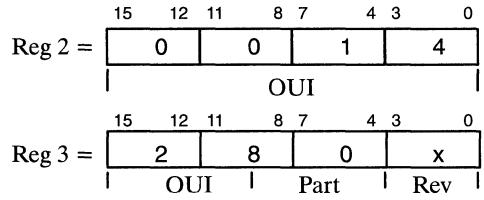


Figure 11. OUI Registers

The Cypress OUI is 00A050h. According to the Ethernet MII standard, twenty-two bits of the OUI are split between Registers 2 and 3. Register 2 contains 16 bits of the OUI and register 3 contains the other 6. Register 3 also contains 6 bits for the CY7C971 part number and 4 bits for the revision number. The register mapping and contents are shown in *Figure 11*.

Auto-Negotiation Registers (Reg. 4–7)

Registers 4 through 7 manage the Auto-Negotiation process. These registers only have meaning when Auto-Negotiation is enabled. Management intervention is not required during the normal Auto-Negotiation process. Management should only intervene with the Auto-Negotiation process in order to influence the outcome.

The Auto-Negotiation Advertisement Register (Reg. 4) holds the 16-bit code word that the CY7C971 advertises over the medium. This code word encodes the capabilities of the CY7C971, the LAN technology (CSMA/CD Ethernet), and fault indications. During power-up or reset, this register will set to the default conditions of the CY7C971 that are dictated by the enable pins. This causes Auto-Negotiation to only advertise the capabilities that are enabled. These enabled capabilities are reflected in the Status register. Management may intervene in the Auto-Negotiation process by writing to this register. Only the operating modes that are enabled in the Status Register will be advertised. Any attempt to advertise a disabled mode (disabled when ENx pin is LOW) by writing to the Advertisement Register will be ignored. Management should restart the Auto-Negotiation process by setting bit 0.9 (Restart Auto-Negotiation Bit) if the contents of the Advertisement Register are changed. *Figure 12* shows a block diagram of how the enable pins affect

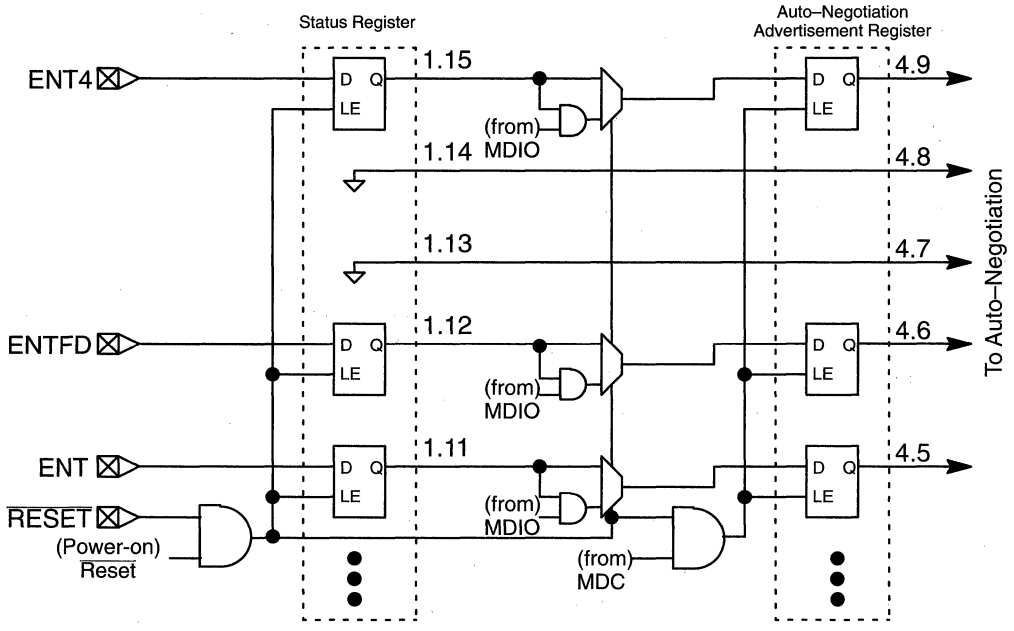


Figure 12. Register Block Diagram

Auto-Negotiation Advertisement and Status Registers.

The Auto-Negotiation Link Partner Ability Register (Reg. 5) contains the code word that has been consistently received from the PHY at other end of the medium. This register is valid when the Page Received bit (6.1) is set in Register 6. Auto-Negotiation uses the received code word to decide the operating mode of the link. The choice is based on the priority resolution table in the Auto-Negotiation standard. 100BASE-T4 has the highest priority. If Auto-Negotiation completes through parallel detection, the contents of this register are invalid. (Parallel Detection part of the Auto-Negotiation process. Its function is to detect the presence of Ethernet transceivers that do not support Auto-Negotiation.)

The Auto-Negotiation Expansion Register (Reg. 6) is a Read-Only register that reports the status of the Auto-Negotiation process. This register should be monitored during the Auto-Negotiation process in order to make sure that code words are being received and that there is not a Parallel Detection fault.

Register 7 is used to hold the Next Page code word that is to be transmitted during next page exchanges. Next Pages are code words that can be sent in addition to the base code word in the advertisement register. The Next Page facility is intended to be used as a simple scheme for passing messages between the PHYs on the medium before the link becomes active. The messages may contain information such as the presence of a fault, for example. The Next Page Transmit Register defaults to 2001H (Null Message) after power-up or a reset.

Cypress Proprietary Register (Reg. 16)

The Cypress Proprietary Register (Reg. 16) contains specific information about the CY7C971. Bit 15 indicates the polarity of the RX_D2± signal pair. When clear, this bit indicates that the polarity of RX_D2± is correct or undetermined. When set, this bit indicates that inverted polarity on RX_D2± was detected and has been corrected. Inverted po-

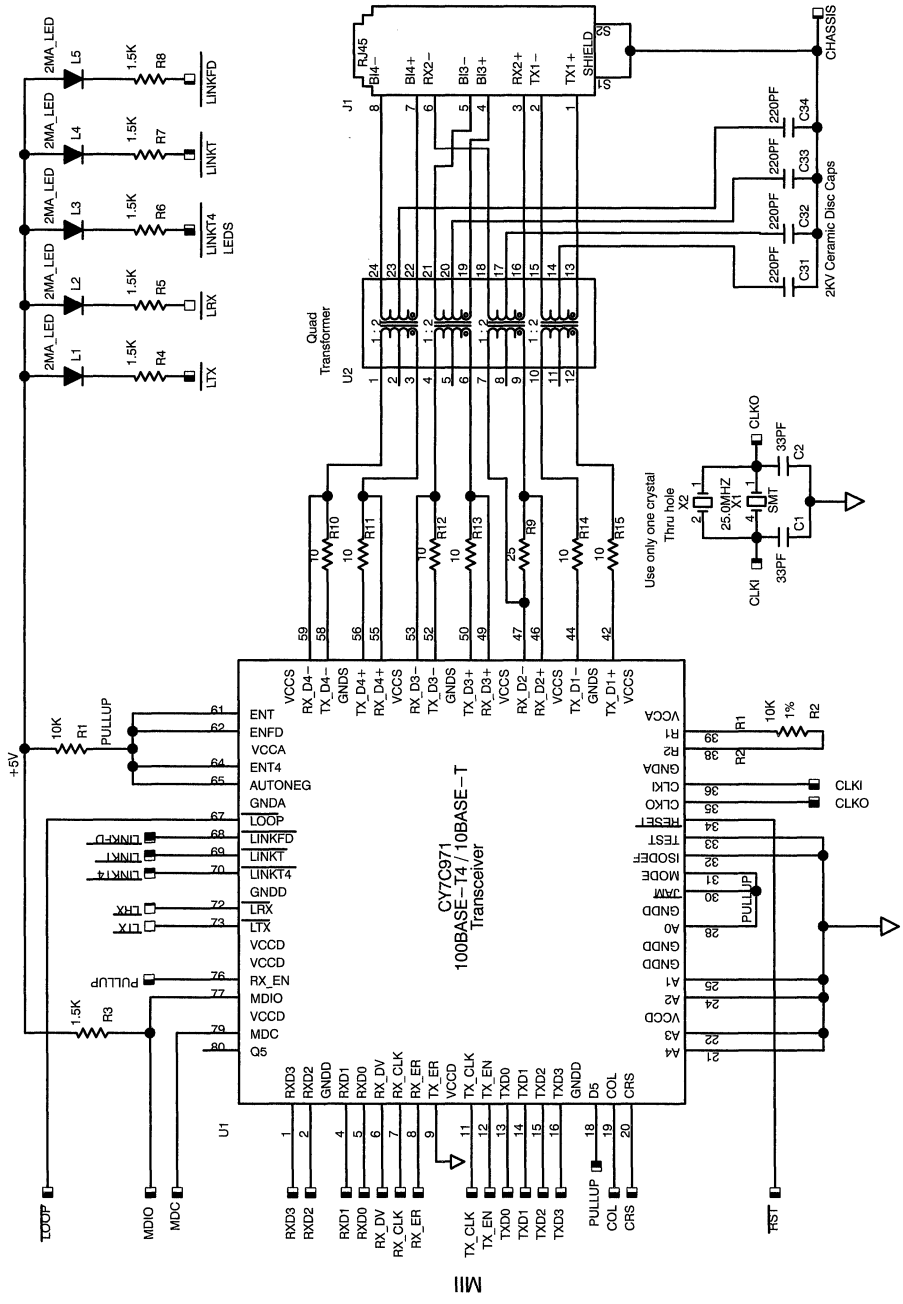
larity is most likely caused by inadvertently reversing the signal wires at the medium connector.

Conclusion

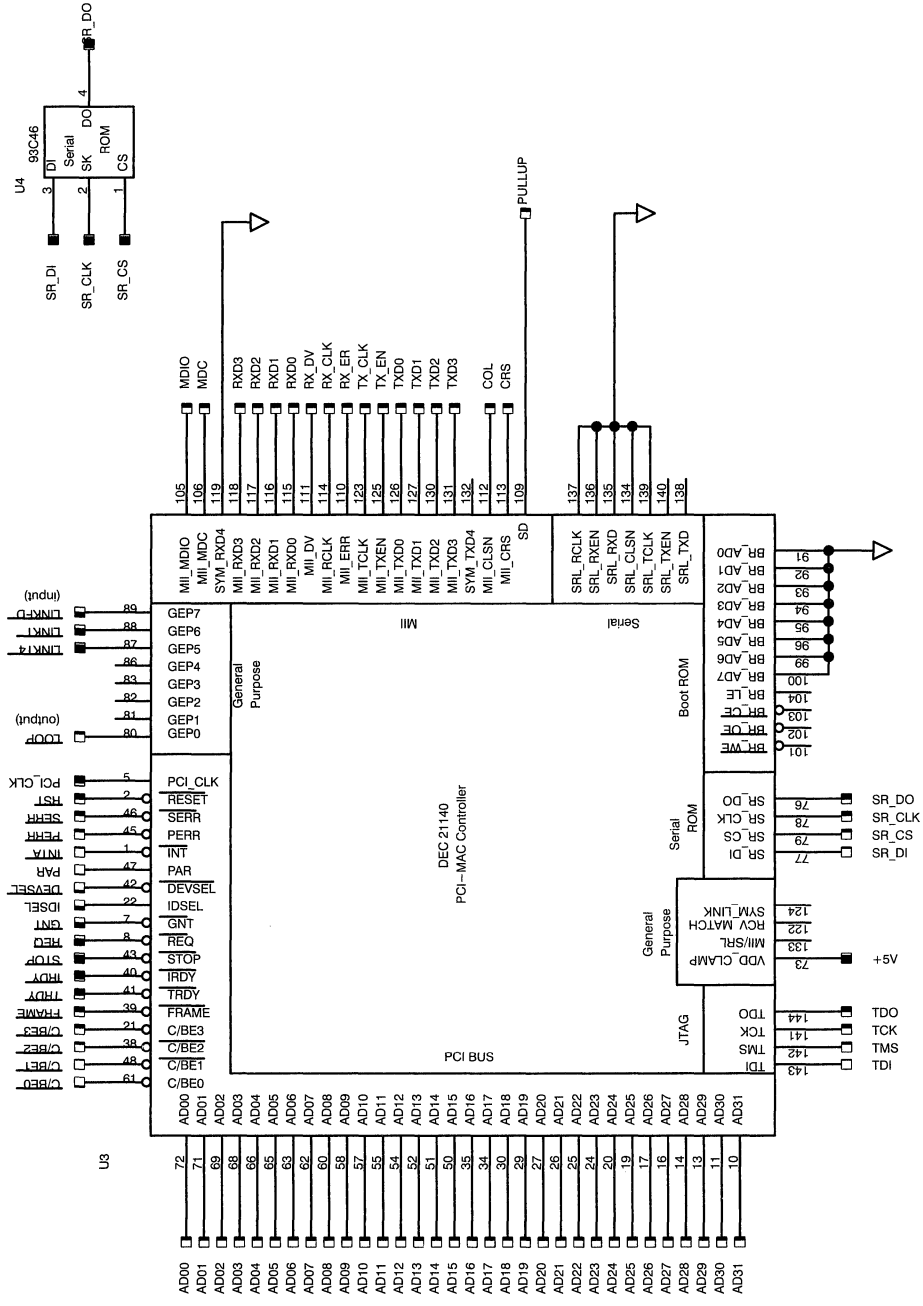
This application note covers the major issues for a dual speed Ethernet/PCI Bus adapter card design using the CY7C971 100BASE-T4/10BASE Transceiver and DEC21140 MAC. The high degree of integration in the CY7C971 keeps the number of external components to a minimum helping to reduce system cost and design effort.

The complete adapter card schematics and a bill of materials are included at the end of this application note (Appendix A and Appendix B, respectively). More information on the CY7C971 can be found in the data sheet. For more information on 100BASE-T4, MII and Auto-Negotiation standards, consult the IEEE 802.3u document: "MAC Parameters, Physical Layer, Medium Attachment Units and Repeater for 100Mb/s Operation."

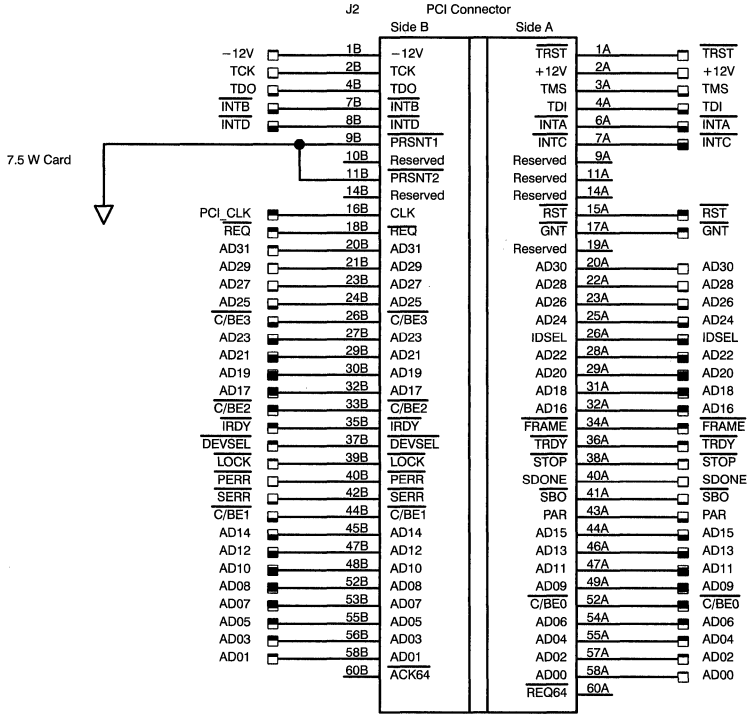
Appendix A. Schematics (Sheet 1 of 4)



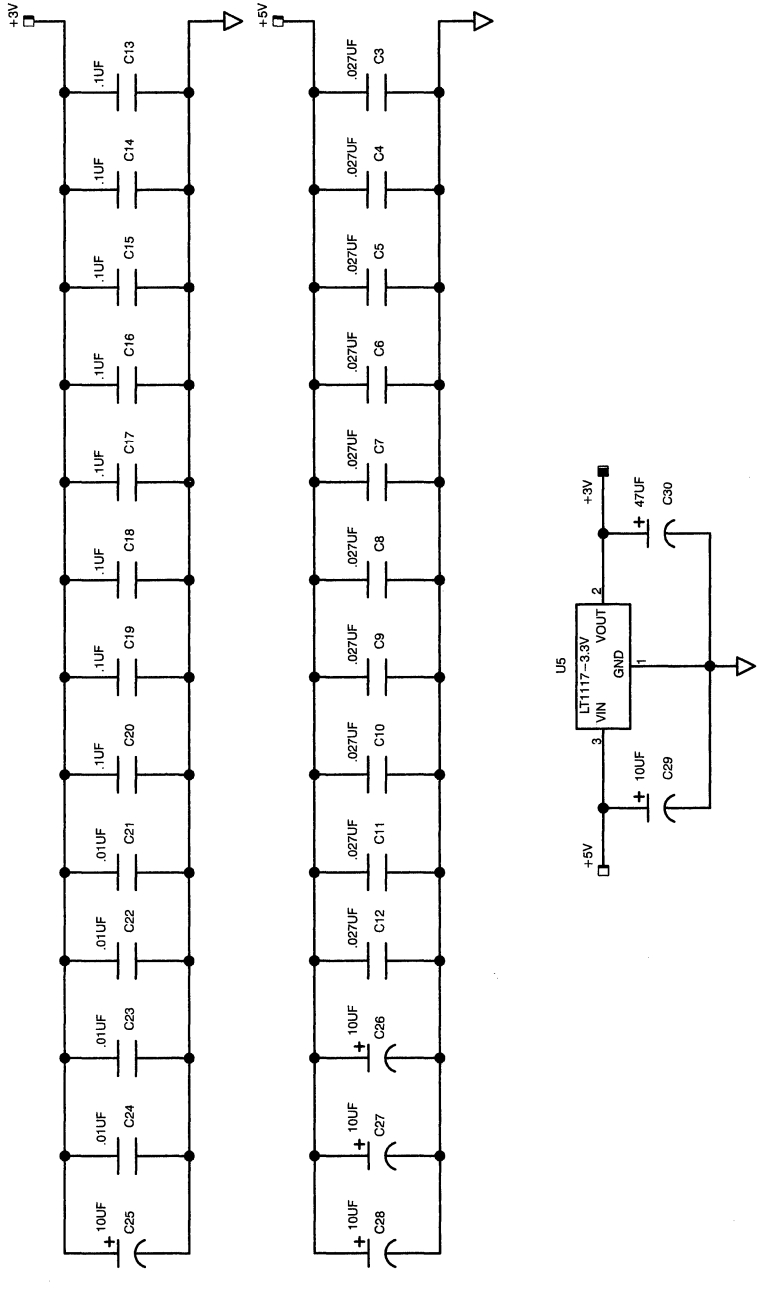
Appendix A. Schematics (Sheet 2 of 4)



Appendix A. Schematics (Sheet 3 of 4)



Appendix A. Schematics (Sheet 4 of 4)



Appendix B. Parts List

Description, Vendor, Part Number	Qty	Reference Designator
10 µF/16V Tantalum Capacitor (EIA Size C) Sprague Elec. 293D106X9016C2	6	C25, C26, C27, C28, C29, C31
47 µF/16V Tantalum Capacitor (EIA Size D) Sprague Elec. 293D476X9016D2	1	C30
.1 µF/50V Ceramic Capacitor (Size 1206) Panasonic ECU-V1H104KBW	8	C13, C14, C15, C16, C17, C18, C19, C20
.01 µF/50V Ceramic Capacitor (Size 1206) Panasonic ECU-V1H103KBM	4	C21, C22, C23, C24
.027 µF/50V Ceramic Capacitor (Size 0805) Panasonic ECU-V1H273KBX	10	C3, C4, C5, C6, C7, C8, C9, C10, C11, C12
33 pF/50V Ceramic Capacitor (Size 0805) Panasonic ECU-V1H330JCG	2	C1, C2
220 pF/2KV Ceramic Disk Capacitor Murata/Erie DE0405B2212KV	4	C31, C32, C33, C34
10.0K ohm 5% 1/8W Resistor (Size 0805) Panasonic ERJ-6GEYJ10.0K	1	R1
10.0K ohm 1% 1/10W Resistor (Size 0805) Panasonic ERJ-6ENF10.0K	1	R2
10.0 ohm 1% 1/10W Resistor (Size 0805) Panasonic ERJ-6ENF10.0	6	R10, R11, R12, R13, R14, R15
24.9 ohm 1% 1/10W Resistor (Size 0805) Panasonic ERJ-6ENF24.9	1	R9
1.50K ohm 5% 1/10W Resistor (Size 0805) Panasonic ERJ-6ENF1.50K	6	R3, R4, R5, R6, R7, R8
2 mA Green LED, PC Board Side Mount IDI 5350T5LC	3	L3, L4, L5
2 mA Yellow LED, PC Board Side Mount IDI 5350T7LC	1	L2
2 mA Red LED, PC Board Side Mount IDI 5350T1LC	1	L1
25.0000 MHz SMT Crystal, Parallel Res 18 pF Epson Amer MA-506 25.000M-AD Epson Amer MA-406 25.000M-G	1	X1
25.0000 MHz HC-49/U Crystal, Parallel Res 18 pF Ecliptek EC250-25.0000	1	X2
Quad 2:1 Transformer, 330 µH Primary, 1500V Valor ST6115 Pulse PE-69001 Bel S553-1204-00	1	U2
CY7C971 100BASE-T4/10BASE-T Transceiver Cypress Sem. CY7C971-NC	1	U1

Appendix B. Parts List (continued)

Description, Vendor, Part Number	Qty	Reference Designator
LT1117 3.3V Regulator Linear Tech. LT1117CST−3.3	1	U5
RJ−45 Modular 8-Pin Shielded Jack Amp 555141−1	1	J1
DEC21140 Fast Ethernet PCI MAC Digital Sem. 21140−AA	1	U3
93C46 1K Serial EEPROM (8-Pin SOIC) National Sem. NM93C46M8	1	U4

Assembly Instructions

1. Assemble only 1 crystal (X1 or X2).

100BASE-T4 Ethernet Repeater

Background

This application note describes the design of a 100BASE-T4 Ethernet Network Repeater using the Cypress CY7C971 PHY and CY7C388A for the core logic. The repeater has the following features:

- 100-Mb/s Shared Bandwidth over Cat. 3 UTP
- 8 Unmanaged Ports
- Integrated Transmit Filters
- Compact Layout
- Low Latency

The function of the repeater is to create a logically shared communication channel between the end stations in the network. The end stations (computer, printer, etc.) communicate with the repeater over dedicated twisted pair links. The repeater listens to the signal being received on one port and "repeats" the restored signal to the other ports. *Figure 1* illustrates the function of the repeater in a 100BASE-T4 Ethernet Network. The repeater in this application note has eight communication ports.

The functional requirements of the 100BASE-T4 repeater are defined in the IEEE 802.3u Standard "MAC Parameters, Physical Layer, Medium Attachment Units and Repeater for 100 Mb/s Operation," Clause 27. The repeater functional requirements are summarized below:

- Detect port activity and receive Ethernet packets
- Restore the shape, amplitude, and timing of the received signals prior to retransmission

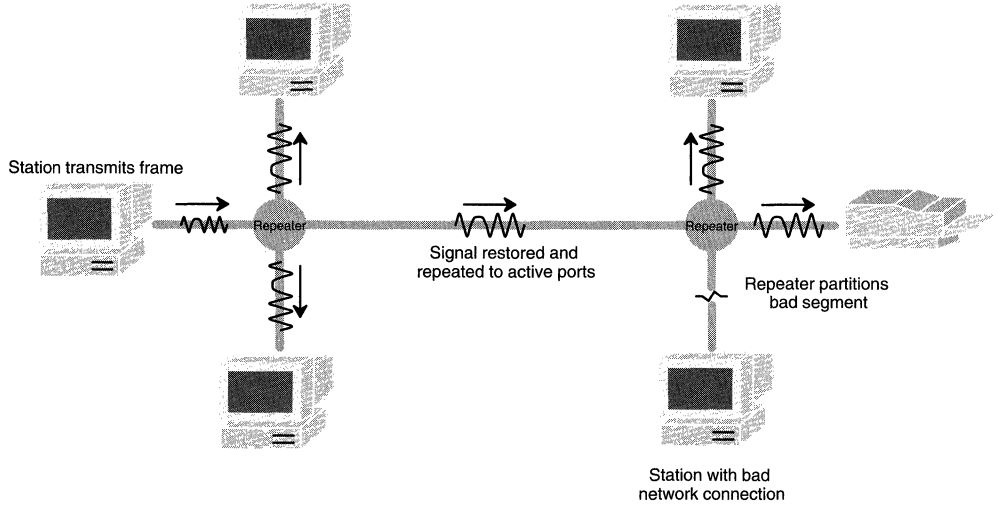


Figure 1. Ethernet Network Built with Repeaters

- Regenerate preamble sequence and prepend it to the received frame
- Forward the Ethernet frame to each of the ports
- Detect collisions between ports and generate jam sequence to all ports
- Protect network from long carrier events (jabber) and repeated collisions (partition)
- Allow installation (removal) of station without network disruption
- Provide basic port control (enable/disable)

Repeater Block Diagram

A block diagram of the 8-port repeater is shown in *Figure 2*. The CY7C971 functions as the physical layer device that interfaces the digital core logic to the twisted-pair medium. Each CY7C971 requires a quad 1:2 transformer for electrical isolation from the medium. The core logic is implemented with a CY7C388A FPGA. This device takes care of the basic repeater functions such as data retiming, sequence generation, and port control.

CY7C971

The CY7C971 (see *Figure 3*) has a special low latency repeater mode that is enabled when the MODE pin is LOW. In this mode, the MII (Media Independent Interface), PCS (Physical Coding Sublayer), and 10BASE-T are disabled. Only the 100BASE-T4 PMA (Physical Medium Attachment) circuits are active. These circuits perform the analog functions required to interface to the twisted-pair media such as transmit filtering, adaptive equalization, and clock recovery. A block diagram of the PMA interface is shown in *Figure 4*.

Media Dependent Interface (MDI)

The CY7C971 provides a simple interface to the 8-pin modular RJ−45 jack. No expensive external filters or components are necessary because all transmit filtering and equalization are performed on-chip. A quad 2:1 transformer for electrical isolation and termination resistors to match the cable impedance are all that is required.

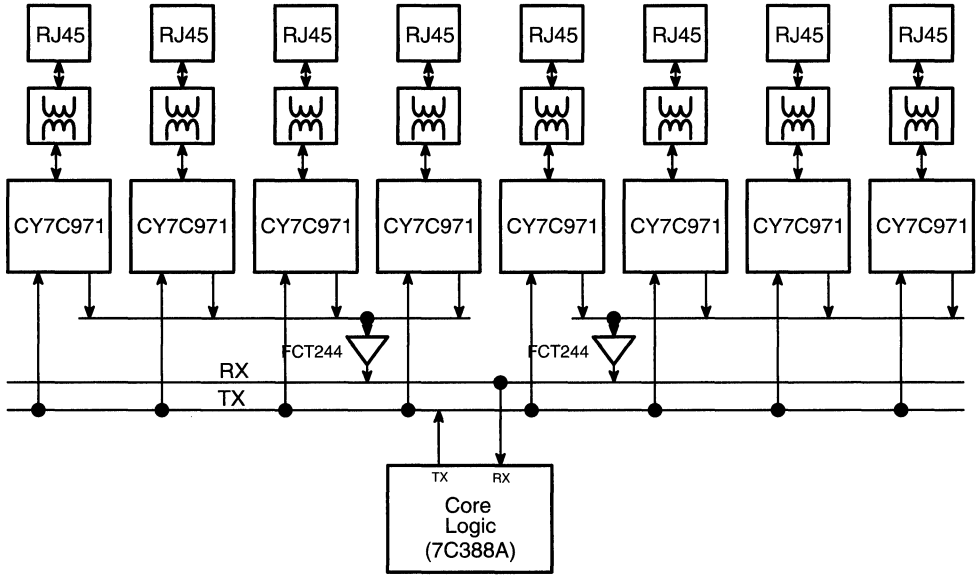


Figure 2. Repeater Block Diagram

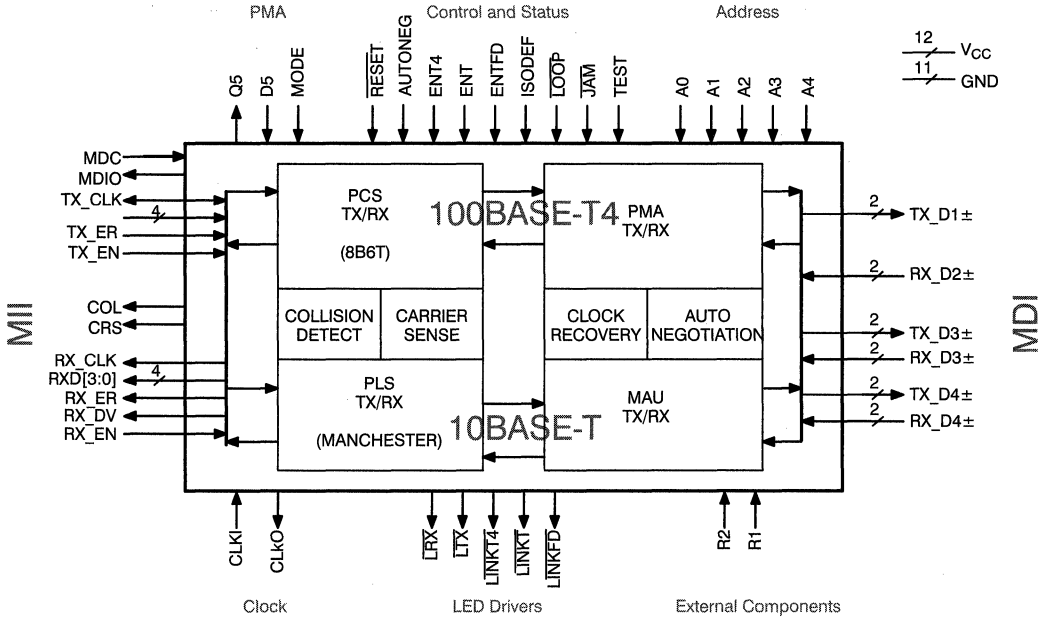


Figure 3. CY7C971 Block Diagram

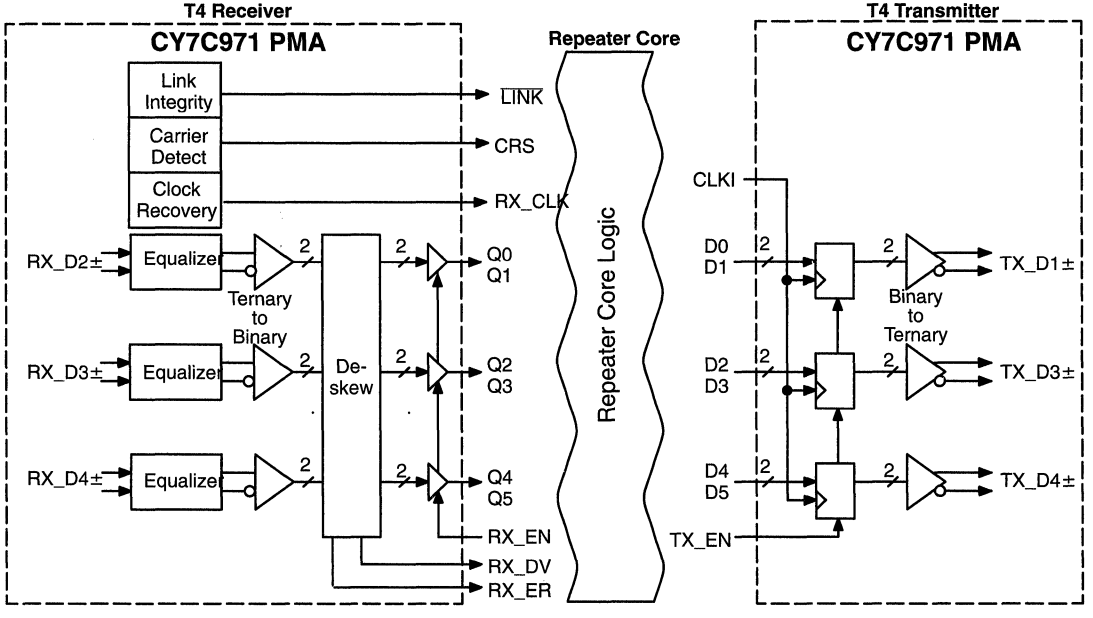


Figure 4. CY7C971 PMA Interface

The output buffer design uses a feedback voltage driver that minimizes power consumption and controls the common-mode output voltage. The transformer provides sufficient common mode rejection over the frequencies of interest so that an external common mode choke is not needed. *Figure 5* shows a schematic of the media interface with the CY7C971.

The characteristic impedance of the twisted pair medium is a nominal 100Ω. The 1:2 transformer reduces (by the square of the turns ratio) medium load impedance to 25Ω on the primary (971) side. The termination resistors and the output buffer impedance together form a matching 25-ohm load. The matching load insures that maximum signal is transferred to the medium and minimizes reflections due to impedance mismatch.

The center taps on the media side of the transformer are connected to the chassis ground through 220-pF (minimum) high-voltage (2 KV) capacitors. These capacitors help absorb common mode noise that is picked up or generated on the twisted pair medium. The capacitors must be capable of withstanding the isolation requirements specified in the 100BASE-T4 standard. High voltage ceramic disc capacitors are economical and work well in this application.

The high precision currents needed for the transmit DAC and equalizer are derived from the external 10KΩ 1% resistor on pins R1 and R2. An internally generated band-gap voltage reference is used by the CY7C971 for all internal reference voltages.

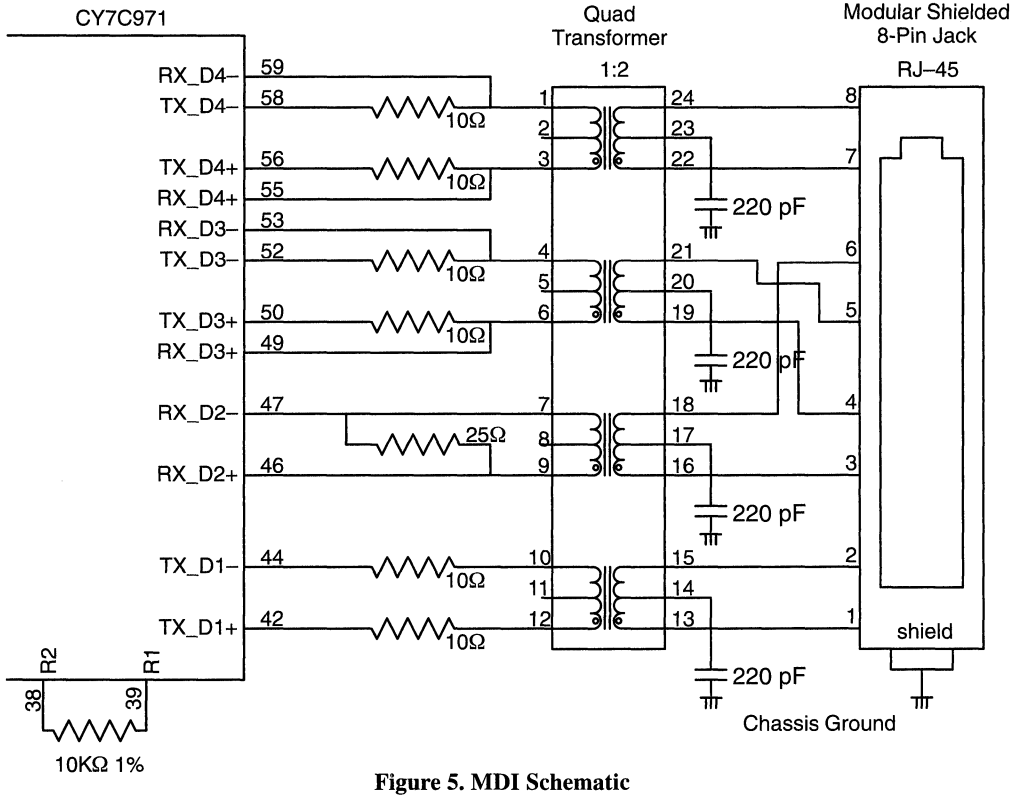


Figure 5. MDI Schematic

LED Pins

Figure 6 shows how the LED pins connect to the LEDs. The $\overline{\text{LINKT4}}$ pin indicates when the CY7C971 is in the link pass state for 100BASE-T4. The CY7C971 will enter a link pass state when properly formed technology dependent link integrity pulses are received from the medium. The $\overline{\text{LINKT}}$ and $\overline{\text{LINKFD}}$ signals remain inactive.

Configuration Pins

The configuration pins are wired for the repeater application as shown in *Figure 7*. The MODE pin is tied LOW to force the CY7C971 into 100BASE-T4 PMA mode. PMA mode disables the MII, PCS (Physical Coding Sublayer), and 10BASE-T. The 100BASE-T4 PMA performs all of the analog functions required to interface to 4 pair Cat. 3 UTP.

The ENT4 pin is wired HIGH to enable 10BASE-T4. The ENT and ENFD pins are wired LOW to disable 10BASE-T and Full Duplex operation. The AUTONEG pin is wired to a header block and pull-up. When a jumper is installed in the header block, Auto-Negotiation is disabled. When the jumper is absent, Auto-Negotiation is enabled.

The ISODEF (Isolate Default) pin is tied LOW in order to force the CY7C971 to power up with the MII ready for normal operation (not isolated). This

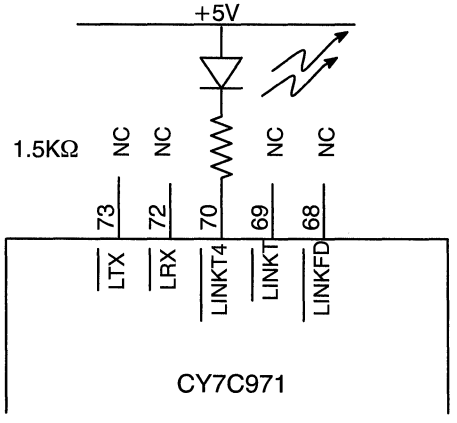


Figure 6. LED Pins

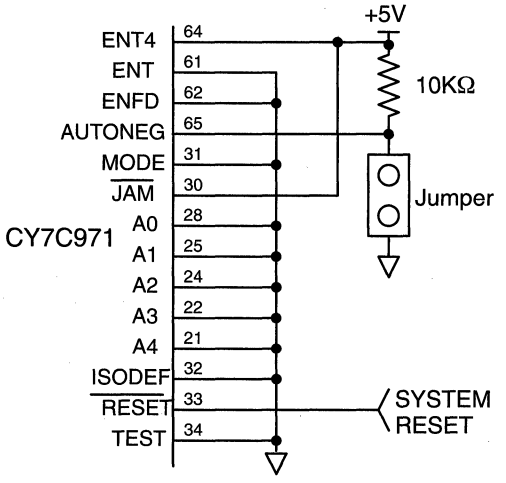


Figure 7. Configuration Pins

repeater application does not use the management port. The address pins can be assigned any address configuration.

The Test pin is tied LOW to permanently disable the 971 test mode. Test mode is used for factory ATE testing only.

The $\overline{\text{RESET}}$ pin should be connected to the system reset pin from the core logic. A system reset is issued at power-up or when the reset button is pushed. If a port is disabled by the core logic, the reset to the port will be active.

Layout Considerations

The repeater design is simple enough to fit on a small 7.75 in x 6.0 board using top-side-only placement. A four-layer PCB construction with dedicated power and ground planes is recommended. The CY7C971 requires a 5V supply. *Figure 8* shows an example of component placement.

The media interface components can be neatly placed in-line with the CY7C971. 0.027 µF decoupling capacitors are used on the CY7C971 power pins. These 0805 SMT capacitors are placed in a row as close to the pins as possible. The termination resistors fit neatly in a row behind the decoupling capacitors. The CY7C971 media interface and power

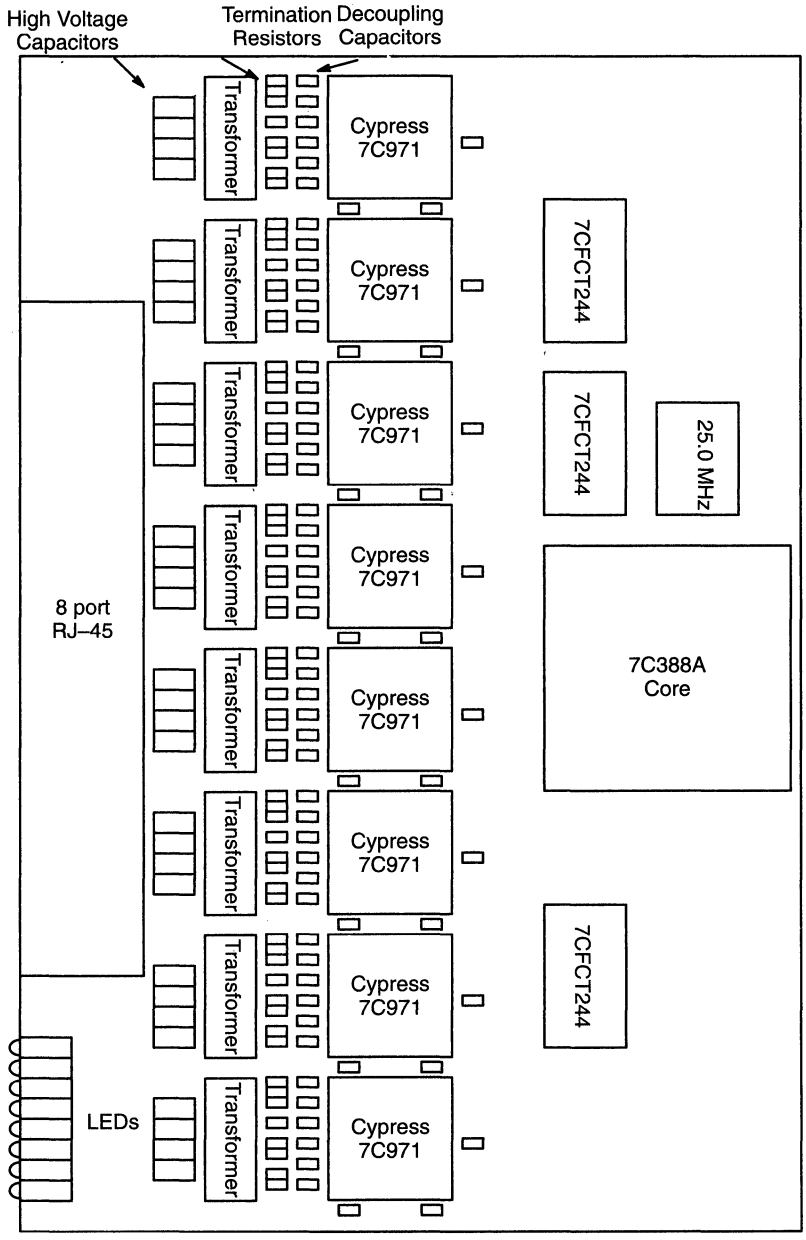


Figure 8. Component Placement

pins are placed in such a way to minimize the use of vias and simplify board layout.

Core Logic

Figure 9 shows a block diagram of the repeater core logic. The blocks perform functions as follows:

- **Port N.** Synchronizes signals and provides control signals to each port, along with detecting jabber and partition conditions.
- **Selection and Clock MUX.** Selects the receive clock from the incoming port and provides a common receive clock for use in retiming the incoming data.
- **RX FIFO.** Used for temporary storage and to retime the incoming data to TX_CLK.
- **Bad Symbol, Jam, Idle, Preamble Generator.** Provides the special characters that are transmitted during different conditions.
- **Output Register.** Provides temporary storage of outgoing data along with retiming to the TX_CLK.
- **Repeater State Machines and Logic.** Controls port selection during data reception. Also, provides collision detection and handling. Included in this block is the control of two expansion ports for use in the design of a stackable repeater.

The core logic is written in Verilog and fills 7K gates of a Cypress CY7C388A 8K pASIC.

Conclusion

This application note covers the major issues for a 8-port 100BASE-T4 Repeater design using the CY7C971 100BASE-T4/10BASE-T Transceiver and CY7C388A 8K FPGA. The high degree of integration in the CY7C971 keeps the number of external components to a minimum, helping to reduce system cost and design effort.

The complete repeater schematics and a bill of materials are available from Cypress Semiconductor. More information on the CY7C971 can be found in the data sheet. For more information on 100BASE-T4, MII, and Auto-Negotiation standards, consult the IEEE 802.3u document: "MAC Parameters, Physical Layer, Medium Attachment Units and Repeater for 100Mb/s Operation."

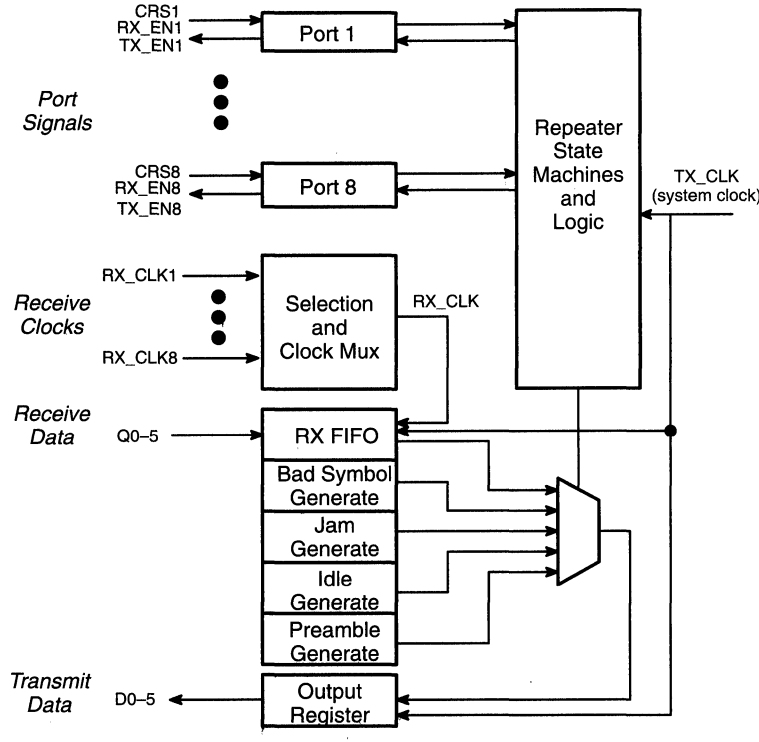


Figure 9. Core Logic

Interfacing with the SST™

This application note describes how to interface the CY7B951 SONET/SDH Serial Transceiver (SST™) with other physical-layer devices. The SST performs clock and data recovery from a SONET/SDH (Synchronous Optical NETwork/Synchronous Digital Hierarchy) 51.84 Mb/s or 155.52 Mb/s interface and can be used in a variety of SONET and ATM applications. The application note will begin with a brief introduction to the SST. Next, interface examples will be given that illustrate how to connect the SST to three different ATM controller devices; the first from PMC-Sierra called the PM5345 SUNI, the second, also from PMC-Sierra, called the S/UNI-LITE, and the third from Integrated Telecom Technologies (IgT) called the WAC-013.

Introduction

The CY7B951 SST is used in SONET/SDH applications to recover clock and data information from a 155.52-MHz or 51.84-MHz NRZ (Non Return to Zero) or NRZI (Non Return to Zero Invert on ones) serial data stream. This device also provides a bit-rate Transmit Clock, from a byte-rate source through the use of a frequency multiplier Phase-Locked Loop (PLL), and differential data buffering for the Transmit side of the system (see *Figure 1*). The pinout is shown in *Figure 2*.

Operating Frequency

The SST operates at either of two frequency ranges. The MODE input selects which of the two frequency ranges the Transmit frequency multiplier PLL and the Receive clock and data recovery PLL will operate. When MODE is connected to V_{CC}, the highest operating range of the device is selected. A 19.44-MHz ±1% source must drive the REFCLK input and the transmit PLL will multiply this rate by 8 to provide an output clock that operates at 155.52 MHz ±1%. When the MODE input is connected to ground (GND), the lowest operating range of the

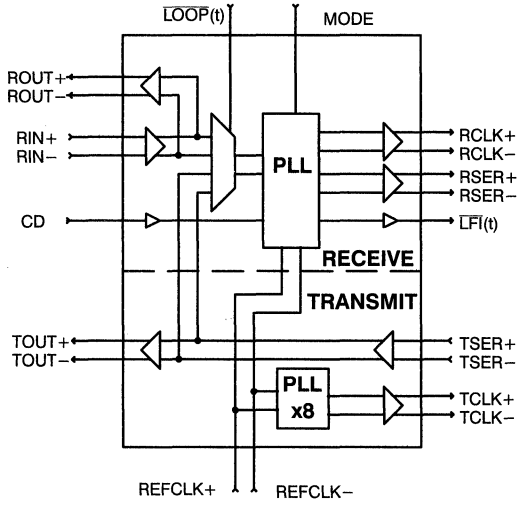


Figure 1. SST Block Diagram

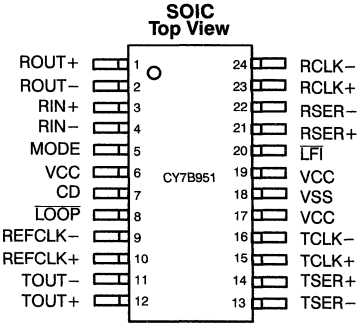


Figure 2. SST Pinout

device is selected. A 6.48-MHz ±1% source must drive the REFCLK inputs and the transmit PLL will multiply this rate by 8 to provide an output clock that operates at 51.84 MHz ±1%. In addition, when the MODE input is left unconnected or forced to approximately $V_{CC}/2$, the device enters Test Mode.

Transmit Functions

The Transmit section of the SST contains a PLL that takes a REFCLK input and multiplies it by 8 (REFCLK*8) to produce a PECL (Pseudo ECL or Positive ECL) differential output clock (TCLK±). The Transmitter has two operating ranges that are selectable with the three-level MODE pin, as explained above. The SST Transmit frequency multiplier PLL allows low-cost byte-rate clock sources to be used to time the upstream serial data transmitter.

The REFCLK± inputs can be configured in three different ways. When both REFCLK+ and REFCLK− are connected to a differential 100K compatible PECL source, the REFCLK input will behave as a differential PECL input. When either the REFCLK− or the REFCLK+ input is at a TTL LOW, the other REFCLK input becomes a TTL-level input allowing it to be connected to a low-cost TTL crystal oscillator. The REFCLK input structure, therefore, can be used as a differential PECL input, a single TTL input, or as a dual TTL clock multiplexing input.

The Transmit PECL differential input pair (TSER±) is buffered by the SST yielding the differential data outputs (TOUT±). These outputs can be used to directly drive transmission media such as Printed Circuit Board (PCB) traces, optical fiber drivers, twisted pair, or coaxial cable.

Receive Functions

The primary function of the Receiver is to generate recovered clock (RCLK±) and data (RSER±) signals from the incoming differential PECL data stream (RIN±). These built-in line receiver inputs, as well as the TSER± inputs mentioned above, have a wide common-mode range (2−5V) and the ability to receive signals with as little as 50 mV differential voltage. They are compatible with all PECL signals and any copper media (such as coaxial cable or twisted pair).

The clock recovery function is performed using an embedded PLL. The recovered clock is not only passed to the RCLK± outputs, but also used internally to sample the input serial stream in order to recover the data pattern. The Receive PLL uses the REFCLK input as a byte-rate reference. This input is multiplied by 8 (REFCLK*8) and is used as a bit-rate reference in comparison to the recovered clock to improve PLL lock time, and to provide a center frequency for operation in the absence of input data stream transitions. The Receiver can recover clock and data in two different frequency ranges depending on the state of the three-level MODE pin, as explained earlier. To ensure accurate data and clock recovery, REFCLK*8 must be within 1000 ppm of the transmit bit rate. The standards, however, specify that the REFCLK*8 frequency accuracy be within 20−100 ppm.

The differential input serial data (RIN±) is not only used by the PLL to recover the clock and data, but it is also buffered and presented as the PECL differential output pair ROUT±. This output pair can be used as part of the transmission line interface circuit for base-line wander compensation, improving system performance by providing reduced input jitter and increased data eye opening.

Carrier Detect (CD) and Link Fault Indicator ($\overline{\text{LFI}}$) Functions

The Link Fault Indicator ($\overline{\text{LFI}}$) output is a TTL-level output that indicates the status of the Receiver. This output can be used by an external controller for Loss of Signal (LOS), Loss of Frame (LOF), or Out of Frame (OOF) indications. $\overline{\text{LFI}}$ is controlled by the Carrier Detect (CD) input, the internal Transitions Detector, and the PLL Out of Lock (OOL) circuitry.

The CD input may be driven by external circuitry that is monitoring the incoming data stream. Optical modules have CD outputs that indicate the presence of light on the optical fiber and some copper-based systems use external threshold detection circuitry to monitor the incoming data stream. The CD input is a 100K PECL-compatible signal that should be held HIGH when the incoming data

stream is valid. When CD is pulled to a PECL LOW, the $\overline{\text{LFI}}$ output will transition LOW, the Receiver PLL will align itself with the REFCLK*8 frequency, and the recovered data outputs (RSER) will remain LOW regardless of the signal level on the Receive data stream inputs (RIN).

In addition, the SST has a built-in transitions detector that also checks the quality of the incoming data stream. The absence of data transitions can be caused by a break in the transmission media, a problem at the transmitter end of the media, or a problem with the transmit or receive media coupling hardware. The SST will detect a quiet link by counting the number of bit times that have passed without a data transition. A bit time is defined as the period of RCLK±. When 512 bit times have passed without a data transition on RIN±, $\overline{\text{LFI}}$ will transition LOW. The Receiver will assume that the serial data stream is invalid and, instead of allowing the RCLK± frequency to wander in the absence of data, the PLL will lock to the REFCLK*8 frequency. This will insure that RCLK± is as close to the correct link operating frequency as the REFCLK accuracy. $\overline{\text{LFI}}$ will be driven HIGH again and the Receiver will recover clock and data from the incoming data stream when the transition detection circuitry determines that at least 64 transitions have been detected within 512 bit times.

The Transition Detector can be turned off by pulling the CD input to a TTL LOW (≤0.8V). When CD is pulled to a TTL LOW, the $\overline{\text{LFI}}$ will only be driven LOW if the incoming data stream frequency is not within 1000 ppm of the REFCLK*8 frequency. $\overline{\text{LFI}}$ LOW in this case will only indicate that the Receiver PLL is Out of Lock (OOL). When $\overline{\text{LFI}}$ is left unconnected, an internal pull-down resistor will pull this input to ground.

Loop Back Testing

The TTL level $\overline{\text{LOOP}}$ pin is used to perform loopback testing. When $\overline{\text{LOOP}}$ is asserted (held LOW) the Transmitter serial inputs (TSER±) are used by the Receiver PLL for clock and data recovery. This allows in-system testing to be performed on the entire device except for the differential Transmit drivers (TOUT±) and the differential Receiver inputs (RIN±). For example, an ATM controller can present ATM cells to the input of the ATM cell processor and check to see that these same cells are received. When the $\overline{\text{LOOP}}$ input is deasserted (held HIGH) the Receive PLL is once again connected to the Receiver serial inputs (RIN±).

The $\overline{\text{LOOP}}$ feature can also be used in applications where clock and data recovery are to be performed from either of two data streams. In these systems the $\overline{\text{LOOP}}$ pin is used to select whether the TSER± or the RIN± inputs are used by the Receive PLL for clock and data recovery.

Power-Down Modes

There are several power-down features on the SST. Any of the differential output drivers can be powered down by either tying both outputs to V_{CC} or by simply leaving them unconnected where internal pull-up resistors will force these outputs to V_{CC}. This will save approximately 4 mA per output pair in addition to the associated output current. If the TOUT± or ROUT± outputs are tied to V_{CC} or left unconnected, the Transmit buffer or Receive buffer path respectively will be turned off. If the TCLK± outputs are tied to V_{CC} or left unconnected the entire Transmit PLL will be powered down.

By leaving both the RCLK± and RSER± outputs unconnected or tied to V_{CC} the entire Receive PLL is turned off. Even though the Receive PLL may be turned off, the ($\overline{\text{LFI}}$ will still reflect the state of the CD input. This feature can be used for aggressive power management.

Interfacing with the PM5345 (SUNI)

The PM5345 is used in ATM applications for SONET frame processing, ATM cell processing, and error monitoring. The PMC-Sierra SUNI device requires Receive serial data aligned with a bit-rate clock. These signals need to be supplied through the RXD± and RXC± inputs respectively. A 155.52-MHz PECL Transmit clock (TXC±) is required to provide PM5345 transmit side clocking. For copper-based systems, the TXD± outputs must be buffered in order to drive transmission lines with low impedances. Lastly, a LOS detection is required from the clock and data recovery engine to

aid in the determination of the LOS, LOF, and OOF error conditions reported by the SUNI device. This signal is brought in through the SUNI GPIN (General Purpose Input). Before the introduction of the SST, clock and data recovery devices were interfaced to the PMC-SUNI as shown in *Figure 3.*

Figure 4 shows the SST signal connections with the PMC-Sierra PM5345 SUNI. The SST, together with the PM5345, provides a complete Physical layer interface. The Receive section of the SST provides serial SONET/SDH data at 155.52 Mb/s to the receive section of the PM5345 (RXC± and RXD±). The Transmit section of the SST provides the transmit side 155.52-MHz clock that is used by the PM5345 TXCI± input by multiplying a 19.44-MHz oscillator by eight. This function eliminates the need for an expensive 155.52-MHz oscillator to be used in the system. The SST buffers the TXD± output signals from the SUNI device for driving copper-based systems or for improved operation in fiber-based systems.

The $\overline{\text{LFI}}$ output is used to drive the GPIN input. This $\overline{\text{LFI}}$ output will transition LOW when any of the following occur: the CD (Carrier Detect) input transitions LOW, the frequency of the incoming data is outside of the lock range of the Receive PLL,

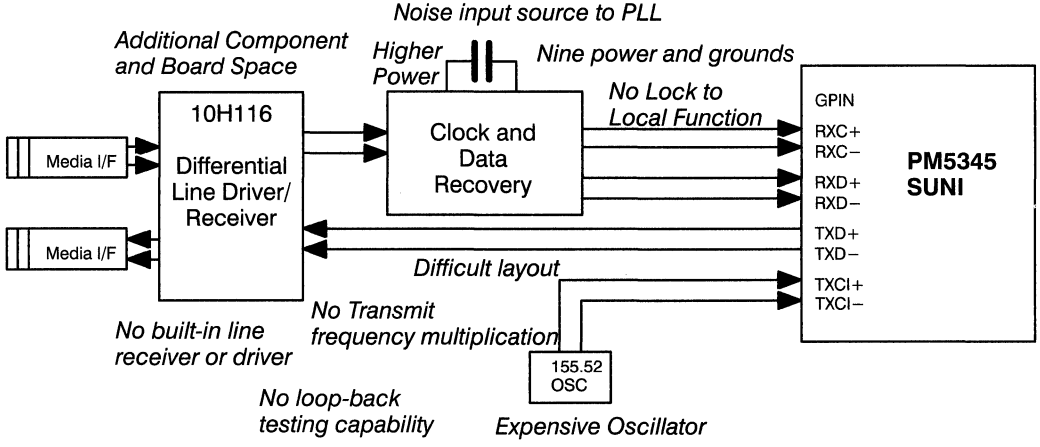


Figure 3. Typical SUNI interface without the Use of the SST

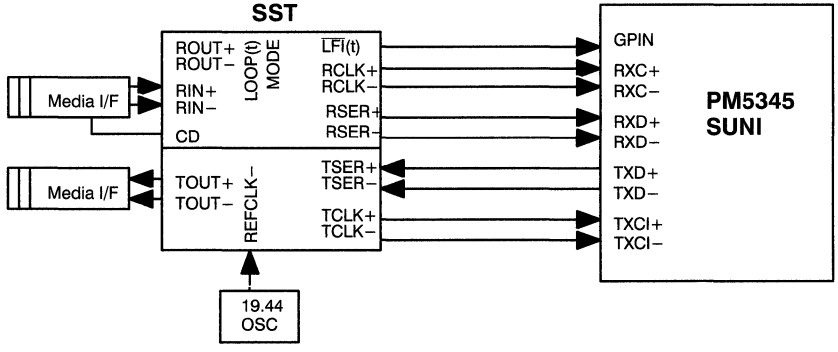


Figure 4. SST to PMC-Sierra PM5345 SUNI Connection Diagram

or there have been no transitions in the incoming data stream for the last 512 bit times. Additionally, when the CD input is forced LOW by an output from a source such as the signal detect of an optical module or an external transition detection circuitry for copper-based systems, the SST will force the RSER± outputs LOW. This will aid the SUNI device in the determination of the LOS state and minimize the length of time needed to determine an error condition.

Figure 5 shows an electrical interface of the SST to the PMC-SUNI device. Each SST PECL output is AC coupled into the SUNI inputs with a .01-µF capacitor, and is loaded with an 80Ω pull-up resistor and a 130Ω pull-down resistor. This scheme allows the SUNI device to self-bias (since the SUNI has a bias circuit built into each PECL input) its inputs and also provides the SST outputs with 50Ω terminations to approximately V_{CC} − 2V. The termination resistors are bypassed with .01-µF capacitors to provide high-speed switching current. For PCB trace impedances higher than 50Ω, the terminating resistors should be scaled accordingly. For example, a 100Ω transmission line would require a pull-up resistor of 160Ω and a pull-down resistor of 260Ω. Terminations for the SST outputs (TCLK, RCLK, RSER) should be placed as close to the SUNI as possible.

The TXD± outputs require different termination resistors values. The ideal biasing voltage for TXD± is 4.2V. This bias is achieved by connecting a 62Ω pull up to TAVD and a 330Ω pull down to GND at the end of the termination line connecting

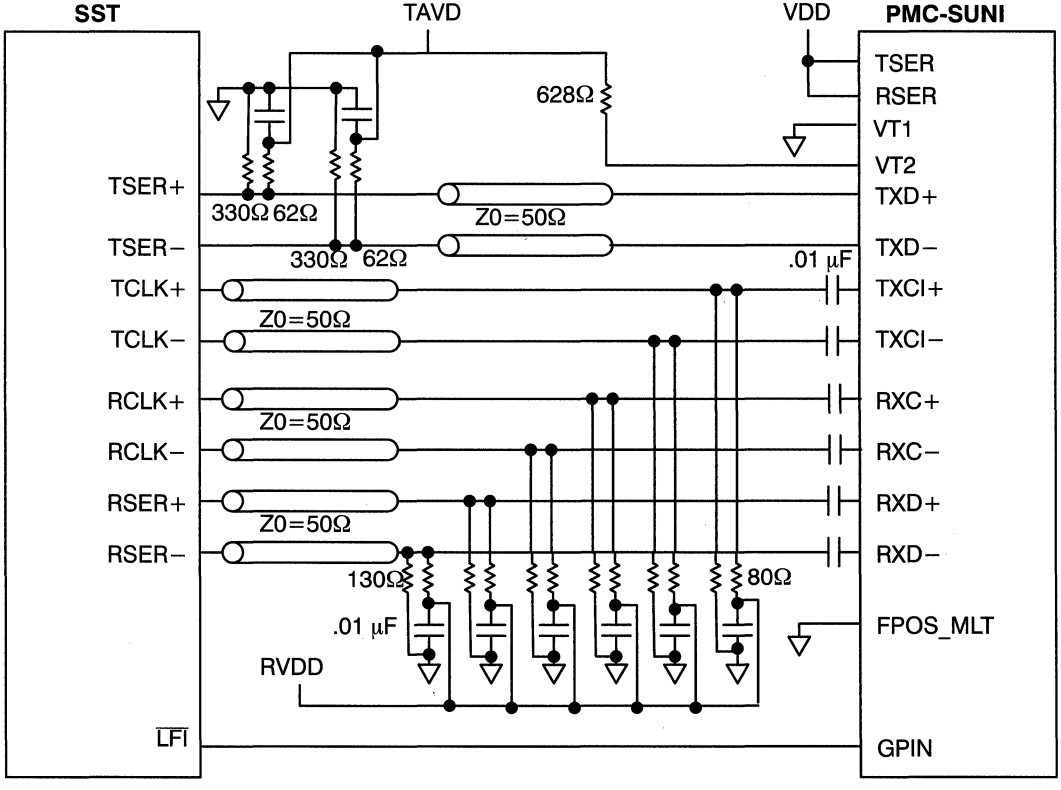


Figure 5. High Performance SST to PMC SUNI Interface

TXD± and TSER±. These resistor values are calculated based on Z_0 = 50Ω. For PCB trace impedances higher than 50Ω, the terminating resistors should be scaled accordingly. For example, a 100Ω transmission line would require a pull-up resistor of 120Ω and a pull-down resistor of 636Ω. In addition, the VT2 resistor should also be scaled from 628Ω to 1260Ω when using 100Ω trace impedances. In general, $R_{VT2} = 12.564 * Z_0$.

Interfacing with the PM5346 (S/UNI-LITE)

The PM5346 is another PMC-Sierra product used in ATM systems for clock and data recovery, SONET frame processing, ATM cell processing, and error monitoring. Its small package size makes it more desirable than the PM5345 in cases where not all of the SONET frame processing functions of the PM5345 are needed. For performance reasons, the PLL of S/UNI-LITE can be bypassed and the SST can be used to perform clock and data recovery functions for the S/UNI-LITE.

Figure 6 shows how to interface the SST to the S/UNI-LITE. When RBYP is tied HIGH, the internal PLL of the S/UNI-LITE is disabled and RRCLK± is used to sample RXD±. In this configuration, the SST is used to supply the bit-aligned RRCLK. This is achieved by connecting RCLK± to RRCLK± and RSER± to RXD± using four equal-length traces. Each of these traces has an 80Ω pull-up to RVDD and a 130Ω pull-down to GND. These termination resistors are bypassed with .01-µF capacitors to satisfy the high-speed switching current

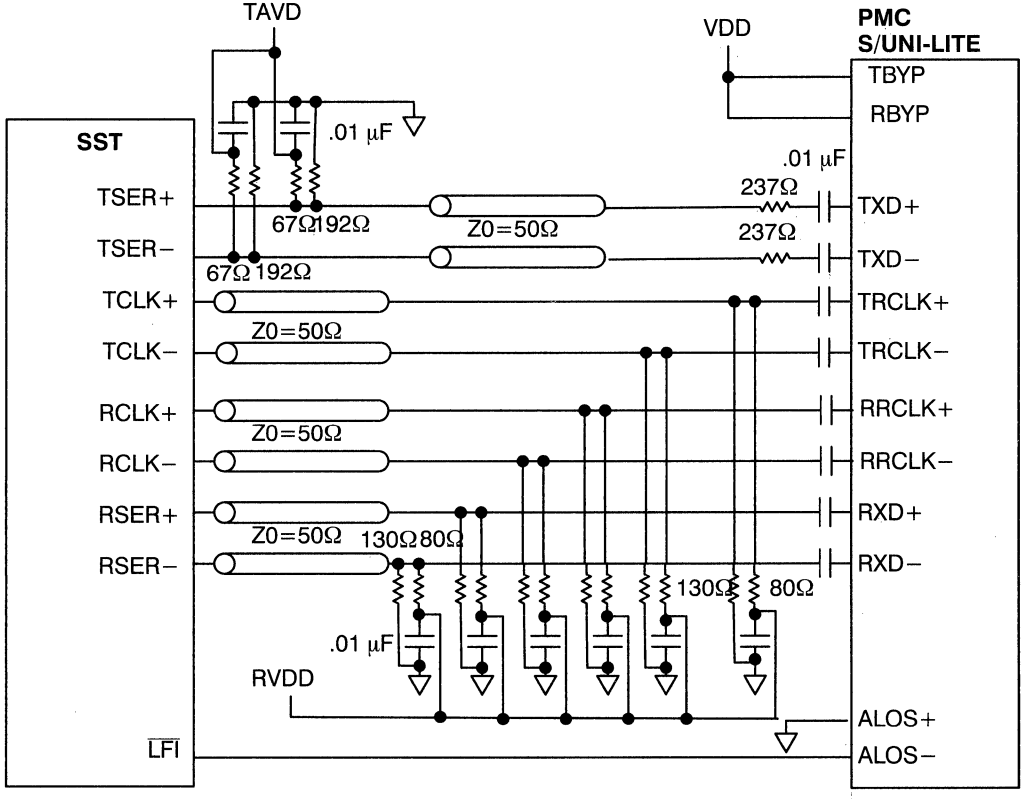


Figure 6. High Performance SST to PMC S/UNI-LITE Interface

requirements. A .01-µF DC-blocking capacitor is used in series with the transmission line to allow the S/UNI-LITE to self-bias its inputs (since the S/UNI-LITE, like the SUNI, also has bias circuits built into each PECL input). All these passive components are placed close to the S/UNI-LITE.

In the same way, the transmit side PLL of the S/UNI-LITE can also be disabled. When TBYP is tied HIGH, the clock multiplication function of the S/UNI-LITE is disabled and the 155.52-MHz or 51.84-MHz clock received from either RRCLK± or TRCLK± is used for clocking the transmit portion of the S/UNI-LITE. If the LOOPT bit of the Master Control register of the S/UNI-LITE is 1, RRCLK will be used and when the LOOPT bit is 0, TRCLK± will be used. TRCLK± is supplied by TCLK± of the SST. The termination/biasing circuit used for this TRCLK connection is the same as that used in the RXD± and RRCLK± connections described previously. These termination/biasing circuits should also be placed as close to the S/UNI-LITE as possible.

For the TXD± to TSER± connections, a 237Ω source resistor in series with a .01-µF capacitor placed closed to the S/UNI-LITE side is used with a 67Ω pull-up to TAVD and a 192Ω pull-down to GND placed close to the SST side to provide the necessary termination and biasing.

Interfacing with the IgT WAC-013.

The Integrated Telecom Technology (IgT) WAC-013 provides SONET frame processing, ATM cell processing, and error monitoring. The IgT device requires differential PECL Receive data (RS_SER_DATA) aligned with a differential PECL bit-rate clock (RS_SER_CLK). These signals represent the recovered clock and data from a SONET/SDH STS-3/STM-1 data stream of 155.52 Mb/s or a SONET STS-1 data stream of 51.84 Mb/s. The WAC-013 also requires a bit-rate transmit-clock (TS_SER_CLK) for Transmit Side clocking. The transmit data (TS_SER_DATA) should also be buffered for driving low-impedance transmission lines or copper transmission media. Prior to the introduction of the SST, clock and data recovery devices were connected to the WAC-013 as shown in *Figure 7*.

Figure 8 shows the SST signal connections with the IgT WAC-013. The SST, together with the WAC-013, provides a complete physical-layer interface. The Receive section of the SST provides serial SONET/SDH data at 155.52 Mb/s or 51.84 Mb/s (depending on the state of the SST MODE pin) to the Receive section of the IgT RS_SER_DATA and RS_SER_CLK inputs. The Transmit section of the SST provides the bit-rate clock (TS_SER_CLK) and Transmit buffering of the TS_SER_DATA outputs. The SST multiples a 19.44-MHz reference

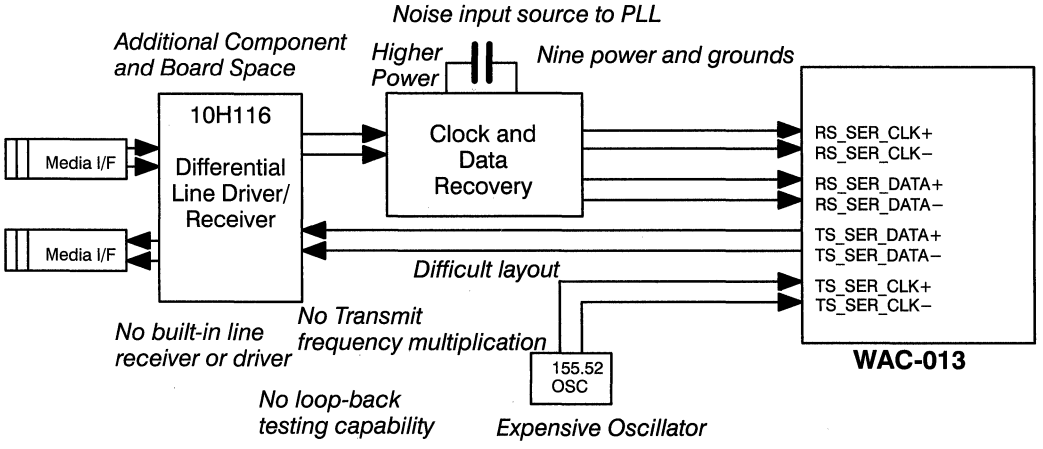


Figure 7. Typical WAC-013 interface without the Use of the SST

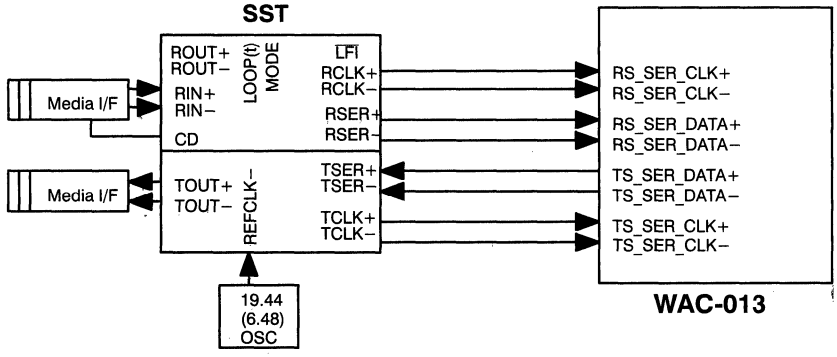


Figure 8. SST to IgT WAC-013 Connection Diagram

clock (6.48-MHz for STS-1 applications) by eight to produce the 155.52-MHz (51.84-MHz) transmit clock. This frequency multiplication function eliminates the need for an expensive 155.52-MHz crystal oscillator.

Figure 9 shows the electrical interface of the SST to the WAC-013. The outputs are loaded and terminated with 80Ω pull-up resistors and 130Ω pull-down resistors at the load. This provides a 50Ω termination to $V_{CC}-2V$. These resistors are also bypassed with a .01-μF capacitor to provide high-speed switching current. For PCB trace impedances higher than 50Ω, the terminating resistors should be scaled accordingly. For example, a 100Ω transmission line would require a pull-up resistor of 160Ω and a pull-down resistor of 260Ω.

Conclusion

The interface examples shown in this note demonstrate how to connect the SST to the PMC-Sierra PM5345 SUNI, the PMC-Sierra PM5346 S/UNI-LITE, and the IgT WAC-013. Together these devices provide a complete physical-layer solution for ATM applications over SONET/SDH at 155.52 Mb/s and 51.84 Mb/s. The SST greatly simplifies the physical-layer implementation with its ability to generate a Loss of Signal indication, its capability to lock to the local reference clock during error conditions, and its capacity to buffer the transmit data stream for driving low-impedance transmission lines. The SST also reduces the cost of physical-layer implementations by eliminating the need for a 155.52-MHz crystal oscillator with its ability to multiply a byte-rate clock to provide the bit-rate transmit source. Cypress's expertise in PLL-based clock and data recovery as well as the added features of the SST provide designers with the capacity to create simple, low cost, and robust ATM physical-layer designs.

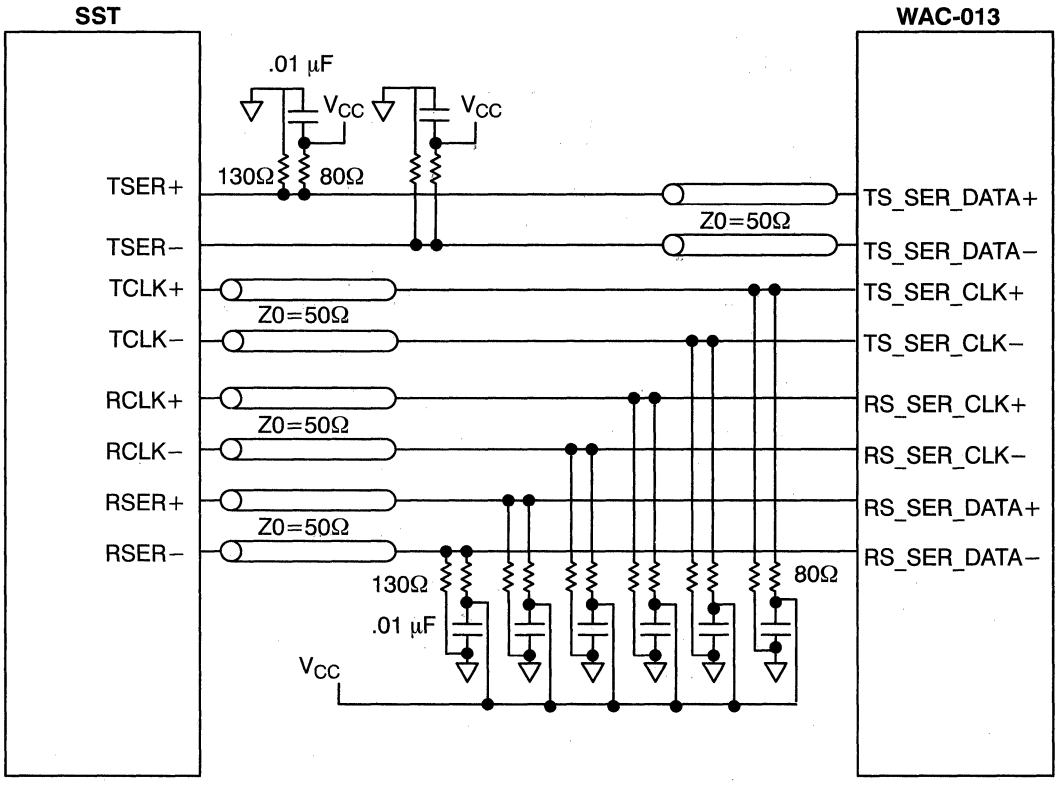


Figure 9. High Performance SST to WAC-013 Interface

SST is a trademark of Cypress Semiconductor Corporation.

Frequently Asked Questions about HOTLink™

The following questions are frequently asked by customers who are evaluating HOTLink™ products. These cursory answers will serve as an introduction for each topic. Separate application notes cover these topics in more complete detail.

1. How far can HOTLink communicate over various media?

HOTLink has no intrinsic distance limit. The two issues that determine the distances over which data can be sent using HOTLink are: (1) the choice of interconnect media (fiber-optic cable, coaxial cable, twisted-pair cable, etc.); and (2) the jitter that accumulates or is injected while the data is in transit over the selected media.

HOTLink can drive all standard fiber-optic interface modules that support standard PECL interface signals. These electro-optical modules are suitable for communicating over distances from a few meters to several kilometers. Fiber-optic interconnect offers the longest distances and the lowest interference potential of all transmission media.

For lower-cost applications, HOTLink can directly drive wire transmission lines. The main distance determining factors when using wire links are related to the characteristics of the cable. Wire transmission lines have significant frequency-dependent attenuation that causes jitter as a direct function of the data rate and the media length. Uncompensated transmission line lengths are limited much more by jitter (and the jitter tolerance of the receiver) than by actual signal attenuation. The detrimental effect of jitter can be lessened with the addition of a suitable attenuation compensation filter that matches the attenuation characteristics of the cable. This filter trades receiver differential voltage amplitude for jitter reduction and increases the possible transmission distance. When using wire transmission lines, other issues beyond transmission distance often determine transmission line suitability. These issues include both radiated emissions and susceptibility to external disturbance that must be examined prior to selection of a link media type.

Some typical wire types and uncompensated transmission distances over which HOTLink can communicate are shown in *Table 1*. A simple compensation filter, built from passive components, can increase reliable transmission distance to more than twice these distances.

For more information see the application note "HOTLink Copper Interconnect—Maximum Length vs. Frequency."

Table 1. Coaxial Cable Types

Coaxial Cable	50Ω	75Ω	75Ω	93Ω
160 Mbaud	RG–58 A/U – 350 ft	RG–6 A/U – 900 ft	RG–59 A/U – 525 ft	RG–62 A/U – 675 ft
266 Mbaud	RG–58 A/U – 225 ft	RG–6 A/U – 600 ft	RG–59 A/U – 350 ft	RG–62 A/U – 400 ft
330 Mbaud	RG–58 A/U – 115 ft	RG–6 A/U – 500 ft	RG–59 A/U – 250 ft	RG–62 A/U – 325 ft

Table 2. Twisted Pair Cable Types

Shielded Twisted Pair	150Ω
160 Mbaud	IBM®–Type 1 – 550 ft
266 Mbaud	IBM–Type 1 – 350 ft
330 Mbaud	IBM–Type 1 – 275 ft

Unshielded Twisted Pair	UTP3	UTP5
160 Mbaud	140 ft	280 ft
266 Mbaud	80 ft	180 ft
330 Mbaud	60 ft	130 ft

2. Can the PECL inputs and outputs of HOTLink products be connected to ECL (–5.2V) products?

The +5.0V PECL inputs and outputs are directly compatible with true ECL (10K, 10KH, 100K, etc.) running on +5V power supplies. Connections between the HOTLink PECL I/O and ECL running on –5.2V is easily accomplished by capacitor-coupling the serial data lines. Details on this coupling technique are included in the Cypress application note "HOTLink Design Considerations."

3. What happens when the ECL inputs of the HOTLink Receiver are left open?

All of the ECL inputs on the HOTLink Receiver have internal pull-down resistors to assure that ECL-emitter follower outputs will see a positive input current (approximately 250 µA into the pin) at all normal ECL voltages. Thus, all single-ended ECL inputs (i.e., A/$\overline{B}$, SI, INB) will float to a logical LOW level. (These pull-downs will not sink enough current to act as the normal ECL output termination. They are only intended to prevent the emitter-follower oscillations caused by negative input-impedance that are possible in some less robust designs.) Open inputs will be interpreted as follows: A/$\overline{B}$ = LOW will cause the Receiver to accept data from the INB serial inputs; SI = LOW will cause the SO output to assume a LOW output state; INB = LOW will be interpreted as an input with no data (assuming A/$\overline{B}$ is also LOW). No data is interpreted as an error (RVS=HIGH & C0.7 in Encoded mode, and Qa–j outputs LOW in Bypass mode) and will cause the internal clock-synchronizer phase-locked loop (PLL) to track the REFCLK input frequency.

The internal resistor network used to pull the differential serial data inputs (i.e., INA± and INB±) will cause unconnected inputs to rest at approximately 2.0V. This resting voltage is a byproduct of the internal resistive attenuator used to enhance input-common mode range. If both inputs of a differential pair are left unconnected, the inputs will be in an undefined state and HOTLink receiver behavior will be unpredictable. Stray, non-differential noise that appears on these unconnected inputs will be amplified and interpreted as serial data. This will cause random parallel-data output changes, and may cause the PLL to wander or drift away from the REFCLK frequency. One input of an intentionally unused differential-pair should be terminated to V_{CC} through a 1–5 KΩ resistor to assure that no data transitions are accidentally created.

4. What special power-supply bypassing is required for HOTLink products?

HOTLink requires no special considerations for power-supply bypassing beyond that normally associated with high speed logic. This typically includes the use of a ground plane, a split V_{CC} plane, and multiple chip bypassing using RF quality capacitors. Each of the ground pins of a HOTLink IC should connect directly to the ground plane using short (<.25") traces and vias. All of the V_{CC} pins should connect to a V_{CC} pad under the HOTLink and then connect to the board V_{CC} through a single via. Connect one 22-nF capacitor for each V_{CC} pin directly from the pin to GND. For more information see the "Using Decoupling Capacitors" application note.

5. If the HOTLink Receiver is switched from INA to INB, how long will it take for the PLL to re-lock?

Assuming that the data on both INA and INB are within the ±0.1% frequency offset described in the HOTLink datasheet, the phase-locked loop (PLL) will acquire and lock to the new data stream within a few byte times. The exact time required involves statistical probabilities related to phase, frequency, and jitter, and cannot be exactly predicted. Empirical testing using normal data patterns shows that the time required to achieve absolute minimum phase error with the new data stream will vary from zero to about ten bytes.

An operational serial link will produce valid parallel data much earlier than the amount of time required to achieve minimum phase error, since instantaneous phase error is accommodated as jitter. The wide jitter tolerance offered by the HOTLink Receiver will minimize the time that data is incorrectly interpreted during phase acquisition. The larger problem facing a system protocol that allows switching of serial data streams, is byte synchronization (byte-framing). After the data-stream has been switched, it must be reframed. This requires that a K28.5 (or two K28.5s within five bytes if multibyte framing is enabled) must be received. The time that elapses before this happens depends on the system protocol and the timing of the data input switch. Correct data might not come out of the HOTLink Receiver for hundreds of byte times due to reframing regardless of speed of phase acquisition.

For more information, refer to the Receiver Data-Phase Acquisition Time section of the "HOTLink Jitter Characteristics" application note.

6. If the connection between the HOTLink Transmitter and Receiver is briefly interrupted, how long will it take for the PLL to re-lock?

The exact behavior of the HOTLink Receiver depends on the length and cause of the interruption. If the interruption is synchronous with the data (i.e., data bits disappear without any significant disturbance to the placement of the final few data transitions), and lasts for less than a few dozen bytes, it is probable that the PLL will relock on the very first bit. If the interruption is asynchronous (i.e., the timing of the final few transitions is disturbed) or if the synchronous interruption lasts longer than a few dozen bytes, the PLL will relock within the first one or two bytes after resumption of the data stream. If a long interruption occurs that is not synchronous to byte boundaries, the receiver may lose byte synchronization when the PLL relocks. In this case, the data will need to be reframed.

If the interruption is asynchronous, and the link interface allows noise to be injected into the serial inputs of the HOTLink Receiver, the time to relock the PLL becomes much harder to predict. If the noise that is being injected causes the PLL to track within its frequency offset limits (approximately ±0.25% of the REFCLK frequency) the PLL will reacquire in a few bytes (typically less than ten) after a good data stream reappears. If the PLL frequency has been moved to its offset limits by the input noise, it may take more than 60–70 bytes before the PLL locks to the good data. When the PLL hits the frequency offset limit, it will recenter itself at the REFCLK frequency and then attempt to lock to the data. While the PLL is out of lock (after experiencing a data stream interruption) the frequency of CKR will not wander beyond the offset limits.

For more information, refer to the Receiver Data-Phase Acquisition Time section of the "HOTLink Jitter Characteristics" application note.

7. If the connection between HOTLink Transmitter and Receiver is broken, what will come out of the receiver?

The exact behavior of HOTLink Receiver is difficult to predict when the serial data link is broken, since there are so many ways that the link itself can behave. The following behaviors are most common;

Bypass Mode–Reframe–OFF (RF = LOW) Clean link break with no extraneous noise input into serial inputs:

- CKR runs at REFCLK frequency.
- $\overline{RDY}$ is always HIGH.
- Qa–j all go LOW or HIGH depending on exact offsets built into transmission line termination. If the terminations are exactly matched, then Qa–j may be indeterminate.

Bypass Mode–Reframe–OFF Noise injection into serial inputs:

- CKR runs at REFCLK frequency ±<1.0% (typically <±0.25%) and may wander between its range limits and the center frequency, randomly controlled by the injected noise.
- $\overline{RDY}$ may rest HIGH or may pulse randomly as false K28.5s are decoded from the noise.
- Qa–j will be indeterminate and may switch randomly.

Encoded Mode–Reframe–OFF Clean break with no extraneous noise input into serial inputs:

- CKR runs at REFCLK frequency.
- $\overline{RDY}$ pulses once per byte.
- Q_{0-7} indicate C0.7, SC/$\overline{D}$ is always HIGH, RVS is always HIGH if there are any offsets built into transmission line termination. If the terminations are exactly matched, then Q_{0-7}, SC/$\overline{D}$ and RVS may be indeterminate.

Encoded Mode–Reframe–OFF Noise injection into serial inputs:

- CKR runs at REFCLK frequency ±<1.0% (typically <±0.25%) and may wander between its range limits and the center frequency randomly controlled by the injected noise.
- $\overline{RDY}$ may pulse randomly or once per byte.
- Q_{0-7}, SC/$\overline{D}$ and RVS may be indeterminate and may switch randomly.

Either Mode–Reframe–ON Noise injection into serial inputs:

- CKR runs at REFCLK frequency ±<1.0% (typically <±0.25%) and may wander between its range limits and the center frequency randomly controlled by the injected noise. If RF has been HIGH for less than 2048 bytes, CKR will stretch randomly as false K28.5s are decoded from the noise. If RF has been HIGH for more than 2048 byte-times, CKR will only stretch when a multiple K28.5 string is decoded from the noise.
- $\overline{RDY}$ may pulse randomly or once per byte.
- Q_{0-7}, SC/$\overline{D}$ and RVS may be indeterminate and may switch randomly.

8. What is the correct operation of the RF input on the receiver? What is the minimum number of K28.5 characters required to insure proper framing? How can I tell if the receiver is framed properly?

Recovery of information from a serial data stream requires recovery of the bit clock (accomplished by the receiver PLL) and byte synchronization (accomplished by the receiver framer). The HOTLink framer is enabled or disabled by the RF input. In well behaved, standardized point-to-point protocols that are seldom switched, the control of the byte framer is managed as a service in the protocol controller. This service monitors when some error criteria have been exceeded, and goes to a framing subroutine. This framer service sets RF=HIGH while framing and LOW during normal message transactions.

In less well behaved systems, or systems that switch data sources often, it may be necessary to leave RF=HIGH for long periods (or permanently). Leaving RF HIGH opens the system to the problem of data corruption in the serial link caused by data patterns that happen to match the SYNC character. Since this Alias SYNC is unlikely to be aligned to the normal byte boundaries, it will cause the framer to align the parallel data to the wrong byte boundary resulting in long running data corruption. When RF is set HIGH, the receiver searches the received data stream for the bit pattern matching K28.5 (001111 1010 or 110000 0101). When it is found, the internal bit counter that controls byte translation is reset and the byte boundaries are aligned to the SYNC character.

HOTLink minimizes the alias SYNC problem by incorporating a multi-byte framer into the receiver. If RF has been HIGH for less than 2048 bytes, as would be typical in protocol driven framing control, a single K28.5 will align the byte boundaries. If RF has been HIGH for more than 2048 bytes, as would be typical in packet switched systems, the multi-byte framer is enabled and a single K28.5 is no longer sufficient to align the byte boundaries. To minimize the risk of alias SYNC, reframing is only allowed when two K28.5s are detected. These two K28.5s can be adjacent, or separated by exactly one, two, or three transmission characters. Any other spacing (i.e., non-integral character separation, or too far between K28.5) is assumed to be caused by transmission errors and will be ignored for framing purposes.

In addition to the upper level protocol error detection mechanisms common in communication links, the HOTLink Receiver offers several indications that a link is misframed. For example, in Bypass mode the $\overline{RDY}$ output pulses once per K28.5 detected. If RF is LOW, the only K28.5 that can be detected is one that is properly framed, and all others will just pass through as part of the received data. If the protocol in use has a maximum packet size or a minimum number of K28.5s, a simple retriggerable-one-shot can be used to detect when framing has been lost. In this example, if the one-shot is retriggered by the properly spaced K28.5s, then the data is properly framed. If the one-shot times-out, indicating that too much time had elapsed between SYNC characters, the data would automatically be reframed by raising RF till the next K28.5 indication.

Another example of HOTLink's indication of a misframed link occurs during Encoded mode. In Encoded mode, the RVS output serves a similar if not quite as obvious function. Normal data being sent over typical data links will have a very low error rate (e.g., bit-error-rates of $1x10^{-12}$ are quite common. BER=$1x10^{-12}$ $\approx$ one error per hour at 266 MHz). Therefore, if RVS is asserted often it can be assumed that the cause is misframing. Another retriggerable-one-shot could be used to detect this condition, or it could be detected by a simple synchronous state machine constructed in a PLD.

For more information, refer to the "HOTLink CY7B933 $\overline{RDY}$ Pin Description" application note.

9. What happens to the receiver's clock and parallel outputs when it reframes?

When a byte boundary realignment occurs, the external timing of the HOTLink Receiver changes to match the new byte alignment. Logic internal to the receiver guarantees that the clock outputs (CKR and $\overline{RDY}$) never glitch. They will stretch to the new byte alignment by adding to the HIGH or LOW time of the output pulse. The exact width of the high or low times of these clock outputs will depend on the exact timing of the realignment, but neither will ever be less than that of a nominal, normally running output (i.e., five bit times, each, minimum).

The data outputs (Q_{0-7}, SC/$\overline{D}$, and RVS) all change at a time determined by internal bit-rate counters, and are timed to assure maximum set-up and hold times to down-stream logic. Since realignment will reset the cycle of the internal counter, it is possible that the outputs will change, and then change again between clock edges when byte realignment happens. Since the clock-cycle stretches, this glitch on the data output remains outside the specified data-access and hold times.

For more information, refer to the "HOTLink CY7B933 $\overline{RDY}$ Pin Description" application note.

10. What does BIST do? How can I add BIST to my system without redoing all calculations for my critical interface timing? What functionality does the BIST test and guarantee?

The HOTLink built-in self-test allows a clear and unambiguous check of the HOTLink Transmitter and Receiver, and the serial link connecting them. As part of an offline diagnostic, this feature allows the user to insure that the interconnect link is fully operational and that any other diagnostic failure indications are caused by system blocks above the physical layer. BIST allows the HOTLink adapter card manufacturer to do a quick link quality test (or node quality test with the use of the loop-back functionality of HOTLink) without the necessity of bringing up a fully functional system to do link testing.

BIST is controlled by unused HOTLink data-enable inputs. Only a few connections and minimal external logic are necessary to add BIST to an otherwise complete system. (See the Cypress application note "HOTLink Built-In Self-Test.") BIST status indications appear on the $\overline{RP}$, RVS(Qj) and $\overline{RDY}$ outputs which are easily monitored by logic internal or external to the data flow controller.

In BIST mode, the HOTLink Transmitter generates a 2^9-1 (511 byte) pseudo-random pattern using its Input register configured as a Linear Feedback Shift register. The HOTLink Receiver compares the serial BIST data stream with identical BIST patterns generated in its Output register. All of the logic in the transmitter (except the input pins) and all of the logic in the receiver (including the output pins and their attached loads) are checked by BIST. All of the serial link interconnect components are exercised with normal data patterns, which are checked byte-by-byte in real time.

11. What fiber-optic components are compatible with HOTLink products?

All standard fiber-optic interface components are compatible with HOTLink products. The following table is a representative but not comprehensive list of optical interface manufacturers. A more complete list of vendors and products is included in the "HOTLink Design Considerations" application note.

AMP/Lytel Division
61 Chubb Way
P.O. Box 1300
Somerville, NJ 08876
(908) 685-2000

CTS Corp
1201 Cumberland Ave
West Lafayette, IN 47906–1388
(317) 463-2565

Hewlett-Packard
Components Division
370 West Trimble Road
San Jose, CA 95131
(800) 535-7449 or (408) 435-6342

Siemens Fiber Optic Components
20F Commerce Way
Totowa, NJ 07512
(201) 890-1606

Sumitomo Electric
Fiber Optics Corporation
777 Old Sawmill River Road
Tarrytown, NY 10591-6725
(914) 347-3770

12. What is the significance of the HOTLink claim of "no external PLL components"?

HOTLink Transmitter and Receiver have completely integrated the PLL clock multiplier and data separator functions. These functions are implemented with high-performance phase-locked loops (PLLs) that have been tuned for maximum performance and minimum system noise sensitivity. In competitive products that purport to offer similar functions, these PLLs are often implemented with external filter and frequency setting components with the goal of achieving maximum performance. But these very same external components are the largest cause of end-user complaints and random system failures because they expose the most critical analog signals in the circuit to the external noises that abound in normal systems. External components require critical, costly and time consuming printed circuit board layout as well as high-speed analog and digital design techniques that are unfamiliar to many system integrators. HOTLink products are designed and built using fully differential analog and digital circuits to give the lowest possible output jitter and highest possible jitter tolerance. There are no external components to compromise system performance in unexpected and unpredictable ways. For more information, refer to the HOTLink Transmitter Jitter section of the "HOTLink Jitter Characteristics" application note.

13. What is the intrinsic bit-error-rate of HOTLink Transmitter and Receiver?

HOTLink BER=Zero. HOTLink Transmitter and Receiver have no intrinsic failure modes. If their power is maintained and if the interface to the link connecting them has reasonable design margin, the total error rate will be exactly that of the interconnect media. Link error rates of $<<1x10^{-15}$ are common and easily achieved. Even with worst-case design derating and end-of-life derating, BER $<<1x10^{-12}$ presents no significant challenge.

The real question being asked is, "What will be my link BER when using HOTLink?" The answer to this question involves the design of the serial transmission link and the margins designed into it. HOTLink will not significantly degrade the BER of the link. For more information, refer to the "Understanding Bit-Error-Rate with HOTLink" application note.

14. How much jitter is created by the transmitter? How much jitter is created by the receiver? What is the significance of the HOTLink Transmitter requirement for a crystal-stable clock source?

The phase-locked loops (PLLs) in the HOTLink Transmitter and Receiver act like low-pass filters to jitter that is embedded in the data or clock signal source. For the transmitter, the signal source is the CKW input. Any jitter that appears at CKW will be passed unattenuated if it has frequency components below the natural frequency of the PLL filter (approximately 500 kHz). Frequency components above the natural frequency will be attenuated at about 6 dB/octave. Frequency components that fall very near the natural frequency of the filter will be slightly amplified (approximately 0.5 dB). These are the normal characteristics of a Type-2, second-order PLL filter. When the transmitter is fed by a low jitter clock source, typical output jitter will be less than 20 ps RMS and 200 ps peak-to-peak. It is possible to measure significantly more jitter than that which is actually present if the complete system is not well understood. A few hundred millivolts of V_{CC} noise, while insignificant to the logic of a normal system board, will add imaginary jitter to the measured output. This imaginary jitter appears because a single ended oscilloscope sees the waveform as if it were measured against a fixed threshold, while the differential serial interface sees V_{CC} noise as a common mode signal to be ignored (e.g., 100 mV of V_{CC} noise could create 100−200 ps of imaginary jitter). Likewise, the normal method of measuring peak-to-peak jitter, an infinite persistence scope trace, will show larger jitter than that contributed by the HOTLink Transmitter. Low frequency jitter (wander) in the oscillator, scope trigger, temperature, and voltage related delay variations will all contribute to the width of the stored scope trace. Delay variations include TTL threshold variations that cause apparent delay variation (e.g., 100 mV of TTL threshold change can cause 100−200 ps of apparent jitter).

The signal source for the receiver is the serial data stream and, like the transmitter, it passes the frequency components of received jitter that fall below the natural frequency of its filter (approximately 300 kHz to 1000 kHz depending on actual data transition density being received). Frequency components above the natural frequency will be attenuated and there is minor jitter peaking at about the natural frequency of the PLL. Since the characteristics of the input jitter will determine the jitter content on the receiver CKR output (the only place to directly measure Rx-PLL jitter) it is somewhat difficult to predict the output jitter. Maximum CKR output jitter is less than 200 ps (peak-to-peak) when the receiver is tracking normal data (BIST data is typical) that exhibits maximum tolerable peak-to-peak jitter. Jitter from normal data is wide-bandwidth, has a significantly high-frequency content, and can have peak-to-peak amplitude of up to about 90% of a bit time. If the serial data contains a significant low frequency jitter component (typical of crystal oscillators and some pulse generators) the output jitter measured on the CKR pin could be much higher. Jitter measurements at the receiver output can be more misleading than those associated with the transmitter serial outputs, since all measurements are made on TTL outputs.

The jitter characteristics mentioned above affect system performance in the following ways. Any low-frequency jitter (below the bandwidth of either transmitter or receiver PLL) will be treated as wander.

For purposes of the PLLs, wander (usually caused by low frequency power supply variations or temperature fluctuations within the timing ICs) will not reduce the system timing margins and will not contribute to bit-error-rate. Wander can affect system timing at interfaces where the transmitter clock source is used to clock information received from a receiver tracking data from another clock source. The variation in clock frequencies may violate set-up and hold times, the exact problems usually solved by FIFO memories in typical communication systems.

High-frequency jitter (at or above the natural frequency of the PLL filters) may contribute to BER. High-frequency jitter can be caused by the clock source, media transfer characteristics, or external noise. The recovered internal bit-rate clock will not track high-frequency jitter above the PLL natural frequency. High-frequency jitter, therefore, may cause a bit edge to move into the receiver sampling window causing the bit to be erroneously sampled (a bit error).

A suitable clock source should be selected with the above effects in mind. The only clock source guaranteed to offer the required stability and high-frequency specifications is a crystal oscillator. High-frequency jitter is minimal, and low-frequency wander is usually small and very low frequency. Frequency accuracy is easily guaranteed by mechanical means, and high accuracy devices are relatively low cost. Free-running resistor-capacitor (RC) oscillators, logic gate ring oscillators or inductor-capacitor (LC) oscillators include too much high-frequency jitter, experience wide frequency variation as a function of process and environmental conditions and thus are unsuitable for this application. See the "HOTLink Jitter Characteristics" application note for more information.

15. Can I use HOTLink for anything other than Fibre Channel/ESCON™ interconnect?

HOTLink has been designed to implement the required performance and specifications of Fibre Channel and ESCON, but has additional user features that encourage use beyond these specifications. The specific timing of the parallel I/O and clock signals allow efficient interconnect with typical generic controllers and FIFO memories. The built-in self-test and the included 8B/10B encoder functions allow users to implement custom protocols that are suitable to any data-movement application. HOTLink is compatible with all common link interconnect media and interfaces. It is a low-cost, low-power, high-performance tool that enables otherwise impractical system innovation. If there is data to move, HOTLink can carry it.

16. Is HOTLink compatible with ATM?

HOTLink is compatible with the 194.40 Mbaud (155.52 MBit/second), 8B/10B interface defined by the ATM Forum. It offers all of the data, special characters and framing behaviors described in the ATM Forum User-Network Interface (UNI) Specification. In particular HOTLink serves as the physical layer interface for the physical layer for 155 Mbps Interface (and its copper variant). When operating in this capacity, HOTLink runs at 194.40 Mbaud and uses the built-in 8B/10B encoder. All required data and special codes and responses are included in HOTLink.

17. Is HOTLink compatible with SONET?

HOTLink is not directly compatible with SONET for at least the following reasons:

- There are no standard SONET frequencies within its operating range of 160–330 Mbaud.
- HOTLink has a 10-bit unencoded interface, and SONET systems use an 8-bit interface.
- SONET requires a much slower rate-of-change of frequency during loss of signal than HOTLink can achieve.

The HOTLink Receiver can tolerate the long strings of zeros contained in SONET serial streams, and future designs will directly accommodate SONET specifications.

18. What is the latency through a HOTLink Transmitter and Receiver?

The input data is stored in the Transmitter Input register on the rising edge of CKW, so this becomes time-zero. Approximately 21 bit-times (i.e., 21 times the period of CKW ÷ 10) minus the t_{PD} of a TTL output buffer (approximately 10 ns) later, the first bit of that data will emerge from the OUTA±, B±, and C± pins. After the transit time of the serial link, which can be significant, that bit will appear at the receiver. Transit times for typical serial links include the propagation delay of the optical modules (typically 5–10 ns for the pair), if any, and the propagation rate in the link media (i.e., approximately 1 ns/ft in copper, and 2 ns/ft in multi-mode optical cable). Approximately 24 bit-times plus the t_{PD} of a TTL output buffer (approximately 10 ns) after the first data bit is received at the input of the receiver, it appears at the Q_{0-7} outputs. Eight bit-times later CKR rises and the data transfer is complete. The total latency of a HOTLink Tx/Rx pair is approximately link delay plus 45 bit-times.

19. Is there a VERILOG or VHDL model of HOTLink?

Logic Modeling offers full function logic models of both the HOTLink Transmitter (CY7B923) and the HOTLink Receiver (CY7B933). These models perform all of the normal chip functions including BIST, Encoded, and Bypass modes of operation. The models accurately model the "real" parts and have been validated by having them run the actual-chip design-simulation vectors and the outgoing-test vectors. Logic Modeling offers a wide variety of standard product logic models that run on various simulations platforms. They can be reached at:

Logic Modeling
19500 N.W. Gibbs Drive
P.O Box 310
Beaverton, OR 97006
Telephone (503) 690–6900
Fax (503) 690–6906

20. I need to estimate the reliability of HOTLink in my design. How many components does it contain?

Table 3. HOTLink Reliability Data

	CY7B923	CY7B933
Number of components	4285	7988
Number of transistors	3813	6855
Number of gates	2072	2960
Percent digital by gate count	85	90
Percent analog by die area	30	20
Die size	96 x 116 mils	126 x 131 mils

Built on Cypress Standard 0.8-micron BiCMOS. Designed for reliable operation at temperatures $-55°C < T_j < 155°C$. All pins characterized to withstand ESD >4400V (HBM). Wafer Fab Capability in San Jose, CA; Round Rock, TX.

HOTLink™ Design Considerations

Application Note Overview

The HOTLink™ family of data communications products provides a simple and low-cost solution to high-speed data transmission. While these products are easy to use, the methods used to connect them to high-speed serial interfaces are often not intuitive. This document provides a basic level of explanation of the parallel and serial interface characteristics, and provides some cookbook solutions for interfacing them to different types of parts and media.

Primary Topics

The primary topics covered in this application note are

- HOTLink Overview
- HOTLink Serial Signal Characteristics
- Terminating HOTLink Serial Signals
- Interfacing to HOTLink
- Serial Link Support Components

HOTLink Overview

HOTLink Features

- Fibre Channel compliant
- IBM® ESCON™ compliant
- ATM Compatible
- 8B/10B-coded or 10-bit unencoded
- 160- to 330-Mbps data rate
- TTL synchronous I/O
- No external PLL components
- Triple ECL 100K serial outputs
- Dual ECL 100K serial inputs
- Low power: 350 mW (Tx), 650 mW (Rx)
- Compatible with fiber-optic modules, coaxial cable, and twisted-pair media
- Built-In Self-Test
- Single +5V supply
- 28-pin SOIC/PLCC/LCC
- 0.8μ BiCMOS

Functional Description

The CY7B923 HOTLink Transmitter and CY7B933 HOTLink Receiver are point-to-point communications building blocks that transfer data over high-speed serial links (fiber-optic, coax, and twisted/parallel-pair) at 160- to 330-Mbits/second. *Figure 1* illustrates typical connections to host systems or controllers.

Eight bits of user data or protocol information are loaded into the HOTLink Transmitter and are encoded. Serial data is shifted out of the three differential positive ECL (PECL) serial ports at the bit-rate (which is ten times the byte-rate).

The HOTLink Receiver accepts the serial bit stream at its differential line receiver inputs, and using a completely integrated phase-locked-loop (PLL) clock synchronizer recovers the timing information necessary for data reconstruction. The bit stream is deserialized, decoded, and checked for transmission errors. The recovered byte is presented in parallel to the receiving host along with the synchronized byte-rate clock.

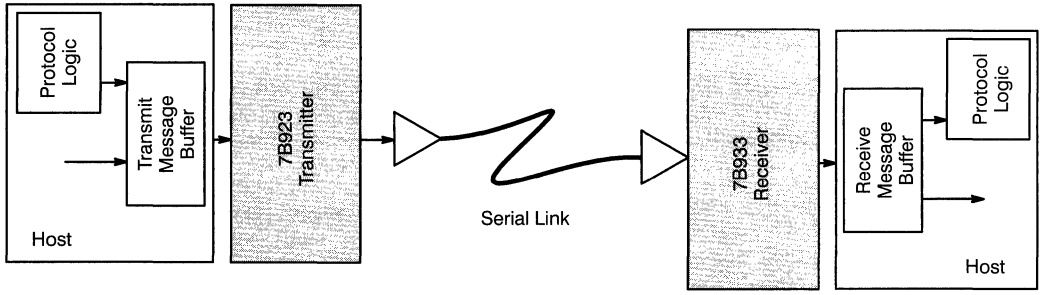


Figure 1. HOTLink System Connections

The 8B/10B encoder/decoder (Reference 1, 2) can be disabled in systems that already encode or scramble the transmitted data. Signals are available to create a seamless interface with both asynchronous FIFOs (i.e., Cypress's CY7C42X) and clocked FIFOs (i.e., Cypress's CY7C44X). A built-in self-test pattern generator and checker allows testing of the transmitter, receiver, and the connecting link as a part of a system diagnostic check.

HOTLink devices are ideal for a variety of applications where a parallel interface can be replaced with a high-speed point-to-point serial link. Applications include interconnecting workstations, servers, mass storage, and video transmission equipment.

CY7B923 HOTLink Transmitter Description

The function of the HOTLink Transmitter is to convert byte-rate parallel data into a high speed serial data stream. A logic block diagram of the transmitter is shown in *Figure 2.*

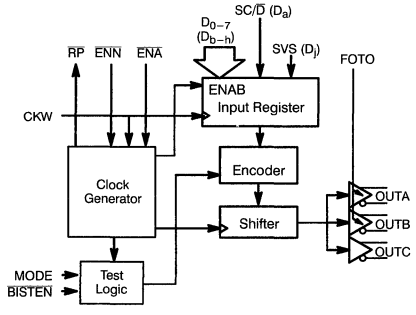


Figure 2. CY7B923 Transmitter Logic Block Diagram

Input Register

The Input register holds the data to be processed by the HOTLink Transmitter and allows the input timing to be made consistent with standard FIFOs. The Input register is clocked by CKW (clock write) and loaded with information on the D_{0-7}, SC/$\overline{D}$ (special character/data select), and SVS (send violation symbol) pins. Two enable inputs ($\overline{ENA}$ and $\overline{ENN}$) allow the user to choose when data is to be sent. Asserting $\overline{ENA}$ (enable, active LOW) causes the inputs to be loaded on the rising edge of CKW. If $\overline{ENN}$ (enable next, active LOW) is asserted when CKW rises, the data present on the inputs on the *next* rising edge of CKW will be loaded into the input register. These two inputs allow proper timing and function for compatibility with either asynchronous FIFOs or clocked FIFOs without external logic.

In BIST mode, the Input register becomes the signature pattern generator by logically converting the parallel input register into a linear feedback shift register (LFSR). When enabled, this LFSR generates a 511-byte sequence that includes all Data and Special Character codes, including the explicit violation symbols. This pattern provides a predictable but pseudo-random sequence that can be matched to an identical LFSR in the HOTLink Receiver. For additional information see the Cypress Semiconductor application note "HOTLink Built-In Self-Test."

Encoder

The Encoder transforms the input data held by the Input register into a form more suitable for transmission on a serial interface link. The code used is

specified by ANSI X3T11 Fibre Channel (Reference 3) and the IBM ESCON channel (Reference 4) (code tables are available in the CY7B923/CY7B933 datasheet). The eight D_{0-7} data inputs are converted to either a Data symbol or a Special Character, depending upon the state of the SC/$\overline{D}$ input. If SC/$\overline{D}$ is HIGH, the data inputs represent a control code and are encoded using the Special Character code tables. If SC/$\overline{D}$ is LOW, the data inputs are converted using the Data code table. If a byte-time passes with the inputs disabled, the Encoder will output a Special Character Comma (K28.5 or SYNC) to maintain link synchronization. The SVS input forces the transmission of a specified Violation symbol to allow the user to check error handling logic in the system controller.

The 8B/10B coding function of the Encoder can be bypassed for systems that include an external coder or scrambler function as part of the controller. This bypass capability is controlled by setting the MODE select pin HIGH. When in bypass mode, D_{a-j} (note that bit order is specified by the Fibre Channel 8B/10B code) become the ten inputs to the Shifter, with D_a being the first bit to be shifted out.

Shifter

The Shifter accepts parallel data from the Encoder once each byte-time and shifts it to the serial interface output buffers using a PLL multiplied bit-clock that runs at 10 times the byte-clock (CKW) rate. Timing for the parallel transfer is controlled by the counter included in the Clock Generator, and is not affected by signal levels or timing at the input pins.

OutA, OutB, OutC

The serial interface ECL output buffers (100K signal levels referenced to +5V) are the drivers for the serial media. They are all connected to the Shifter and contain the same serial data. Two of the output pairs (OUTA± and OUTB±) are controlled by the FOTO input and can be disabled by the system controller to force a logical zero (i.e., "light off") at the outputs. The third output pair (OUTC±) is not affected by FOTO and will supply a continuous data stream suitable for loop-back testing of the subsystem.

OUTA± and OUTB± will respond to FOTO input changes within a few bit times. However, since FOTO is not synchronized with the transmitter data stream, the outputs will be forced off or turned on at arbitrary points in a transmitted byte. This function is intended to augment an external laser safety controller and as an aid for Receiver PLL testing.

In wire-based systems, control of the outputs may not be required, and FOTO can be strapped LOW. The three output pairs are intended to add system and architectural flexibility by offering identical serial bit streams with separate interfaces for redundant connections or for multiple destinations. Unneeded outputs can be left open or wired to V_{CC} to disable and power down the unused output circuitry.

Clock Generator

The clock generator is an embedded phase-locked loop (PLL) that takes a byte-rate reference clock (CKW) and multiplies it by ten to create a bit-rate clock for driving the serial shifter. The byte-rate reference comes from CKW, the rising edge of which clocks data into the Input register. This clock must be a crystal-referenced pulse stream that has a frequency between the minimum and maximum specified for the HOTLink Transmitter/Receiver pair. Signals controlled by this block form the bit-clock and the timing signals that control internal data transfers between the Input register and the Shifter.

The read pulse ($\overline{RP}$) is derived from the feedback counter used in the PLL multiplier. It is a byte-rate pulse stream with the proper phase and pulse widths to allow transfer of data from an asynchronous FIFO. Pulse width is independent of CKW duty cycle, since proper phase and duty cycle is maintained by the PLL. The $\overline{RP}$ pulse stream will insure correct data transfers between asynchronous FIFOs and the transmitter input latch with no external logic.

Test Logic

Test logic includes the initialization and control for the built-in self-test (BIST) generator, the multiplexer for Test mode clock distribution, and control logic to properly select the data encoding. Test logic is discussed in more detail in the CY7B923/CY7B933 HOTLink datasheet.

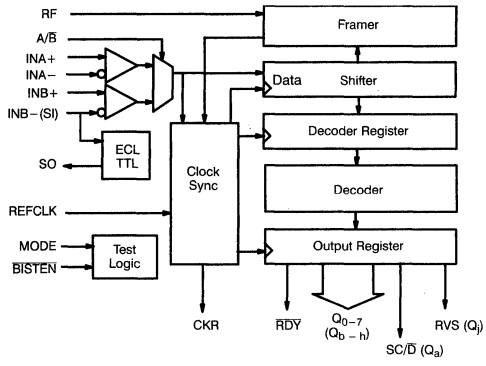


Figure 3. CY7B933 Receiver Logic Block Diagram

CY7B933 HOTLink Receiver Description

The function of the HOTLink Receiver is to convert a high-speed serial data stream into byte-rate parallel data. A logic block diagram of the receiver is shown in *Figure 3*.

Serial Data Inputs

The HOTLink Receiver has two differential line receivers (INA± and INB±) that can be selected as inputs for the serial data stream. INA± or INB± is selected with the A/$\overline{B}$ input. INA± is selected when A/$\overline{B}$ is HIGH and INB± is selected when A/$\overline{B}$ is LOW. The threshold of A/$\overline{B}$ is compatible with ECL 100K signals. TTL logic elements can be used to select the INA± or INB± inputs by adding a resistor voltage divider to a TTL driver connected to A/$\overline{B}$ (see *Figure 35*). The differential sensitivity of INA± and INB± will accommodate wire interconnect with filtering losses or transmission line attenuation greater than 20 dB ($V_{DIF} \geqq 50$ mV). These inputs can alternatively be directly connected to fiber-optic interface modules (any ECL logic family, not limited to ECL 100K) with up to 1.2V of differential signal. The common-mode tolerance accommodates a wide range of signal termination voltages. The highest HIGH input that can be tolerated is $V_{IN} = V_{CC}$, and the lowest LOW input that can be interpreted correctly is V_{IN} = GND+2.0V.

ECL-TTL Translator

The function of the INB(INB+) input and the SI(INB−) input is determined by the connection on the SO output pin. If the ECL/TTL translator function is not required, the SO output is wired to V_{CC}. A sensor circuit detects this connection and causes the inputs to become INB± (a differential line-receiver for serial-data input). If the ECL/TTL translator function is required, the SO output is connected to a normal TTL load (typically one or more TTL inputs, but no pull-up resistor) and the inputs become INB (single-ended ECL 100K-level serial-data input) and SI (single-ended ECL 100K-level status input).

This positive-referenced ECL-to-TTL translator is provided to eliminate external logic between an ECL carrier-detect or link status signal and a TTL input in the control logic. The input threshold is compatible with ECL 100K-levels (+5V referenced).

Clock Sync

The Clock Synchronizer function is performed by an embedded phase-locked loop (PLL) that tracks the frequency of the incoming serial bit-stream and aligns the phase of its internal bit-rate clock to the serial data transitions. This block contains the logic to transfer the data from the Shifter to the Decode register once every byte. The counter that controls this transfer is initialized by the Framer logic. CKR is a buffered output derived from the bit counter used to control Decode register and Output register transfers.

Clock output logic is designed such that when reframing causes the counter sequence to be interrupted, the period and pulsewidth of CKR will never be less than normal. Reframing may stretch the period of CKR by up to 90%, and either CKR pulsewidth HIGH or pulsewidth LOW may be stretched, depending on when reframe occurs.

The REFCLK input provides a byte-rate reference frequency to improve PLL acquisition time and limit unlocked frequency excursions of CKR when no data is present at the serial inputs. The frequency of REFCLK is required to be within ±0.1% of the frequency of the clock that drives the transmitter CKW pin.

Framer

Framer logic checks the incoming bit stream for the pattern that determines the byte boundaries. This combinatorial logic filter looks for the ANSI Fibre Channel symbol defined as a Special Character Comma (K28.5) (Reference 3). When it is found, the free-running bit-counter in the Clock Sync block is synchronously reset to its initial state, thus framing the data on the correct byte boundaries.

Random errors that occur in the serial data can corrupt some data patterns into a bit pattern identical to a K28.5, and thus cause an erroneous data-framing error. The RF input prevents this by inhibiting reframing during times when normal message data is present. When RF is held LOW, the HOTLink Receiver deserializes the incoming data without trying to reframe the data to incoming patterns. When RF rises, $\overline{RDY}$ is inhibited until a K28.5 has been detected, after which $\overline{RDY}$ resumes its normal function. While RF is HIGH, it is possible that an error could cause misframing, after which all data will be corrupted. Likewise, a K28.7 followed by D11.x, D20.x, or an SVS (C0.7) followed by D11.x will cause erroneous framing. These sequences must be avoided while RF is HIGH.

If RF remains HIGH for greater than 2048 bytes, the framer switches to double-byte framing, requiring two K28.5 Special Characters within five bytes.

Shifter

The Shifter accepts serial data from one of the Serial Data input pairs one bit at a time, as clocked by the Clock Sync logic. Data is examined by the Framer on each bit, and is transferred to the Decode register once per byte.

Decode Register

The Decode register accepts data from the Shifter once per byte as determined by the logic in the Clock Sync block. It is presented to the Decoder and held until it is transferred to the output latch.

Decoder

Parallel data is transformed from ANSI Fibre Channel 8B/10B codes (Reference 3) back to "raw data" in the Decoder. This block uses the standard decoder patterns found in the Valid Data Characters and Valid Special Character Codes and Sequences (code tables are available in the CY7B923/CY7B933 datasheet). Data patterns are signaled by a LOW on the SC/$\overline{D}$ output and Special Character patterns are signaled by a HIGH on the SC/$\overline{D}$ output. Unused patterns or disparity errors are signaled as errors by a HIGH on the RVS (Received Violation Symbol) output and by specific Special Character codes.

Output Register

The Output register holds the recovered data (Q_{0-7}, SC/$\overline{D}$, and RVS) and aligns it with the recovered byte clock (CKR). This synchronization insures proper timing to match a FIFO interface or other logic that requires glitch free and specified output behavior. Outputs change synchronously with the rising edge of CKR.

In BIST mode, this register becomes the signature pattern generator and checker by logically converting itself into a Linear-Feedback Shift-Register (LFSR) pattern generator. When enabled, this LFSR generates a 511-byte sequence that includes all Data and Special Character codes, including the explicit violation symbols. This pattern provides a predictable but pseudo-random sequence that can be matched to an identical LFSR in the transmitter. When synchronized, it checks each byte in the Decoder with each byte generated by the LFSR and indicates errors using RVS. Patterns generated by the LFSR are compared after being buffered to the output pins and then fed back to the comparators, allowing test of the entire receive function.

In BIST mode, the LFSR is initialized by the first occurrence of the transmitter BIST loop start code D0.0 (D0.0 is sent only once per BIST loop). Once the BIST loop has been started, RVS will be HIGH for pattern mismatches between the received sequence and the internally generated sequence. Code rule violations or running disparity errors that occur as part of the BIST loop do not cause an error indication. $\overline{RDY}$ pulses high once per BIST loop and can be used to check test pattern progress. The receiver BIST checker can be reinitialized by leaving and re-entering BIST mode.

Test Logic

Test logic includes the initialization and control for the built-in self-test (BIST) checker, the multiplexer for Test mode clock distribution, and control logic for the decoder. Test logic is discussed in more detail in the CY7B923/CY7B933 HOTLink datasheet.

HOTLink Serial Signal Characteristics

The serial interfaces on the HOTLink Transmitter and Receiver are based on the standard for high-speed digital logic called emitter-coupled-logic or ECL. This form of logic has been used commercially in integrated circuits since the early 1960s, and prior to that it was implemented in discrete form.

ECL is a non-saturating form of digital logic. ECL gets its name from how the emitters of a differential amplifier in the circuit are connected. The main features of this logic family are very high speed, low noise, and the ability to drive low-impedance transmission lines.

In the past, many engineers have avoided ECL as a logic family because it was different from the TLL-compatible families with which they were more familiar. Proper use of ECL requires the understanding and application of transmission lines, line termination, and power supply bypassing. Because of the faster speeds present in the newer TTL compatible families, these same disciplines are now required for TTL circuits as well.

ECL Signal Level Reference

The primary differences between ECL and other logic families are the signal levels used to represent the HIGH and LOW logic levels.

In the TTL and CMOS logic families, a LOW is usually some level close to V_{SS}, and a HIGH is usually some level close to V_{CC}. The ground or reference point for these measurements is usually the V_{SS} point, with V_{CC} set to +5V from that ground reference.

In standard ECL this changes significantly. Instead of having the ground reference at V_{SS}, it is placed at V_{CC}. This means that both HIGH and LOW logic levels exist at potentials that are negative with respect to ground. Standard ECL is specified as operating with a negative supply (−4.5V to −5.2V for V_{EE}). Since ground is only a reference point, it is also possible to operate ECL with a positive supply. When used in this mode ECL is usually referred to as PECL which means Positive ECL.

ECL Basic Switch

Internally, ECL gates (or switches) operate using a current source whose current is directed through one of two paths back to V_{CC}. A schematic of this basic ECL switch is shown in *Figure 4* (Reference 5).

In this ECL switch, the state of the switch is determined by the voltage drop across R1 and R2. The output signal swing is set by the size of these resistors and the magnitude of the current passed through them.

The base of Q2 is biased at a fixed voltage called V_{BB}. This voltage determines at what level of V_{IN} on Q1 that the majority of the current flowing in the switch changes from R1 to R2. If V_{IN} is set to the same voltage as V_{BB}, the current divides equally between R1 and R2. Increasing V_{IN} by 125 mV above V_{BB} causes essentially all the current to be run through Q1 (and hence R1). Lowering V_{IN} to 125 mV below V_{BB} causes essentially all the current to flow through Q2. This means that an input swing of as little as 250 mV can cause the ECL gate to switch completely from a 0 to a 1. To provide noise immunity and allow operation over a wide variety of conditions, the actual signal swing specified for ECL signals is around 800 mV.

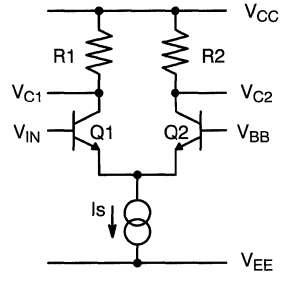


Figure 4. Basic ECL Switch

Emitter-Follower

The switch shown in *Figure 4* can react very quickly but, because of its high-value resistor pull-ups (R1 and R2), its switching delay varies directly with load capacitance. To allow larger loads to be driven, and to make the output voltages compatible with the input of subsequent gates, additional transistors are added in an emitter-follower configuration as illustrated in *Figure 5*.

These emitter-follower transistors have a very low on impedance (5–7Ω). This allows ECL gates to drive transmission lines having impedances as low as 50Ω, and can supply load currents of up to 50 mA.

The emitter-follower transistors have an uncommitted emitter as their output. This allows the transistor to source, but not sink, current. This is effectively the opposite of an open-collector output in a TTL part. To allow the output to function correctly, it requires a load that operates as a pull-down.

ECL Signal Levels

ECL signals operate over a very narrow and tightly controlled range. These signal levels are referenced from the V_{CC} pins of the parts. *Figure 6* shows the relationships of the different output and input levels for ECL gates. The names of these levels are detailed in *Table 1*.

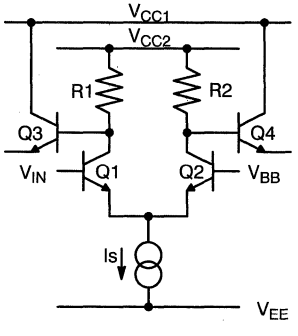


Figure 5. Buffered ECL Switch

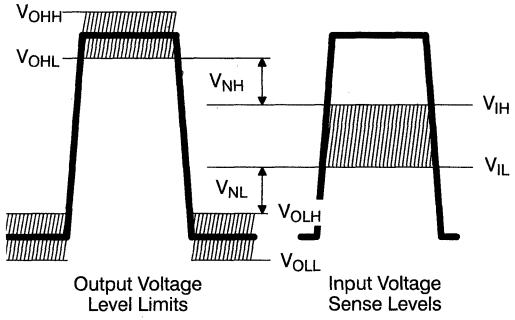


Figure 6. ECL Signal Levels

Table 1. ECL Signal Level Names

Name	Description
V_{OHH}	Highest Output HIGH Voltage
V_{OHL}	Lowest Output HIGH Voltage
V_{OLH}	Highest Output LOW Voltage
V_{OLL}	Lowest Output LOW Voltage
V_{IH}	Lowest Input HIGH Voltage Threshold
V_{IL}	Highest Input LOW Voltage Threshold
V_{NH}	High Input Noise Margin ($V_{OHL}-V_{IH}$)
V_{NL}	Low Input Noise Margin ($V_{OLH}-V_{IL}$)

ECL Output Signal Levels

ECL outputs are all referenced from V_{CC}. A typical ECL driver has an output-HIGH level (V_{OH}) of V_{CC} – 0.85V and an output-LOW level (V_{OL}) of V_{CC} – 1.7V. These typical values are seldom specified for parts because a good design must be done using the range limits for these signals as listed in *Table 1*. Actual values for these levels vary by individual part type and ECL family.

ECL Input Signal Levels

ECL Inputs are also referenced from V_{CC}. A typical ECL receiver has an input-HIGH (V_{IH}) threshold of V_{CC} – 1.1V and an input-LOW (V_{IL}) threshold of V_{CC} – 1.47V. These differences between the output and input HIGH and LOW values translate directly into the usable noise margin (V_{NH} and V_{NL}) of a system.

Viewing ECL Signals

Proper viewing of ECL signals requires use of an oscilloscope and probes with sufficient bandwidth to see the important features of the waveforms. Depending on the speed of the signals being viewed, different scope and probe characteristics are required.

Oscilloscope Bandwidth

Oscilloscope bandwidth is not a simple number; it is based on the combined bandwidths of multiple pieces of the measurement system. These can include the oscilloscope, the scope probe amplifier, the probe itself, and possibly other components.

The calculation for bandwidth is based on an inverse sum-of-squares as shown in *Equation 1*.

$$bw = \frac{1}{\sqrt{\left(\frac{1}{bw_1}\right)^2 + \left(\frac{1}{bw_2}\right)^2}} \qquad \text{Eq. 1}$$

Thus a scope with a 1-GHz bandwidth probe using a 1-GHz bandwidth amplifier would only have a usable bandwidth of 700 MHz.

The current ANSI Fibre Channel standard specifies the minimum system bandwidth for testing as 1.8 times the baud rate. For testing with the HOTLink parts (330 Mbaud), this translates to a minimum system bandwidth of 600 MHz. This is translated into a viewable rise time using *Equation 2* (Reference 6).

$$t_r = \frac{0.35}{bw} \qquad \text{Eq. 2}$$

This means that the oscilloscope and probes, having a 600 MHz bandwidth, can display signals with rise-times no faster than 600 ps, without having more than 3 dB of attenuation.

Note: Various scope manufacturers use different conventions to specify bandwidth for their equipment; i.e., specified bandwidth is not necessarily where the displayed waveforms are 3 dB down in amplitude.

Scope Probes

Scope probes are available with many different characteristics. The three main types are referred to as passive high-impedance, active high-impedance, and passive low-impedance.

Passive high-impedance probes usually range from as low as 10-kΩ to 10-MΩ load impedance. This number identifies the loading effect of the probe when attached to a circuit. The best feature of high-impedance probes is that their impedance is usually much larger that those of the circuit under test and thus do not present any appreciable DC load to the measured signal when present.

Passive high-impedance probes do suffer one major drawback: significant capacitive loading. Most high-impedance probes present from 5 pF to 20 pF of capacitance at the probe tip. This capacitance affects measurements in two ways; it slows down the circuit being measured, and it degrades the rise-time of the probe. The upper bandwidth limit for passive high-impedance probes is around 400 MHz.

Active high-impedance probes combine a high bandwidth amplifier with the probe to improve the overall bandwidth of the system. These probes usually exhibit load impedances of 10 kΩ to 10 MΩ but have load capacitances of less than 3 pF. This type of probe has a typical upper bandwidth limit of around 1 GHz.

Care should be taken when using active probes as the manufacturers specified bandwidth may not be where the signal measured is 3 dB down. To achieve the higher bandwidths some active probes have non-linear responses to equalize the probe response. When presented with edge rates or frequency components beyond the specified probe bandwidth, the probe and scope may actually display a distorted waveform having more high-frequency components present than are actually in the measured signal.

Passive low-impedance (resistive divider) probes are used for the highest frequency work. These probes are available in load impedances from 50Ω to 5 kΩ, and present load capacitances of 1 pF or less. A typical upper bandwidth limit for these probes is around 3 GHz. Unlike the high-impedance probes, low-impedance probes are designed to connect to a 50Ω transmission line system and do not require compensation. The probe itself is an extension of the 50Ω transmission line present in the scope, and

contains a precision resistive-divider at the probe tip.

The main drawback of passive low-impedance probes is the load impedance they present to the circuit. The rule of thumb for probes is that the probe impedance needs to be an order of magnitude greater than the impedances present around it to avoid any appreciable distortion. To get around this the probe is often designed as part of the system under test, such that its impedance is factored into the design. When the probe is not present it may be necessary to change component values or configurations to compensate for the absence of the probe (Reference 7).

Table 2 shows a summary of typical oscilloscope probe characteristics. For proper viewing of HOTLink ECL signals, an oscilloscope should have a minimum system bandwidth of 600 MHz. In most cases this will require use of low-impedance probes.

Table 2. Typical Probe Characteristics

Probe Type	Z	C_{load}	BW (MHz)
Passive High-Z	10 k– 10 MΩ	5–20 pF	400
Active High-Z	10 k– 10 MΩ	3 pF	1000
Passive Low-Z	50–5 kΩ	1 pF	3000

Probe Grounding

As with any measurement, a good ground is mandatory. What is often misunderstood is just what *is* a good ground. At the frequencies used with HOTLink, a long looping ground lead is about as good as no ground at all. Three factors come into play: the reflections caused by the scope probe, and the ground inductance and parasitic capacitance limiting the probe's bandwidth. A simple rule of thumb for ground leads is that they exhibit about 1 nH of inductance for each millimeter of length. As the length of the probe's ground lead increases, the probe's resonance point decreases.

To view a signal with minimal distortion, the probe's resonant frequency must remain above the highest frequency signal component of interest. The graph

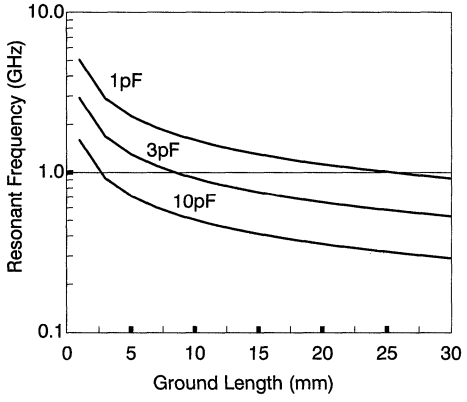


Figure 7. Scope Probe Resonant Frequency

in *Figure 7* shows how a scope probe's resonant frequency varies for different lengths of ground loop inductance and tip capacitance. This graph is based on *Equation 3* with the diagram of a low-impedance probe shown in *Figure 8*.

$$\omega = 2\pi f = \frac{1}{\sqrt{LC}} \qquad \text{Eq. 3}$$

From this graph it is quite apparent that a ground lead of only 10 mm cuts the resonant frequency of the probe by 75%. For signal viewing at HOTLink serial data rates it is usually necessary to use coaxial scope-tip sockets soldered directly to a circuit board, or some other probe type that probes for signal and ground without a loose ground lead (Reference 8).

Probing From V_{CC}

The normal mode for probing ECL is to use V_{CC} as the ground reference. In this mode the signal being viewed is is below ground and is relatively close to the ground reference. If the overall circuit design uses TTL parts in a mix with the negative referenced ECL, the TTL signals will all exist above ground. If

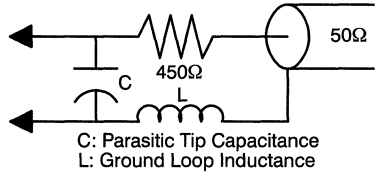


Figure 8. Scope Probe Tip Schematic

the ECL parts are operated in a PECL mode where they share a common V_{CC} supply with other TTL or CMOS parts, all probing should be done from TTL ground, which is the V_{EE} side of the ECL parts.

Probing From V_{EE}

When V_{EE} is used as the scope ground, other issues may come into play. In this mode the ECL signal is now positioned almost 4V above the reference point. While many scopes are able to perform a DC offset to make the ECL signal viewable, some do this at the expense of sensitivity. In other words, a signal that is viewable at 100 mV/div when offset less than 2V, may only be viewable at 500 mV/div when offset by 4V. Since the total signal swing for ECL signals is only 800 mV, it may be difficult to see a detailed representation of the waveform at this resolution.

Another problem with measuring from V_{EE} is that all the references in the ECL part are regulated from V_{CC}, not V_{EE}. This means that any amplitude changes or ripple in the power supply are now added into the displayed waveform.

One way around the offset problem is to AC couple the signal into the scope. Some scopes offer this as a front panel set-up selection, while others require the addition of a wide-bandwidth DC-blocking capacitor in line with the scope probe. Either of these will remove all DC components from the signal under test, and allow the signal to be displayed at the maximum resolution of the oscilloscope.

Wide-bandwidth capacitors designed for this function are available from most test equipment manufacturers for use with existing probes and scope amplifiers. Some common capacitor types for SMA connector probes are the Tektronix 015-1-13-00 and Hewlett-Packard 11742A. For BNC connectored probes the Hewlett-Packard 10240B is also available.

Sample ECL Waveforms

ECL signals, when properly biased, terminated, and bypassed, are very clean and stable. Any noticeable overshoot on signals is usually caused by reflections from improperly terminated transmission lines or improper probing. *Figure 9* shows what a pristine single-ended ECL waveform should resemble when viewed on a scope.

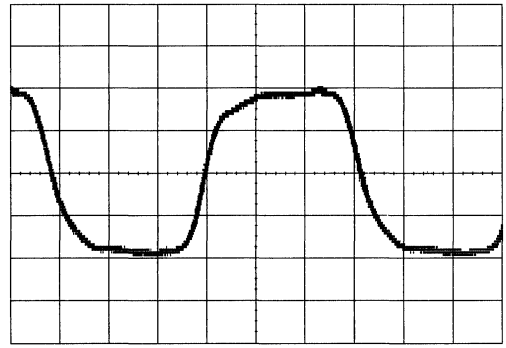


Figure 9. Good ECL Waveform, Single-Ended vs. V_{CC} Ground

Both the rising and falling edges are quite symmetrical and approximate an RC charge/discharge curve. The peak-to-peak range of the transition covers approximately 800 mV and is centered around V_{CC} − 1.3V. This signal was measured using a 500Ω , 1.5-GHz bandwidth low-impedance probe, on a scope having 1-GHz bandwidth. This signal was measured with V_{CC} as the probe ground. The probe load impedance (500Ω) was combined with other bias resistors to present a 50Ω to V_{CC} − 2V load on the signal.

With incorrect termination, a waveform such as that illustrated in *Figure 10* can result. Here the spike in the middle of a low area may cross the receiver V_{IH} threshold and cause the receiver to start to switch.

ECL Logic Families

Just as the TTL compatible world has its 7400, 74LS, 74H, 74S, 74AS, 74ALS, etc. logic families that have evolved over time, so does ECL. The most common families still in use are referred to as 10K (e.g., SL10104), 10KH (e.g., MC10H116), and 100K (e.g., F100150). These ECL families differ in terms of speed, signal levels, noise margins, and temperature and voltage stability.

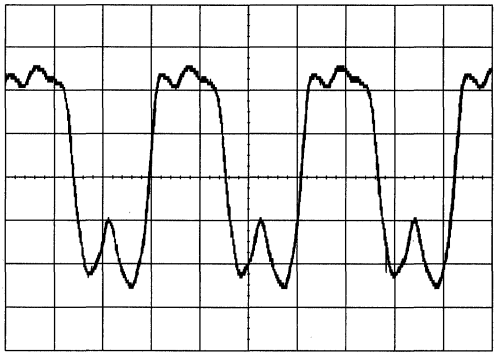


Figure 10. Bad ECL Waveform

10K ECL

The 10K ECL family has been around since 1971. It provides propagation delays of 2 ns with slow 3.5-ns edge rates (10%−90%). The voltage swings and switching thresholds of this logic family are relatively insensitive to variations in the power supply voltage but are affected by operating temperature (−30°C to +85°C). The V_{BB} bias network is fixed at V_{CC} − 1.29V, and is compensated for voltage and temperature. In the basic 10K ECL switch the current source is unregulated and consists of a single resistor between V_{EE} and the tied emitters of the differential amplifier. The transfer curves of a simple 10K gate are illustrated in *Figure 11* and detail how this family is sensitive to temperature variations in both inputs and outputs (Reference 19).

10KH ECL

To improve system speeds, the 10KH ECL family was introduced in 1981. It reduced propagation delays to 1 ns while edge rates were set to 1.8 ns. Because the thresholds and voltage swings remain the same in 10KH as in 10K, these two ECL families are fully compatible with each other. The temperature and voltage compensated V_{BB} reference network from 10K parts was replaced with a fully compensated and regulated supply. To improve the V_{OL} levels the resistor current source was replaced with a regulated current source. This allowed the collector resistors in the ECL switch to be matched and have similar switching characteristics. The transfer curves of a simple 10KH gate (see *Figure 12*) illustrate how this family improves noise margins over 10K ECL, yet remains sensitive to temperature variations. The 10KH family also is specified to operate over a narrower temperature range (0°C to 75°C) than 10K ECL (Reference 19).

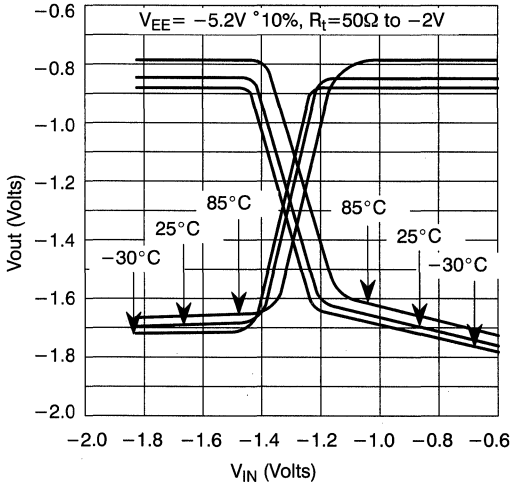


Figure 11. 10K ECL Transfer Functions

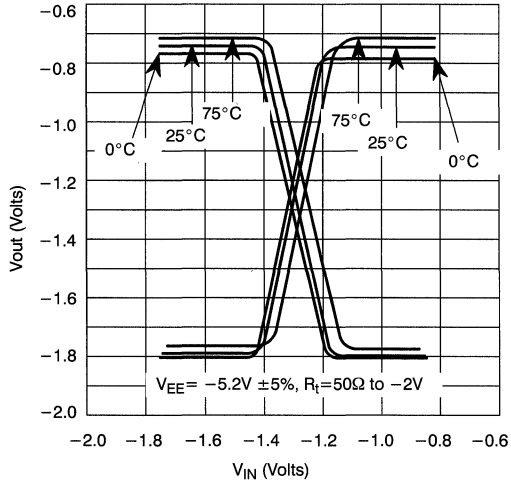


Figure 12. 10KH ECL Transfer Functions

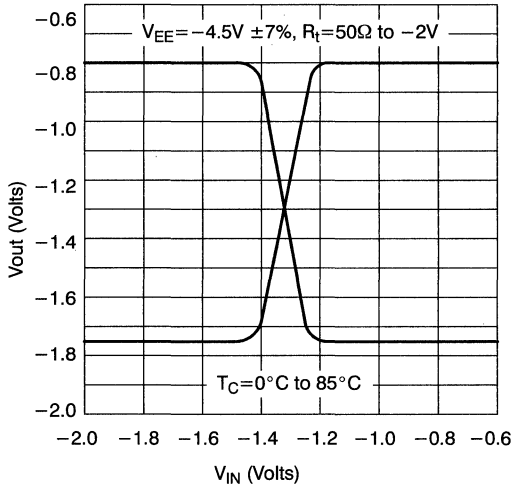


Figure 13. 100K ECL Transfer Functions

100K ECL

The 100K ECL family is a faster and easier to use ECL logic family. Introduced in 1973, this family improved on the internal structures to provide 750-ps propagation delays and 1-ns edge rates. In addition to speed improvements, the 100K ECL family was the first to introduce full compensation. This means that all the critical structures in the parts are now compensated for variations in voltage and temperature. This minimizes differences in propagation delays from one stage to the next that limit the maximum operating rate of a system. This stability is illustrated in the transfer curves in *Figure 13* (Reference 5).

In the 100K ECL family the operating temperature range is expanded to 0°C to 85°C but the nominal operating voltage changes from −5.2V to −4.5V.

HOTLink ECL Outputs

All ECL outputs of the HOTLink Transmitter are ECL 100K-level compatible. This means that these outputs meet or exceed all voltage, current, and edge rates specifications of 100K ECL and will interoperate with other 100K ECL parts. This signal level compatibility is required by the ANSI Fibre Channel standard (Reference 3).

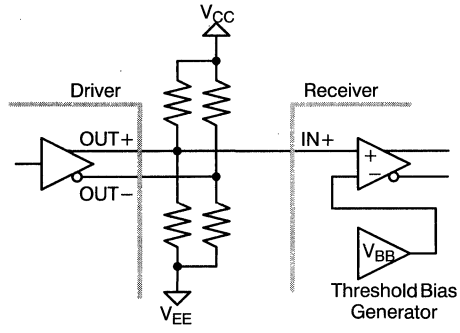


Figure 14. Single-Ended Connection

The HOTLink ECL outputs actually are substantially better than the 100K ECL specification, allowing operation with 5V±10% supplies over the full −55°C to +125°C temperature range. This allows the HOTLink parts to be used in a TTL, PECL, or ECL environment.

The HOTLink Transmitter has six ECL outputs configured as three differential pairs: OUTA±, OUTB±, and OUTC± (see *Figure 2*). These differential outputs may be used to communicate with ECL compatible receivers in either single-ended (strongly discouraged) or differential (preferred) modes.

HOTLink Transmitter Single-Ended Connections

A single-ended connection is used most often for logic functions. In this type of a connection, a single output of a driver is attached to a single input of a receiver. The receiving element is thus dependent on the driver and interconnect for maintaining the input signal in the narrow voltage bands specified for a valid logic 1 or 0.

Figure 14 illustrates the basic components of a single-ended connection. The driver differential pair outputs are biased to allow them to switch. The receiver, as with all ECL gates, is based on a differential amplifier. In the case of a single-ended receiver, the second input into the differential amplifier is not present at an external pin on the chip, but is instead connected internally to a V_{BB} reference voltage. As the signal present on IN+ goes either above or below the internal threshold set by V_{BB}, the receiver will switch.

While connections of this type are perfectly fine for logic functions, they should be avoided for a communications link. In a single-ended environment, any signal level differences (caused by temperature, logic family, transients, power supply noise, etc.) directly affect the received signal timing. In a logic function this timing variation limits a design both in determining how fast the system may operate, and in how much noise margin is present.

In a communications link these variations in timing translate directly into jitter in the serial data stream. Jitter affects a serial link by limiting not only how fast the link can operate (data rate) but also how far the data can be sent. Jitter is discussed in detail later in this document.

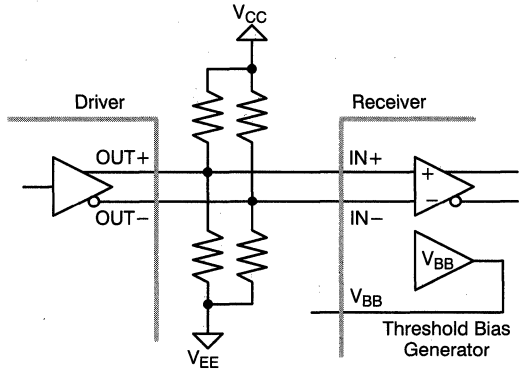


Figure 15. Differential Connection

The only expected single-ended connection on a HOTLink Transmitter is for a local loopback function to a HOTLink Receiver (when the INB− input is not available for a differential connection because it has been used as an ECL-to-TTL translator). In this connection it is expected that the transmitter and receiver are in relatively close proximity, such that the connection between them is more on the order of a logic connection than a communications link. The small amount of jitter caused by the single-ended connection will be far below the jitter susceptibility of the HOTLink Receiver.

HOTLink Transmitter Differential Connections

A differential connection is the preferred attachment for HOTLink Transmitter serial outputs. In a differential connection both outputs a of a driver are connected to the true and complement inputs of an ECL-compatible receiver. When connected in this fashion the majority of the interconnect dependencies are removed. The main advantages of a differential connection are insensitivity to the logic family, operating temperature, and power supply variations. In addition, the connection is now immune to most common-mode noise.

Figure 15 illustrates the basic components of a differential connection. The driver differential pair outputs are biased to allow them to switch. Now both true and complement inputs of the the receiver differential amplifier are available at external pins and are connected to the complementary outputs of the driver.

Some ECL differential receivers may also provide an external V_{BB} reference. This reference is provided for those cases where a driver is connected single-ended to one of the differential receiver inputs. The other receiver input must then be connected to the V_{BB} reference to allow the receiver to switch. With a true differential connection this V_{BB} output should remain open.

The main concerns in a differential connection are signal skew and crosstalk. Skew is the difference in arrival time of the OUT+ and OUT− signals at the receiver. Crosstalk is the coupling of energy between these same two signals.

As the amount of signal skew present in a differential connection is increased, the effective signal rise and fall times at the differential receiver also increase. In systems with large amounts of signal skew, it is possible for short pulses to never be detected by the receiver.

The main cause for signal skew is asymmetric routing of the true and complement signals between the driver and the receiver. A 1-inch difference in routing length is equal to about 150 ps of signal skew. This problem is corrected by maintaining matched signal runs between the HOTLink Transmitter and the ECL differential receiver.

The main cause for crosstalk is long parallel signal runs. The adjacent lines act as coupling transform-

ers and transfer energy from one to another. One cure for this is to limit the length of the connection by placing the ECL differential receiver as close to the HOTLink Transmitter as possible. Other possibilities are to route the two signals on opposite sides of a circuit board with an interposed power plane to act as a shield. If routing is to remain on the same plane, the crosstalk affects can be minimized by horizontally separating the two signals as far as possible or by routing a ground trace (with many vias to attach the ground trace to the ground plane) between the two signals.

HOTLink ECL Inputs

The ECL inputs on the HOTLink Receiver are also ECL 100K-level compatible. Similar to the transmitter, these inputs have also been enhanced to operate over a wider range than standard 100K ECL.

The differential INA± and INB± inputs offer improved minimum sensitivity of 50 mV, compared to 150 mV for the few 100K ECL differential receivers available. These inputs may be connected directly to either power rail without damage to the part, or changing the internal thresholds of other sections of the receiver. These same differential inputs also operate with a 3V common-mode rejection range (V_{CC} down to V_{CC} – 3V) that is twice the 1.5V range of standard 100K ECL differential receivers (V_{CC} – 0.5V down to V_{CC} – 2V).

Note: While differential outputs are quite common on ECL parts, true differential inputs are rare. The most common usage for differential inputs is on line receivers and clock drivers. The common-mode range on some parts with differential inputs is quite limited and should not be expected to operate over even a narrow range unless explicity stated in the manufactures datasheet.

The INA± inputs of the HOTLink Receiver should always be connected to a differential signal source. Since there is no V_{BB} reference output on the receiver there is no way to properly bias the second input of the differential receiver.

The INB± inputs may be configured to operate either as a differential receiver (in which case it should be connected to a differential signal source) or as two single-ended receivers. When operated as two single-ended receivers (as configured using the SO pin) the INB+ input operates as a 100K ECL single-ended receiver for serial data, while the INB−(SI) input operates as a 100K ECL single-ended receiver for an ECL-to-TTL level translator. The V_{BB} reference for these signals is available only inside the HOTLink Receiver and is not brought to an external pin. Signals connected to these single-ended inputs must now ensure operation within the 100K threshold levels.

Mixing ECL Logic Families

It is often desirable to use ECL parts of different families together in the same design. This can be done if certain rules are followed. The main reasons for these rules are the variability in signaling levels in ECL 10K family parts. *Figure 16* shows a DC-level comparison for 100K ECL outputs driving single-ended 10K ECL inputs.

In this configuration there is only 20 mV of margin between the 100K V_{OHL} and the 10K V_{IH} at the upper end of the temperature range. With 10K parts driving other 10K parts (assuming a common operating temperature) this is not a problem as the internal V_{BB} reference in each part follows a similar temperature shift. If the case temperature of the receiving 10K part can be kept below 35°C (100-mV margin), it can safely be used with 100K ECL parts for logic functions.

While the V_{OLH} specification appears to also have a noise margin problem, it does not. What occurs here is a condition where the receiver may be operated outside its linear region; i.e., 1s and 0s will be detected properly but the timing response may not match the manufacturer's data sheet.

Figure 17 shows the opposite configuration with 10K ECL logic driving either a single-ended 100K ECL receiver or a HOTLink Receiver. Here there are no tight margin areas between input and output thresholds. This means that 10K ECL parts can safely be used to drive 100K ECL inputs over their full temperature range.

Figure 17 also highlights the enhanced input range for the HOTLink Receiver. Unlike the narrow input range present on standard ECL families, the

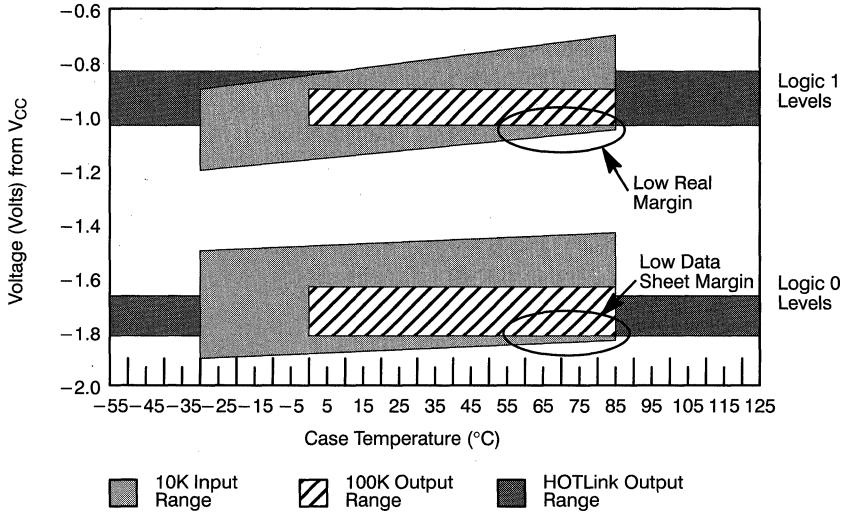


Figure 16. 100K ECL Driving 10K ECL

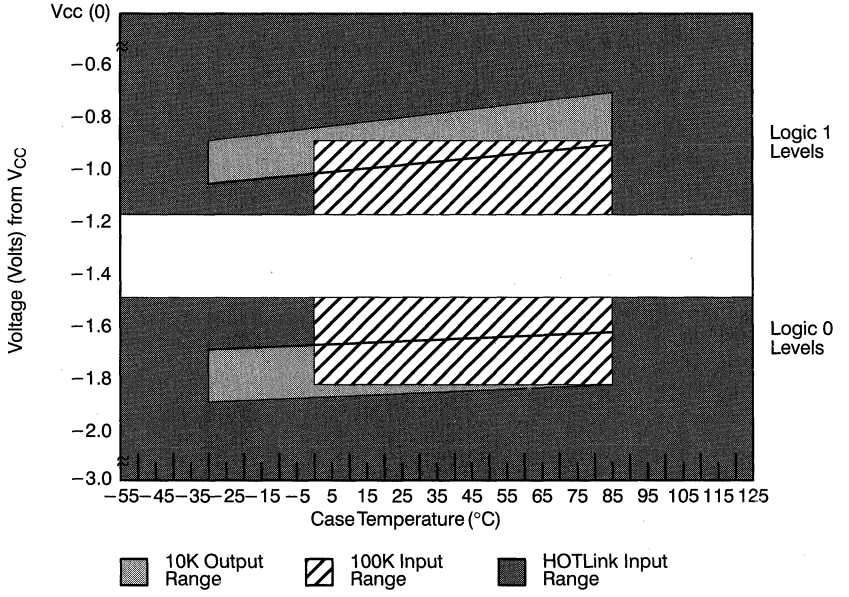


Figure 17. 10K ECL Driving 100K ECL

ECL inputs on the HOTLink Receiver maintain normal operation over the entire V_{CC} to V_{CC} – 3V range.

Single-Ended Connections

Both of these comparisons are based on single-ended connections, where only a single ECL output is used to drive the receiving internally referenced single-ended gate. In these cases, the other input to the receiving differential amplifier is connected internally to a V_{BB} reference. This type of connection should not be used to drive the INA± or INB± differential inputs of the HOTLink Receiver.

Differential Connections

One of the biggest advantages of ECL is the ability to communicate in a differential mode. This mode is relatively rare on logic parts (most commonly used for clock drivers and line receivers), as it requires both the driving and receiving parts to have both true and complement outputs and inputs respectively. When connected in this manner, the receiving part is no longer comparing the input signal to its V_{BB} reference, but instead compares the true and complement inputs to each other.

When used in this mode there is no problem using 100K ECL with 10K ECL at any temperature. Because an ECL receiver only requires around 250 mV of *difference* to fully switch, and the difference between the outputs of a differential driver remains near 800 mV, any differential connection has a minimum of twice the noise margin of a single-ended connection.

This type of connection is also immune to minor differences in the reference voltages between parts. Because the connection is differential, any common-mode voltages present on the received signals (due to power supply differences, AC coupling, ground shift, etc.) within the common-mode range are canceled out in the receiving differential amplifier. Some ECL parts with differential inputs can accept up to 1V of common-mode offset on the received signal without degradation of performance. The enhanced 100K ECL compatible inputs of the HOTLink Receiver can accept inputs between V_{CC} and V_{CC} – 3V, offering a common-mode range of 3V.

HOTLink Transmitter Connections

Unlike conventional negative-referenced ECL, the high-speed outputs on the HOTLink Transmitter are implemented in 100K positive-referenced ECL (PECL). This allows the TTL and ECL interfaces on the transmitter to operate from a common +5V power supply.

The HOTLink Transmitter has three differential output sections: OUTA±, OUTB±, and OUTC±. In addition to operating as 100K ECL-compatible signals, these outputs have been enhanced with additional features.

Power Saving Mode

A standard ECL output structure uses a constant current source at the base of a differential amplifier (see *Figure 5*). In these standard parts, this current source is enabled and dissipating power even when the outputs are not used.

The HOTLink Transmitter ECL outputs, while still operating as true 100K ECL outputs, incorporate some additional structures (see *Figure 18*) to save power when the outputs are not used. The differential amplifier (D1) under normal conditions will direct the I_S current from the current source through its internal transistors. As this current is switched, the output driver transistors (Q1 and Q2) change their operation point and the amount of current

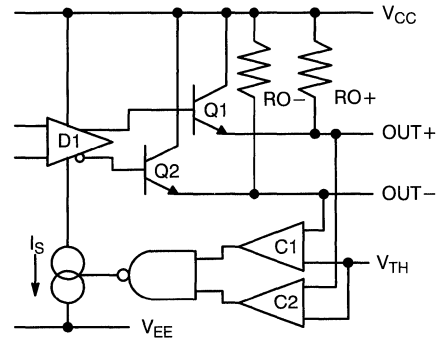


Figure 18. HOTLink Transmitter ECL Output

they source (a properly biased ECL output sources current in both 1 and 0 states; i.e., it does not turn off). Each output driver (Q1 and Q2) contains a high value pull-up resistor (RO+ and RO−) and a voltage comparator (C1 and C2). When both voltage comparators of a HOTLink differential output detect a voltage above a 100K ECL output-high level (V_{TH}), the current source (I_S) for that differential output pair is disabled. This results in a current savings of around 5 mA (25 mW) for each unused output pair.

FOTO Control of OUTA± and OUTB±

The HOTLink transmitter OUTA± and OUTB± differential outputs have an additional control input not present in the OUTC± output pair. While the OUTC± outputs are always enabled to follow the serial data stream generated in the HOTLink Transmitter shifter, the OUTA± and OUTB± outputs are not. These outputs are also controlled by a TTL-level input called FOTO (fiber-optic transmitter-off). While OUTA± and OUTB± are disabled, the OUTC± pair remains active and can be used for a local loopback source.

This FOTO signal is used to force the differential outputs of the OUTA± and OUTB± drivers to a state where a logical 0 is being driven. This state corresponds to a condition on optical modules where no light is transmitted. While not required for LED-based optical modules, this capability is required for laser-based links (see ANSI Z136.1 and Z136.2, F.D.A regulation 21 CFR subchapter J, and IEC 825) (References 9, 10, 11, 12, 13).

ECL Output Biasing

ECL outputs have specific loading requirements to insure proper operation. Because of the open-emitter structure of an ECL output, it can source current but cannot sink current. To allow the output to switch, some form of pull-down is required on the output. This pull-down usually takes the form of a resistive load; either to V_{EE} or $V_{CC} - 2V$.

Most ECL outputs are specified for driving load impedances as low as 50Ω. Because an ECL output does not swing rail-to-rail, this load is usually specified at $V_{CC} - 2V$, a point slightly below the ECL V_{OL}. At this point, when the ECL gate is driving a logic-0 level signal, a small current is running through the load resistor to keep the output transistor in the active region. Typical currents sourced when driving a logic-1 (I_{OH}) and logic-0 (I_{OL}) are calculated using *Equations 4* and *5* respectively, where R_T is the effective load impedance and V_{TT} is the effective bias voltage.

$$I_{OH} = \frac{V_{OH} - V_{TT}}{R_t} = \frac{(-0.9) - (-2.0)}{50\Omega} = 22mA \quad \text{Eq. 4}$$

$$I_{OL} = \frac{V_{OL} - V_{TT}}{R_t} = \frac{(-1.7) - (-2.0)}{50\Omega} = 6mA \quad \text{Eq. 5}$$

These I_{OH} and I_{OL} values are the basis for the timing and signal levels in the HOTLink datasheet. For other values of I_{OH} and I_{OL}, the transmitter will exhibit slightly different characteristics. These current flows can be achieved in many ways. The four most common methods are

- Shunt bias to V_{TT} bias voltage
- Shunt bias to V_{EE} bias voltage
- Thévenin bias to V_{TT} bias voltage
- Y-bias to V_{TT} bias voltage

Shunt Bias to V_{TT}

In shunt bias, as illustrated in *Figure 19*, a single resistor is used as a pull-down load on an ECL output to some bias voltage. When biased to V_{TT}, a single 50Ω resistor (R_T) from the ECL output to V_{TT} is all that is necessary. This requires an additional power supply to provide the ($V_{CC} - 2V$) V_{TT}-level. This termination type dissipates the least average-power (13 mW) of any output load type. It is often used in large ECL systems, in systems where overall power dissipation is a major concern, or where there is enough ECL present to warrant its design and implementation.

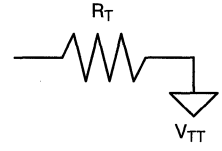


Figure 19. Shunt Bias to V_{TT}

Shunt Bias to V_{EE}

ECL outputs may also be biased to the V_{EE} supply as illustrated in *Figure 20*. Here a load resistance (R_T) of near 270Ω is connected to the V_{EE} supply to provide a similar current load for the ECL output driver. This value is determined by taking the average current flow for both a 1 and a 0 at the midway point (V_{BB}) in the output swing. The calculation for this is shown in *Equation 6*.

$$R = \frac{V_{EE} - V_{BB}}{\frac{I_H + I_L}{2}} = \frac{5 - 1.3}{\frac{22+6}{2}} = 264\Omega \qquad \text{Eq. 6}$$

Unlike the shunt bias to V_{TT}, this bias arrangement dissipates a significant amount of power in both the 1 and 0 states (47 mW average). This bias type (due to mismatched RC charge and discharge rates) exhibits a faster falling edge than rising edge. Because of this, its use is usually limited to logic functions, and is discouraged for serial links and for biasing differential output pairs. This is discussed in detail later in this document.

Thévenin Bias to V_{TT}

In a Thévenin bias network, a pair of resistors (R1 and R2) is used to create a load whose Thévenin equivalent matches that of a single resistor attached to a specific bias voltage (V_{TT}). For ECL this voltage is usually V_{CC} – 2V. These resistors are connected as illustrated in *Figure 21*. The values of R1 and R2 are solved using *Equations 7* and *8*.

$$R1 = \frac{V_{EE} \times R_T}{V_{EE} - V_{TT}} \qquad \text{Eq. 7}$$

$$R2 = \frac{V_{EE} \times R_T}{V_{TT}} \qquad \text{Eq. 8}$$

Solving for 50Ω and V_{CC} – 2V yields values of 82Ω and 120Ω for a 5V system. While this combination does provide a similar dynamic load to the shunt bias to V_{TT}, it dissipates nearly an order of magnitude more power (138 mW) than its shunt to V_{TT} equivalent.

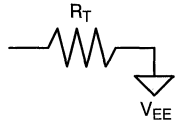


Figure 20. Shunt Bias to V_{EE}

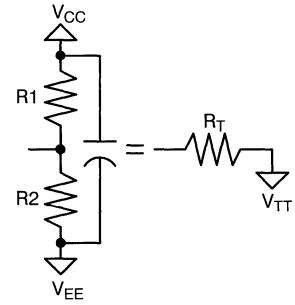


Figure 21. Thévenin Bias Equivalent

The capacitor shown in *Figure 21* is needed to allow R1 and R2 to provide the proper load for AC signals. In a Thévenin equivalent circuit, the power supply is assumed to be a short circuit. While this may be accurate for DC or very low frequency AC signals, the power supply appears as a near infinite impedance at RF frequencies. The bypass capacitor across R1 and R2 is used to create an AC short. This capacitor must be sized to operate as a short near the frequencies in use. For HOTLink-based systems this capacitor should probably be in the range of 300 pF to 0.01 µF.

Y-Bias to V_{TT}

Unlike the three previously described terminations, the, Y-bias can only be used with differential outputs. In this configuration the active ECL output (logic 1) is used to source current for a voltage divider, while the inactive ECL output (logic 0) is pulled down to the bias voltage created by this divider. A schematic of this bias network is illustrated in *Figure 22*.

Here R_T is the desired load impedance, usually 50Ω to V_{CC} – 2V for ECL systems. R_L is determined by summing the currents of a logic 1 and a logic 0 (as shown in *Equations 4* and *5*), and calculating the resistance necessary to drop the remaining voltage. This calculation is shown in *Equation 9* and solved for a 50Ω R_T.

$$R_L = \frac{V_{CC} - V_{EE} + V_{TT}}{I_{HI} + I_{LOW}} = \frac{3V}{28mA} = 107\Omega \qquad \text{Eq. 9}$$

This type of bias provides a significant power savings over a Thévenin bias because only a single pull-down resistor is used to dissipate power for two outputs. For a 50Ω equivalent load the power dissipation is only 110 mW for two outputs (55 mW for one). Just as with the Thévenin bias, a capacitor is necessary to create an AC short.

Matched Loading

Just as the differential amplifier in an ECL switch directs current flow, so do the emitter-follower output transistors. As these transistors are turned on an off, large amounts of current are switched through the driver's V_{CC} package pins. Because of the inductance present in these pins, transients can be induced in the internal V_{CC} supply.

Fortunately the effects of this lead-inductance only manifest themselves when the current through the V_{CC} supply pin *changes*. If the current is kept stable, no transients are induced. Due to the differential configuration of many ECL outputs, it is possible to keep this current stable by having matched loads on the true and complement outputs of the differential driver. This means that if a design uses one or both outputs of a differential driver, they both should drive loads of the same magnitude.

Figure 23 shows a differential output driver connected to a load including the package inductance present on the V_{CC} power pin. As the differential driver changes state, the overall current through L1 remains the same (assuming that both R_T loads are the same value).

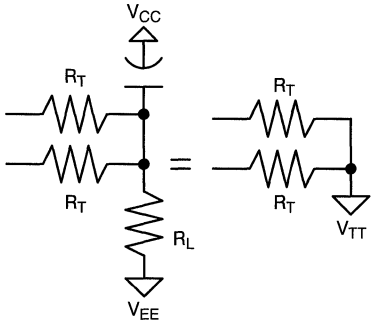


Figure 22. Y-Bias Network

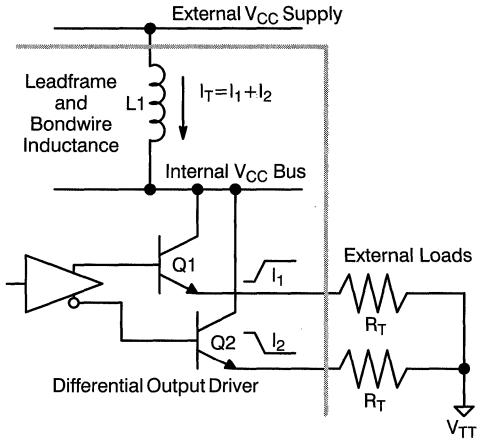


Figure 23. Loaded Differential Driver

If one of the two R_T load resistors is removed, some very undesirable things start to happen. The first is that the external power supply must now react to a dynamic rather than a static need for current. This increases the amount of power-supply bypassing that is needed next to the ECL driver V_{CC} pin. The second is a variation in the internal and external V_{CC} supplies caused by the dynamic current flow. This effect is illustrated in the following approximation.

For a single ECL output the current difference from a logic 1 to a logic 0 (into a 50Ω to V_{CC} – 2V load) is 16 mA (see *Equations 4* and *5*). From the ECL 100K family datasheets we know that signal transition times may be under 500 ps. By assuming the rise and fall portions of the signal are related to a triangular waveform, this transition may be roughly converted to a fundamental frequency using *Equation 10*.

$$F = \frac{1}{2 \times T_r} = \frac{1}{2 \times 500E^{-12}} = 1GHz \qquad \text{Eq. 10}$$

The Fourier series for a triangular waveform is listed in *Equation 11*. This illustrates that most of the energy content is present at the fundamental frequency with much smaller components present at the higher odd harmonics. To simplify the following calculations only the fundamental frequency is assumed to be present (Reference 24).

$$\frac{8V}{\pi^2}(\cos\omega_o t + \frac{1}{9}\cos 3\omega_o t + \frac{1}{25}\cos\omega_o t + ...) \qquad \text{Eq. 11}$$

If a package pin inductance of 4 nH is assumed (typical for many surface mount components), *Equation 12* can be used to determine the impedance of the package at this frequency.

$$X_L = 2\pi FL = 2\pi \times 1E^{9} \times 4E^{-9} = 25\Omega \quad \text{Eq. 12}$$

Using Ohm's Law we can now convert this change in current into an internal voltage change, as illustrated in *Equation 13*.

$$V = I \times X_L = 16mA \times 25\Omega = 400mV \quad \text{Eq. 13}$$

This temporary difference between the internal V_{CC} and the external V_{CC} supply is the same phenomenon known in a TTL environment as ground bounce.

All of this, of course, is based on the assumption that the output will be able to switch at this speed (500 ps) and provide the specified current (16 mA) when presented with a high-impedance source. What actually occurs is that the output edge slows down to match the current transfer permitted by the on-resistance of the output driver transistor and the package reactance.

Most ECL parts use a couple of different techniques to combat this problem. Both are quite simple to implement. The first is to use a separate package pin to provide power to the emitter-follower output transistors. This prevents any V_{CC} shift caused by the output drivers from affecting the sensitive differential amplifiers and voltage references present in other parts of the device.

The second method is to maintain a balanced load on the differential output drivers. Since the rising and falling edge rates of ECL are very symmetrical, $\Delta I_1 = \Delta I_2$. Because these changes in output current are symmetrical, $\Delta I_T \cong 0$. From *Equation 13* we know that any induced ΔV is directly proportional to ΔI; thus as ΔI goes to 0, so does ΔV.

AC Characteristics of Output Drivers

In an ECL driver, the time it takes for the signal to rise is largely determined by its internal resistors and parasitic capacitors (C_{int} and R_{int} in *Figure 24*), since the emitter-follower can supply large currents to charge the load capacitance. The DC voltage to

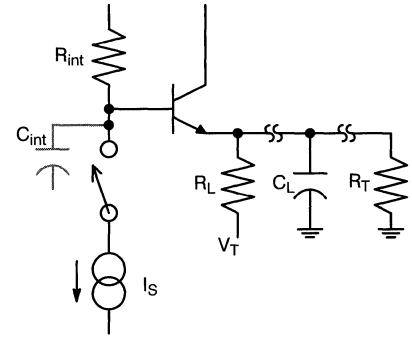


Figure 24. ECL Output Driver with Loading

which the output rises is determined by the emitter-follower transistor characteristics and the internal driver resistor (R_{int}) value. However, the AC voltage (overshoot, ringing, etc.) is determined primarily by the load characteristics. A capacitive load (along with the inductance found in the package, printed circuit traces, and other load components) causes the output to rise significantly beyond the anticipated DC output level, since the emitter-follower cannot supply any compensating current at the top of its transition.

Unlike the output rise time, the fall time is primarily determined by the time constants of the load capacitance and pull-down circuit. The output LOW voltage (V_{OL}), is determined by R_{int}, I_S, and the characteristics of the emitter-follower transistor. In a properly designed system the load circuit has time constants comparable to (or shorter than) the internal fall time, such that the emitter-follower can source a small amount of current during the entire time it is switching from HIGH to LOW. If this is not true, the emitter-follower transistor will be shut off for part of the transition time, and the output will follow the time constant of the load.

Figure 25 illustrates the effects of two different load or bias circuits. The assumption in both of these examples is that the load circuit controls the fall time of the signal, and that the pull-down current is being supplied by a resistor to a V_T of either V_{CC} – 2V or V_{EE} (+3V or ground for a PECL environment). In the dashed curve, the standard ECL load of 50Ω to V_{CC} – 2V is used, causing an output current of

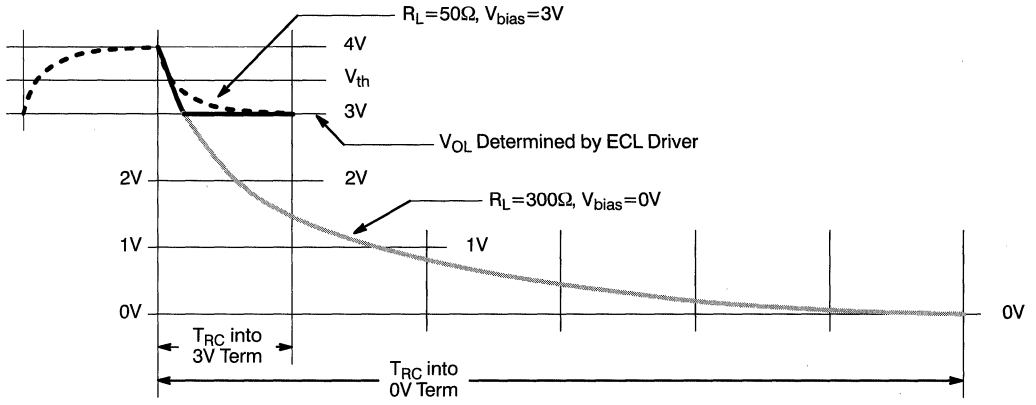


Figure 25. Falling Edge Rate Comparison for Bias to V_{TT} and V_{EE}

approximately 20 mA when the output is HIGH, and 5 mA when the output is LOW. This load (or its equivalent) can be created using all of the previously described bias networks except shunt bias to V_{EE} (shown in the solid curve).

The same amount of pull-down current can be realized with a single resistor (R_L in *Figure 24*) in a shunt bias to V_{EE} configuration. To get a comparable output current (and assure comparable voltages at the output) the pull-down resistor would be chosen to sink approximately the average of I_{OH} and I_{OL} when connected to a voltage midway between V_{OH} and V_{OL} (see *Equation 6*). The I_{OH} and I_{OL} currents listed here would yield a pull-down resistor of around 300Ω. This type of bias is perfectly correct and adequate for ECL logic circuits where the mismatch between rise and fall times is absorbed into the normal logic delays and set-up times. In a data transmission system the effects of this type of output bias can be unpredictable and will often degrade performance.

Figures 25 and *26* illustrate the difference in output fall time assuming a constant load capacitance, with the only variation being the bias resistor and voltage. The 50Ω load resistor (dashed line) follows an RC discharge curve which ends at $V_{CC} - 2V$. For normal loading this *soft* edge rate more closely matches the rise time of the output as controlled by the emitter-follower, and is less affected by variations in load capacitance and reflection currents.

The 300Ω load resistor (solid line) follows an RC discharge curve which would normally end at V_{EE}

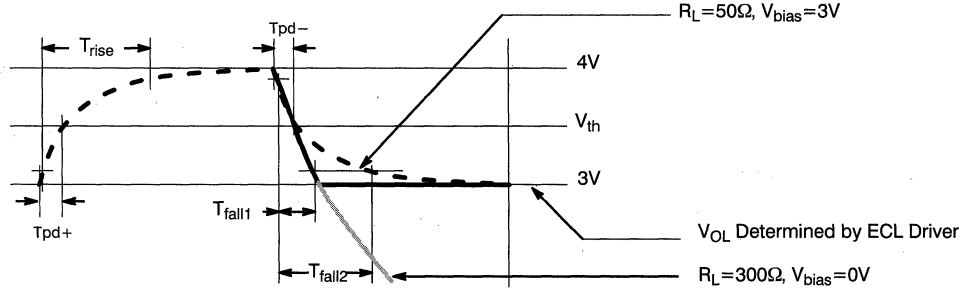


Figure 26. Expanded Detail of Falling Edge Rate Comparison

(ground). While this appears to have a crisper edge rate, it will be more severely affected by load capacitance variation and transmission line reflection currents that must be accommodated.

Figure 26 shows that with either pull-down the total voltage swing is the same and is determined by the internal voltage swing of the driver, as buffered by the emitter-follower transistor. While the RC curve for the 300Ω pull-down continues to V_{EE}, the emitter-follower is turned on and sourcing current at the V_{OL} point and does not allow the output to continue farther down the curve.

In either configuration the signal delays match, since both falling edges cross the mid-swing line at the same time, but the rise and fall times are different. These rise and fall times determine the higher frequency spectral components of the waveform. Differences in these spectral components affect the termination efficiency and waveform distortion caused by cable attenuation (Reference 14).

Transmission Line Termination

While often confused with ECL output biasing, termination of transmission lines is something quite different. Because of the reactive characteristics of transmission line termination, the resistors used for termination may often be used as part of the output bias network, but they perform different functions.

Due to the high switching speeds of ECL, most of the interconnect between parts cannot be treated as simple connections. They must instead be treated as transmission lines. The distance between parts, in conjunction with the signal loading and rise and fall times, is used to determine at what point the interconnect must be treated as a transmission line. The general assumption is that short lines do not require termination, while long ones do. The determination of what is a long line is made using *Equation 14* (Reference 5).

$$\ell \max = \frac{1}{2}\sqrt{\left(\frac{C_L}{C_O}\right)^2 + \left(\frac{Tr}{\delta}\right)^2} - \frac{C_L}{2C_O} \qquad \text{Eq. 14}$$

The values for this equation for microstrip construction on G10/FR4 type board would be

- l max – maximum unterminated line length
- Tr – source 20% to 80% rise time
- C_L – load capacitance (2 pF assumed for a load)
- δ – delay per unit length (0.148-ns/inch)
- C_O – capacitance per inch

Running this calculation for various impedance and rise-time combinations yields the lengths listed in *Table 3.* Lengths beyond those listed here require termination.

Table 3. 100K ECL Maximum Unterminated Line Length (in inches), Microstrip Construction

	Line Length (in inches)		
Z_O	0.5 ns	1 ns	1.5 ns
50Ω	1.38	3.06	4.74
62Ω	1.32	2.99	4.67
75Ω	1.25	2.91	4.59
90Ω	1.18	2.82	4.50
100Ω	1.14	2.76	4.44

The lengths listed in *Table 3* assume digital switching characteristics. The HOTLink ECL serial signals are, for the most part, analog in nature. This effectively shortens the maximum unterminated length. For HOTLink serial signals, any ECL trace greater than one inch in length should be terminated.

The objective of transmission line termination is to prevent reflection of power from the destination back to the source. This is accomplished by terminating a transmission line in its characteristic impedance (Z_O). The two basic types of line termination are referred to as series and parallel termination.

The actual amount of the source signal reflected is based on how well the line impedance matches the destination impedance. This determines how much voltage is reflected back into the transmission line. This ratio of reflected voltage to incident voltage is called the reflection coefficient ρ (rho) and is shown in *Equation 15* (Reference 5).

$$\frac{V_r}{V_i} = \rho = \frac{R_T - Z_O}{R_T + Z_O} \qquad \text{Eq. 15}$$

Series Termination

Series termination (sometimes referred to as source termination) requires that the load be high-impedance to properly operate. This type of line termination is **not** recommended for use with HOTLink because of the reactive nature of all parts at the high frequencies present on the HOTLink ECL signals.

Parallel Termination

In parallel termination the desired characteristic is to terminate the *end* of the line (rather than the source) in its characteristic impedance. This results in a reflection coefficient of zero; i.e., no signal is reflected. This type of termination is implemented the same as shunt bias networks. *Figures 27* and *28* show the two equivalent forms of parallel termination.

Parallel termination offers the advantages of allowing distributed loads on the transmission line, and of having the termination network also operate as the bias network.

In the single-resistor form of parallel termination illustrated in *Figure 27*, the R_T resistor is sized to match the Z_O impedance of the transmission line. This termination form has the same advantage as the single resistor shunt bias because it dissipates less overall power than the Thévenin equivalent termination. It also has the same drawback of requiring a separate power supply.

In a Thévenin equivalent termination (illustrated in *Figure 28*) two resistors (R1 and R2) are used to form an equivalent circuit to that in *Figure 27*. *Table 4* lists the R1 and R2 resistor values for a number of common transmission line impedances. This table assumes operation with a 5V source and a termination voltage of V_{CC} − 2V, and selects the nearest standard 1% resistor value when an exact match is not available. These values are calculated using the same *Equations 7* and *8* as used for calculating a Thévenin bias network (Reference 15).

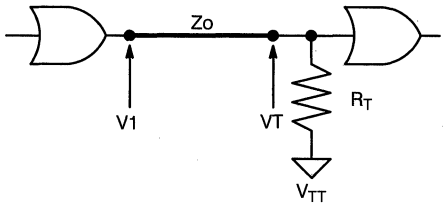


Figure 27. Parallel Termination to V_{TT}

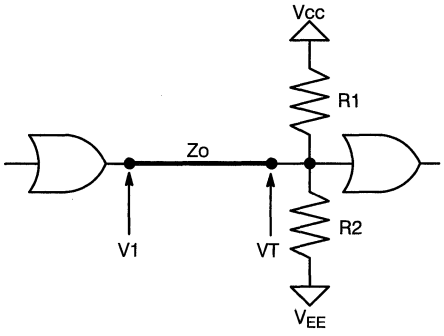


Figure 28. Thévenin Equivalent Parallel Termination

Table 4. Thévenin Bias Resistor Values

Z_O	R1	R2
50Ω	82.5	124
70Ω	118	174
75Ω	124	187
80Ω	133	200
90Ω	150	226
100Ω	165	249
120Ω	200	301
150Ω	249	374

Terminating HOTLink Transmitter ECL Signals

The HOTLink CY7B923 transmitter has three different ECL differential output pairs named OUTA±, OUTB± and OUTC± (see *Figure 2*). How (or if) these outputs are terminated is dependent on what the output is used for.

OUTC±

The OUTC± outputs of the HOTLink Transmitter are not controlled by the transmitter FOTO signal and are thus always enabled to drive serial data. While fully capable of driving either optical mod-

ules or copper cables, it is expected that the most common usage of this differential output will be as a local loopback to a HOTLink CY7B933 Receiver INB± inputs.

This signal may be connected to the HOTLink Receiver either differentially or single-ended. When connected differentially, the OUTC+ output is connected to the INB+ input, and the OUTC− output is connected to the INB− input. When connected single-ended, the OUTC+ output is connected to the INB+ input.

Note: For the INB+ input to be used differentially, the SI/SO ECL-to-TTL translator (mapped through the INB− input) must be disabled. This is done by connecting the SO output directly to V_{CC}.

Once the connection is made, the type of termination required is determined by the distance between the HOTLink Transmitter and the HOTLink Receiver. If the distance is kept short enough (under one inch) (Reference 5) no termination is required and the output only needs to be biased. This can be done with a single pull-down resistor to V_{EE}. While this type of termination does induce some jitter into the serial data stream (due to mismatched rise and fall times), the amount is well within the receiver limits.

If the distance is greater than one inch, the line should be terminated (Reference 5). To do this correctly requires determination of the characteristic impedance of the board traces used to connect the source and destination. Please see the Cypress Semiconductor application note "Driving Copper Cables with HOTLink" for information on how to determine the characteristic impedance of various types of transmission lines (Reference 16).

For local connections that do not travel through external transmission media (i.e., coax, twisted-pair, optical fiber, etc.) parallel termination may be used. The important consideration here is that both the OUTC+ and OUTC− outputs must be terminated/biased into the same size of load to maintain a current balance inside the HOTLink Transmitter.

If neither of the OUTC± outputs are used, both outputs should be left open or pulled up to V_{CC} to disable the current source for the differential driver (see *Figure 18*).

OUTA± and OUTB±

The OUTA± and OUTB± outputs of the HOTLink Transmitter are both controlled by the FOTO signal which is required to meet laser safety regulations for communications links (References 9, 10, 11, 12, 13). Other than this special enable signal, these outputs operate the same as the OUTC± outputs.

Driving Optical Modules

When connecting to optical modules, it is best to drive the optical module data inputs differentially. This provides the highest noise immunity for the system, and the lowest signal jitter. When used with *de facto* standard optical modules this becomes mandatory because the optical modules have a differential data input, yet do not provide a V_{BB} supply to bias the other input of the differential amplifier of the optical transmitter. Because this interface is intended for driving some external segment of optical cable, series termination (which uses shunt bias to V_{EE} and increases jitter) should not be used. Since the HOTLink parts will most probably be the only ECL parts in the system, the recommended termination is a Thévenin or Y-termination.

Both the Thévenin and Y-terminations provide the bias necessary for the ECL signal to switch, and the impedance necessary to terminate a transmission line. One of these types of termination/bias should be used even when the distance from the HOTLink Transmitter to the optical transmitter is short. This is necessary to maintain symmetrical rise and fall times for the OUTx± differential outputs.

PECL Optical Modules

Interfacing to optical modules in PECL mode is quite simple, requiring only a few passive parts. The schematic in *Figure 29* illustrates the connections and parts necessary for this type of connection.

One of the key items often missed in this type of connection is proper bypassing of the termination/bias networks. The theory behind a Thévenin network is that the power supply is considered as a short for AC. While this may be true for near DC applica-

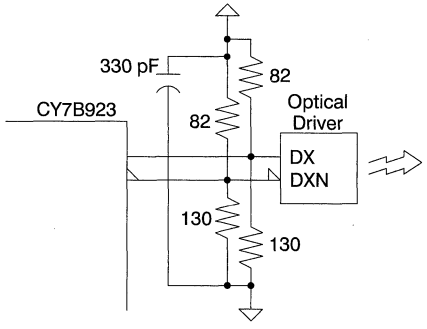


Figure 29. HOTLink Transmitter-to-PECL Optical Module

tions, the base frequencies and harmonics present in the HOTLink Transmitter output are far beyond any frequency the power supply itself could pass.

To make the power supply a short, a capacitor must be placed across the Thévenin pair. The size of the capacitor is determined by the frequency of operation of the serial link. A good rule-of-thumb is to pick the largest value capacitor whose series resonant frequency is 30% above the highest baseband frequency of the baud rate of the serial data (Reference 17). Since the data is sent using an NRZ modulation (non-return-to-zero), the highest baseband frequency is one half the serial bit-rate (Reference 18).

Another important characteristic is the dielectric type of the capacitor. For this type of analog operation, a good high frequency RF type capacitor must be specified. This means specifying either NP0 or C0G type capacitors.

Standard ECL Optical Modules

Those optical modules with the case connected to V_{CC} are designed for use in a negative DC supply system. These types of modules may also be driven by a HOTLink Transmitter.

By far the simplest method is to connect the module the same as a PECL module, with the exception of the Case pins. Here, instead of attaching the Case pins to ground (V_{EE}), they are attached to V_{CC}. If the case is metallic in nature, care must then be exercised such that it does not come into direct contact with ground.

If the optical module is to be used below ground, it must be AC coupled to the HOTLink Transmitter. This type of connection is illustrated in *Figure 30*.

The HOTLink Transmitter outputs are biased the same as for a PECL optical module. AC coupling capacitors are used to connect the HOTLink Transmitter positive-referenced ECL outputs to the negative-referenced ECL inputs of the optical module. These coupling capacitors actually operate as a bandpass filter, centered around their series resonant frequency. To pass additional low- or high-frequency components, additional capacitors should be placed in parallel with the coupling capacitors.

Capacitively coupled signals require DC restoration and, if the connection length warrants, transmission line termination. DC restoration is necessary to place the signal swings in the input range of the ECL receiver. Unlike ECL outputs, which are biased to a level slightly below their V_{OL}(min)-level (V_{CC} – 2V), AC coupled ECL inputs need to be biased to the center of the receiver input range. This is the same as the V_{BB} reference point of V_{CC} – 1.3V. In *Figure 30*, this reference point is created from a resistive divider network, and bypassed with a 0.01-µF capacitor to provide the dynamic current response needed for the differential inputs.

While many optical modules or ECL gates generate a V_{BB}-level, this output *must not* be used to bias this

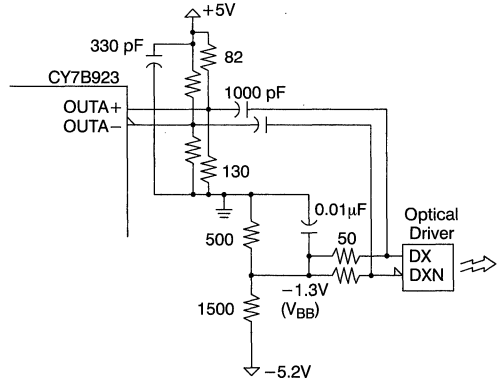


Figure 30. HOTLink Transmitter-to-Negative-Referenced ECL Optical Module

reference point because it cannot provide sufficient dynamic current. The V_{BB} output of an optical module, or other ECL gate, is an unbuffered tap of the internal V_{BB} reference. While fully capable of delivering the few μA of current necessary to drive an input, it cannot tolerate the transient currents present at the end of a low-impedance transmission line. Because the V_{BB} source is unbuffered, this also means that any external transients applied to it will move the V_{BB} reference *inside* the receiver, with unpredictable consequences.

While it is possible to create a V_{BB} power amplifier (by using multiple ECL buffers in parallel) to create a buffered form of V_{BB}, such amplifiers should not be used with HOTLink. They are prone to oscillation and ringing. Such amplifiers should also not be used for DC restoration (as needed here) because the V_{BB} amplifier is not quite DC stable; i.e. its output usually contains a low-level (10–50 mV) oscillation whose frequency is set by the delay through the part. This low-level noise is not a problem for logic applications, but for analog applications causes increased jitter on the biased signals.

In this example, the V_{EE} for the optical module is set to −5.2V. This is a common supply voltage for ECL circuits. If a different supply voltage is used, the values in the resistive divider must be changed to maintain the V_{BB} reference point at V_{CC} − 1.3V.

One drawback of this circuit is the inability to react to a DC state in the data stream. If the HOTLink Transmitter is set to transmit all 1s or all 0s (e.g., FOTO is set to disable transmitting), the optical module inputs will both return to a V_{BB}-level. At this level the optical module's output will probably oscillate due to the high gain present in the optical module's ECL-to-optical translator. In this AC coupled configuration (when operated with laser-based optical drivers) it is necessary to use some method other than FOTO to meet the laser safety restrictions (References 9, 10, 11, 12, 13).

Driving Copper Media

The ANSI Fibre Channel Standard currently identifies both coaxial cable and shielded twisted-pair as supported copper media types. The HOTLink Transmitter easily interfaces to these and many

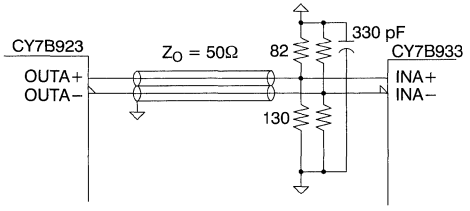


Figure 31. Direct-Coupled, Copper Interface

other types of copper media, and allows communicating on them at distances well beyond the lengths called out in the ANSI Standard (Reference 3).

Numerous characteristics determine how far a signal can be transmitted on copper media. The most important of these are:

- Voltage amplitude of the signal fed into the cable
- Jitter and ringing on the source signal
- Attenuation characteristics of the cable
- Length of the cable
- What (if any) equalization is used in the system
- Receiver loading and sensitivity

Coupling to the cable (transmission line if on a backplane) may be done in multiple ways, depending on the media type and distances involved.

Direct Coupled

For those instances where the signal never leaves the same chassis (or even the same board) it is possible to directly couple to the media. Here the media is effectively the circuit board traces, runs of twisted-pair, twinax, or dual coax. The main criteria here is that there must be no chance for a significant V_{CC} reference difference (transient or DC) between the HOTLink Transmitter and HOTLink Receiver, including any common-mode induced noise. For the HOTLink Receiver, this maximum difference is around 1V. Under these conditions the HOTLink Transmitter and Receiver may be connected as illustrated in *Figure 31*.

While *Figure 31* shows a 50Ω transmission line, the actual impedance can be higher than this. For other impedance values it is necessary to change the Thévenin termination networks.

When sent through twin coaxial cables (as shown in *Figure 31*) or two separate transmission lines, care must be taken to make sure that both lines are electrically the same length. Any difference in length causes one of the two transmitted signals to arrive at the receiver input either leading or lagging the other. This difference manifests itself as jitter in the receiver. If twisted-pair or twinax is used instead, both the OUTA+ and OUTA− signals combine to form a single signal sent down a balanced transmission line.

Capacitor Coupled

For configurations where it is possible to have significant ground or reference differences, some form of AC coupling becomes necessary. If the signals remain in a well protected environment (minimal EMI/ESD exposure) this AC coupling can be performed with capacitors. When this is done, bias/termination networks are required at both ends of the cable. A schematic detailing this type of connection is shown in *Figure 32*.

Good low-loss RF-grade capacitors should be used for this application. These parts are available in many different case types and voltage ratings. The capacitors used must be able to withstand not just the voltage of the signals sent, but of any DC difference between the transmitter and receiver and the maximum ESD expected. A typical 1000-pF 50-WV C0G capacitor would be available in an 0805 surface mount case size (0.08″L x 0.05″W x 0.02″H). For on-board applications a 50-WV rating should be sufficient. While capacitors with much higher breakdown voltages are available, both cost and space make their use prohibitive. This same 1000-pF C0G

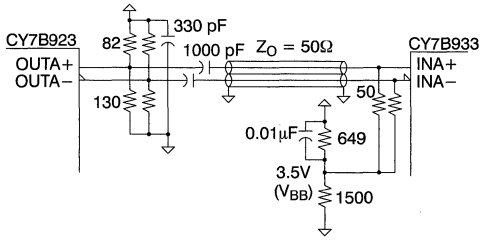


Figure 32. Capacitive-Coupled, Copper Interface

capacitor at 5-kV breakdown is almost a half cubic inch in size (Reference 15).

This type of coupling is very similar to that used to drive an optical module that is not at the same reference as the HOTLink Transmitter. Since the HOTLink Receiver and an optical module both operate with ECL 100K-level compatible inputs, this should be expected.

In this configuration, the receiver reference point is set slightly different from that for a standard ECL receiver. Part of this is due to the HOTLink Receiver being designed for operation at +5V rather than −5.2V or −4.5V. The other is that the HOTLink Receiver has a wider common-mode range than standard 100K ECL parts. To allow operation over the widest range of signal conditions the V_{BB} bias network on the receive end of the transmission line is set to the center of the HOTLink Receiver 3V common-mode range at V_{CC} − 1.5V.

This capacitively coupled interface is not recommended for cabling systems that leave a cabinet or extend for more than a few feet. This is primarily due to

- Limited voltage breakdown under ESD situations of the coupling capacitors
- ESD susceptibility of the receiver due to transients induced in the cable
- Limited common-mode rejection at the receiver end

Addition of a second set of coupling capacitors at the receive end may improve some of these characteristics, but it will not remove them.

Transformer Coupled

The preferred copper attachment method is to transformer couple to the media. Transformers have multiple advantages in copper-based interfaces. They provide

- High primary-to-secondary isolation
- Common-mode cancelation
- Balanced-to-unbalanced conversion

The transformer is similar to a capacitor in that it also has passband characteristics, limiting both low

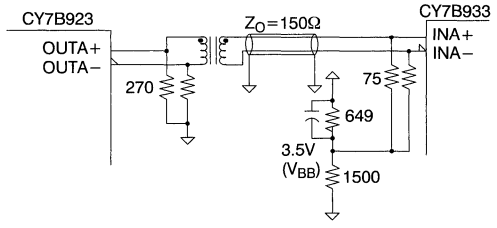


Figure 33. Transformer-Coupled, Copper Interface

and high frequency operation. Proper selection of a coupling transformer allows passing of the frequencies necessary for HOTLink serial communications. A schematic detailing a transformer coupled interface is shown in *Figure 33*.

This transformer-coupled configuration has many similarities to the capacitively coupled interface. It still provides DC isolation between the HOTLink Transmitter and Receiver, and requires the V_{BB} bias and termination network at the receiver.

The connection at the HOTLink Transmitter is quite different now. The output bias network is now a simple pull-down to V_{EE}. While this causes the transmitter outputs to have asymmetric rise and fall times, it does not add to the system jitter. Instead, the true and complement outputs combine in the transformer to provide a single signal with symmetrical rise and fall times. This bias arrangement also the has the advantage of delivering the entire transmitter output voltage swing into the transformer, rather than part into the transformer and part into the bias network.

The configuration shown in *Figure 33* uses only a single transformer and either 150Ω twinax or twisted-pair as the transmission line. This can be done because the transmission system remains balanced from end to end. Here the primary functions of the transformer are to provide isolation and common-mode cancelation.

In a single transformer configuration the transformer should be placed at the source end of the cable. Unlike the HOTLink differential receiver, which has a full 3V common-mode range, an ECL output (when sourcing a zero or LOW-level) will respond to high-going signals picked up on the transmission line.

In *Figure 34* a second transformer is added to the transmission system at the receiver end of the cable. This configuration allows use of either balanced or unbalanced (coaxial) transmission lines. The configuration shown here is a 75Ω coaxial cable system. Here the first transformer is used for balanced-to-unbalanced conversion, while the second transformer provides unbalanced-to-balanced conversion.

HOTLink Receiver ECL Inputs

The HOTLink Receiver has five 100K ECL (PECL) compatible inputs: INA+, INA−, INB+, INB−(SI), and A/$\overline{B}$ (see *Figure 3*). The A/$\overline{B}$ input is used to select which serial data input (INA± or INB±) is fed to the receiver PLL and shifter.

The INA± differential input is normally used for the primary received data input. This input is only functional as a differential receiver. To use it as a single-ended receiver, a V_{BB} reference would have to be attached to one of the INA± inputs. Since the HOTLink Receiver does not provide a V_{BB} output, this must come from either an external ECL gate or a resistive divider. Because neither of these sources can be guaranteed to be at the exact internal V_{BB} reference of the HOTLink Receiver (and will thus introduce jitter into the system), operation of INA+ in single-ended mode is not recommended. Also, operation in single-ended mode generally takes twice the signal swing (100 mV for HOTLink) for a receiver to properly detect data.

The INB± differential input is expected to be used as the local loopback receiver. It is capable of being operated as a differential receiver, or as two single-ended receivers.

To operate the INB± inputs as a differential receiver it is necessary to have the SO output either

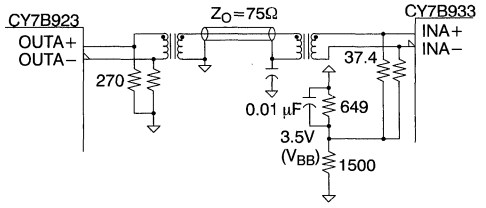


Figure 34. Dual Transformer-Coupled, Copper Interface

directly connected to V_{CC} or pulled up to V_{CC} through a resistor (minimum of V_{CC} – 250 mV). This pin, while normally used as an output, has a voltage comparator on the output to both disable it and to operate the INB± inputs as a differential pair. When used as a differential receiver the INB± inputs operate the same as the INA± inputs.

If the SO pin is instead allowed to remain in the standard TTL output range (below V_{CC} – 850 mV), it is enabled as a TTL-level driver, and is the output end of an ECL-to-TTL level translator. In this mode the HOTLink Receiver INB+ input is a single-ended ECL receiver for serial data, while the INB– input becomes the input end of the ECL-to-TTL translator. The expected use of this translator is for converting an ECL carrier-detect signal to TTL levels.

ECL Input Levels

Unlike standard 100K ECL logic, the HOTLink ECL inputs are designed to operate, not only over the full 100K ECL voltage and temperature range, but substantially beyond as well.

Normally 100K ECL inputs should never be raised above V_{CC} – 700 mV. If this occurs, the input transistor saturates and can damage other internal structures in the gate. Because the HOTLink Receiver is designed for use in a communications environment, its input structures are more robust and can be taken all the way up to V_{CC} with no degradation in performance. This provides a common-mode operating range more than twice that of standard ECL.

The HOTLink ECL Receivers also provide higher gain than that available from standard 100K ECL. The receiver is able to fully detect 1s and 0s with as little as 50 mV of differential signal at the inputs. Those few 100K ECL parts capable of differential operation usually specify this at 150–200 mV.

The HOTLink ECL inputs are also robust on the V_{IL}(min) side. When operated in differential mode these inputs provide full functionality down to V_{CC} – 3V, yielding a full 3V common-mode operating range. For single-ended operations these same inputs can be taken all the way to V_{EE} (ground or 0V).

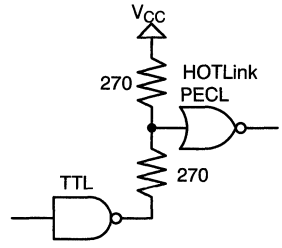


Figure 35. TTL-to-HOTLink PECL Interface

Controlling A/B from TTL

While the A/B path select on the HOTLink Receiver is a PECL input, it can be controlled from a TTL driver with as few as two resistors. Controlling a traditional PECL input from TTL normally requires a third resistor to limit the high state to the specified V_{IH}(max). Only a two resistor divider is needed with the HOTLink Receiver (as illustrated in *Figure 35*) because it can tolerate a full V_{CC}-level on its ECL inputs.

HOTLink Receiver Biasing

Unlike ECL outputs, which always require an output bias to create the output-low level, ECL inputs instead require levels within their input range to allow them to switch. When the HOTLink Receiver is directly connected to the biased output of either a 10K, 10KH, or 100K ECL driver (see *Figures 16* and *17*), these conditions are satisfied.

PECL Optical Modules

Connecting a PECL optical module to the HOTLink Receiver is the same as connecting two ECL parts together. This is connection is illustrated in *Figure 36*.

A bias network is required on the output of the optical module to allow it to switch. A Thévenin or Y-bias network should be used on the high-speed serial lines (RO and NRO as illustrated in *Figure 36*) to keep induced jitter to a minimum. The signal- or carrier-detect output (SIGO) of the module is considered a logic level signal and only requires a pull-down type of biasing to allow the output to switch.

If the distance between the optical module and the HOTLink Receiver is short (see *Table 3*) then the bias network may be placed anywhere between the optical module and the HOTLink Receiver. If this distance is long, then the interconnect traces must be treated as a transmission line and the bias network must be moved to the receiver to also act as line termination. If the transmission line impedance is other than 50Ω, then different values of resistors are necessary (see *Equations 7* and *8,* and *Table 4*).

Standard ECL Optical Modules

Optical modules with the Case pins connected to V_{CC} are designed for use in a negative DC supply system. These types of modules may also drive a HOTLink Receiver.

By far the simplest method is to connect the module the same as a PECL module, with the exception of the Case pins. Here, instead of attaching the Case pins to ground (V_{EE}), they are attached to V_{CC}. If the case is metallic in nature, care must then be exercised such that it does not come into direct contact with ground.

If the optical module is used below ground it must be AC coupled to the HOTLink Receiver. A schematic detailing this type of connection is shown in *Figure 37.*

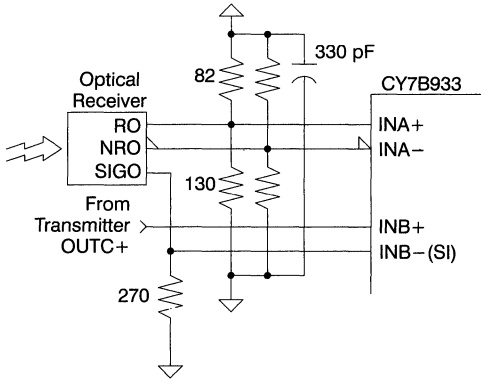


Figure 36. PECL Optical Module-to-HOTLink Receiver

From a parts count standpoint this type of connection should be avoided if at all possible. Just as with the HOTLink transmitter-to-negative referenced ECL optical modules, this interface requires biasing on both sides of the AC coupling capacitors.

Because the signal detect output of the optical module is not an AC signal, capacitive coupling cannot be used to feed this signal into the HOTLink Receiver INB– input. The simplest thing to do here is to use an external ECL-to-TTL translator (as illustrated in *Figure 37*) to convert the signal-detect output to a positive referenced TTL environment.

The INA± differential inputs must be biased to near the midpoint of the common-mode range of the HOTLink Receiver. The two 50Ω resistors tied to this synthesized reference point are sized to properly terminate the transmission line impedance of the interconnect.

Receiving from Copper Media

The direct-coupled, capacitor-coupled, and transformer-coupled configurations for copper interconnect are covered in the HOTLink transmitter-to-copper interface section of this document, with schematics of these connections illustrated in *Figures 31* through *34.*

Signal-Detect for Copper Interface

When interfacing to optical modules, the generation of a carrier- or signal-detect function is a simple connection to an ECL output. With a copper interface, this signal-detect function must be built from other components.

The key to a good signal-detect implementation is to create one that accurately detects the presence or absence of a valid data stream, yet does not load or distort the received signal. A sample carrier-detect circuit is shown in *Figure 38.*

This circuit uses a reference divider-network similar to that in *Figure 37,* except that an additional voltage reference point is created. This new reference point sets a threshold for received amplitude at which the signal detect circuit will start to respond. For this example, this reference point is set to 100 mV above the carrier detect receiver V_{BB} reference point. This 100-mV offset is also necessary to prevent the

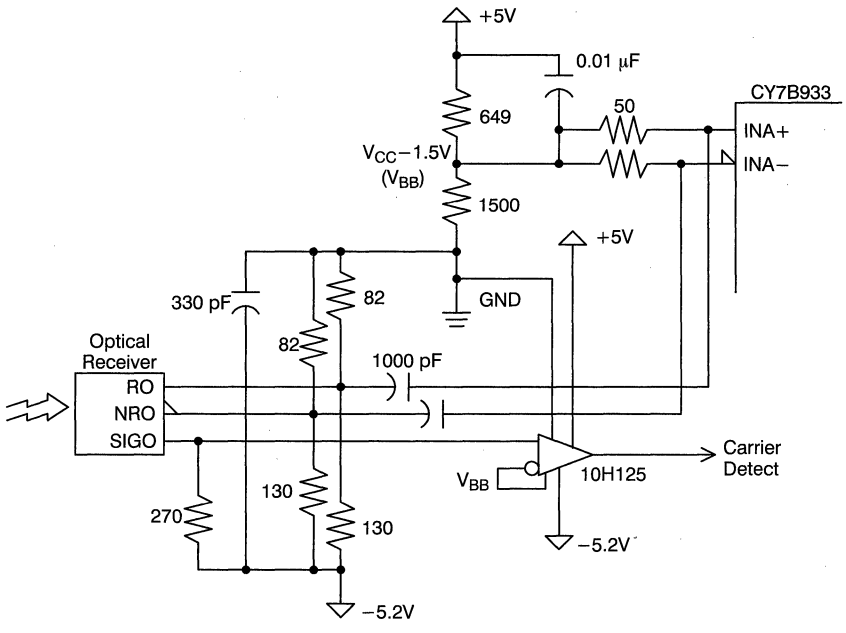


Figure 37. Negative-Referenced Optical Module-to-HOTLink Receiver

10H116 amplifiers from oscillating when no signal is present.

A 10H116 was selected here for numerous reasons. It is small (20-pin PLCC), fast (1 ns), and does not have 50-kΩ pull-down resistors built into its input structures. While these pull-down resistors (present on most ECL parts) are very handy for logic design, they have a significant impact when used for fast analog applications as done here.

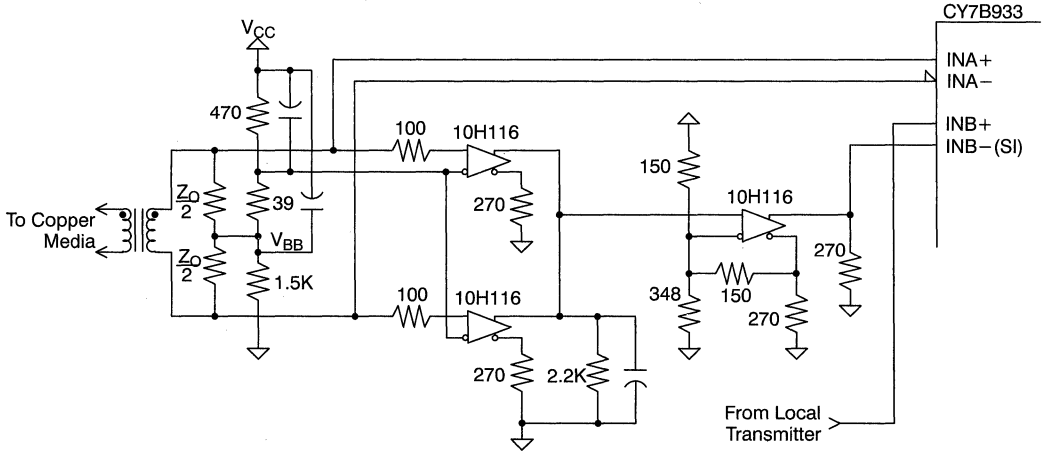


Figure 38. Copper Interface Signal Detect Circuit

Two sections of the 10H116 are used as received signal level comparators. One looks for logic-1 levels while the other look for logic-0 levels. The output of these two comparators are wire-ORed together and feed an RC network. The capacitor in this network is charged when either of the comparators is turned on, and discharges through a bleeder resistor when neither comparator is on.

The third section of the 10H116 also operates as a comparator, evaluating the voltage level on the RC network. Because the level on this capacitor changes so slowly, and ECL operates as an analog amplifier, positive feedback was added to cause the comparator to switch faster and to full ECL levels. The amount of hysteresis is set by the feedback resistor. For slow changing signals of this type, a minimum of 150 mV of hysteresis is recommended.

Copper Signal Characteristics

Communication on copper-based media is very similar to communication on optical fiber. Both suffer from increasing signal degradation with increasing media length.

The transmitted signal is composed of multiple frequency components, and requires a fairly wide bandwidth media to propagate those signal components. A large part of the bandwidth requirement is determined by the 8B/10B code and NRZ modulation used in HOTLink for communication.

NRZ Modulation

NRZ is an acronym for non-return-to-zero. This is one of the most basic types of data encoding whereby a signal is HIGH for a 1 and LOW for a 0. The upper waveform in *Figure 39* illustrates an NRZ data stream. Other forms of modulation (Manchester, Biphase, etc.) are used in data communications that encode clock information as part of the 1s and 0s. With an NRZ data stream, a phase-locked-loop is necessary to recover the bit-clock to allow data to be captured (Reference 18).

8B/10B Code Dependencies

A phase-locked-loop (PLL) requires transitions meeting specific criteria to allow it to recover a clock. If binary data were sent serially using only an NRZ modulation, long periods could exist where no transitions are sent. During these periods (if they are long enough) the receiving PLL can drift such that it is no longer able to properly recover the data sent. 8B/10B encoding is used to ensure that sufficient transitions are present in the NRZ data stream such that the receiving PLL remains synchronized to the data.

The 8B/10B code is a run-length limited code. This means that there are limits to the maximum and minimum length of a continuous sequence of 1s or 0s in the data stream. The code operates by converting an 8-bit data byte (with uncontrolled transitions) into a 10-bit transmission character (with controlled transitions). The 8B/10B code is referred to as a 1:5 code because the minimum number of consecutive 1s or 0s is one, while the maximum number is five (References 1, 2).

Translating these code limits into frequencies gives the baseband limits of the code. For example, with a serial bit-rate of 300 MHz, a pattern sent with the

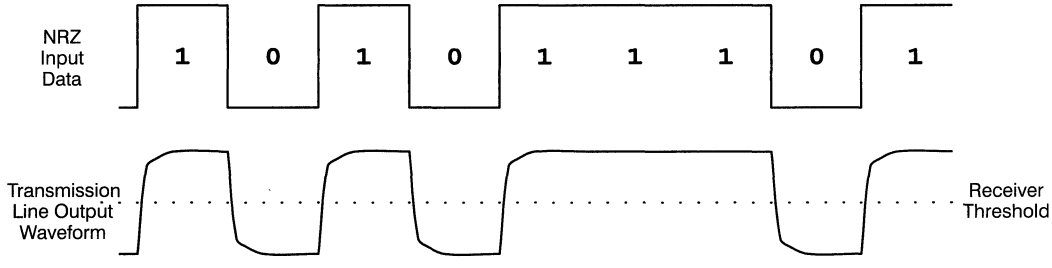


Figure 39. Short Time Constant Transmission Line Response

maximum number of consecutive 1s and 0s (five high, five low) would be equivalent to a 30-MHz square wave. Using the highest rate of alternating bits of 0 and 1 gives a frequency of 150 MHz.

As far as signal propagation goes, these numbers only refer to a sinusoidal frequency. Since square waves are used at the source, there are many additional higher-frequency harmonics present. To propagate a reasonable signal it is recommended that the system bandwidth also include at a minimum the 3rd harmonic of the highest baseband frequency, and preferably through the 5th harmonic.

For our previously described example operating at a bit-rate of 300 MHz, the necessary system bandwidth would be

$$BW = (3 \times F_{MAX}) - F_{MIN} \quad \text{Eq. 16}$$

$$BW = (3 \times 150MHz) - 30MHz = 420MHz \quad \text{Eq. 17}$$

Transmission Line Effects On Serial Data

In a perfect world a perfect square wave could be launched down a perfect transmission line and it would come out the end looking the same as it went in. Unfortunately, the laws of physics make such a transmission line impossible.

Instead, transmission lines have significant amounts of parasitic capacitance, inductance, resistance, and the terminations are reactive in nature. This means that a lossy system exists. The cable attenuation characteristics of copper cables are such that the higher frequencies have greater losses than the lower frequencies (see *Figure 77* for some sample cable attenuation curves).

When data is sent through such a lossy medium, distortion occurs. The higher frequency spectral components are significantly reduced in amplitude, while the lower frequency spectral components are reduced by a lesser amount. In addition, the higher frequency spectral components propagate faster than the lower frequency components. The square waves fed into the cable come out looking like RC charge/discharge curves.

These frequency-selective losses are equivalent to a time constant. For very short transmission lines (or very slow data rates) this time constant is short enough that transmitted 1s and 0s can completely charge or discharge the transmission line for each bit sent. The input and output signal waveforms for a transmission line of this type are illustrated in *Figure 39*.

Because the line can fully charge or discharge on even the fastest possible transition, the time to reach the receiver threshold is always the same. This allows the data out of the receiver to look just like the data sent into the transmission line.

As a transmission line is lengthened, its time constant increases. When the time constant is large enough that the line can no longer be fully charged and discharged in a single bit time, the received data edges become time displaced from their desired positions. Since coding theory refers to each transmitted 0 and 1 as a symbol, this type of distortion is called *intersymbol interference* or ISI. For communications systems, distortion of this type is called data-dependent jitter (DDJ).

Input and output waveforms for a long time constant transmission line are shown in *Figure 40*. The receiver output is added to illustrate the edge displacement. As the transmission line becomes increasingly longer it is even possible for some single-bit transitions to not be detected at all by the receiver (based on the data pattern sent) because they fail to cross the receiver threshold. This may be corrected through use of frequency compensation circuits at either the source (precompensation) or destination (equalization) ends of the transmission line.

8B/10B Code Running Disparity

The 8B/10B code attempts to limit the maximum distance (voltage) from the receiver threshold that a transmitted signal can reach, by controlling the DC signal content of the characters sent and the maximum separations between 1s and 0s used to represent each character. To do this the 8B/10B code provides two 10-bit sequences to represent each 8-bit data value. The difference between these patterns is the ratio of 1s to 0s. To determine which of the two values to send, the HOTLink Transmitter counts the number of 1s and 0s used to send each 10-bit transmission character (when operated with

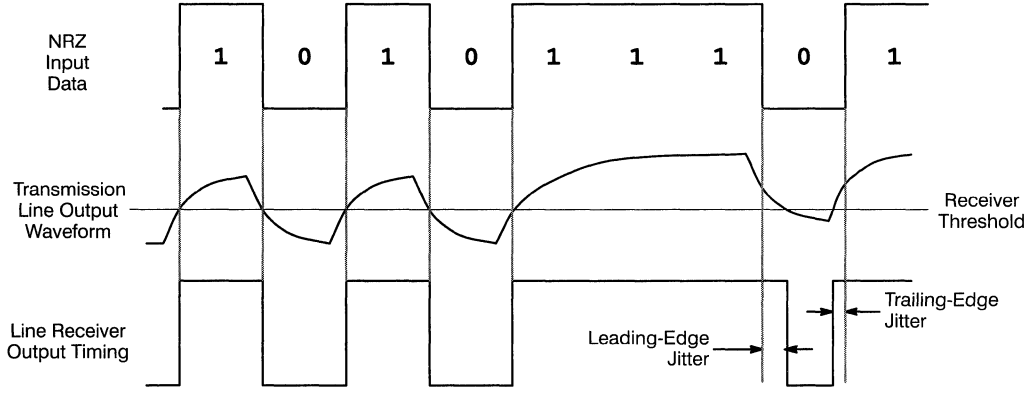


Figure 40. Long Time Constant Transmission Line Response

the encoder enabled). If the net result is more 1s than 0s (referred to as positive running disparity), the following data byte is encoded using the form with more 0s than 1s. If the net result is more 0s than 1s (referred to as negative running disparity), the following data byte is encoded using the form with more 1s than 0s. The goal of this is to maintain as near as possible a net value of DC over time for the serial data sent to minimize baseline wander.

Baseline Wander

Methods of data encoding that are not DC balanced (i.e., 4B/5B as used with FDDI) suffer from a characteristic known as baseline wander. This is a side effect of an AC coupled system attempting to propagate a signal that contains a DC component.

Baseline wander is a (relatively) long-term, low-frequency effect, generated when the average DC-level of a transmitted signal varies with the data sent. This DC component is lost because the transmission system is AC coupled. At the receiving end of the cable this appears as data that does not remain centered around the receiver threshold. This effect is illustrated in *Figure 41*.

If the receiver was actually presented with perfectly square pulses (with transitions that always crossed the receiver threshold) then baseline wander would not be a problem. Unfortunately, what are actually sent and received are more in the form of trapezoids with measurable rise and fall times. The farther a signal drifts from being centered around the receiver threshold, the more that the threshold crossings are time displaced. This time displacement is also known as jitter.

Jitter

Jitter is a high-frequency deviation from the ideal timing of an event. Many different aspects of a serial link can affect the total jitter present in the link. Those based on real and repeatable direct measurements are referred to as *deterministic jitter*. Other effects, which are not directly repeatable and are more probabilistic in nature, are called *random jitter*.

Deterministic jitter itself may be broken into two major components: those based on the accuracy of the duty cycle of the information, and those based on the interaction of the 1s and 0s due to the limited bandwidth of the transmission system. The jitter that affects adjacent edges and duty cycle is called duty cycle distortion (DCD). The jitter based on the data patterns sent is called data-dependent jitter (DDJ).

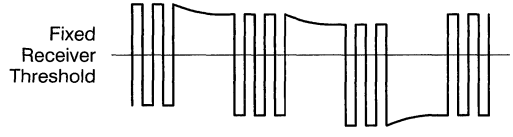


Figure 41. Baseline Wander Example

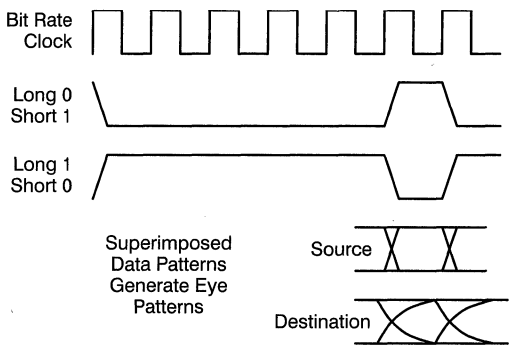


Figure 42. Eye Pattern Generation Waveforms

Data-Dependent Jitter Characteristics

Data-dependent jitter (DDJ) is a measurement of intersymbol interference based on the maximum timing deviations caused by a worst-case data pattern. DDJ is affected by many environmental characteristics, in addition to the code used. These include the length of the cable, the attenuation characteristics of the cable, the integrity of the signal launched into the cable, and how well the cable is terminated. Because of the frequency selective attenuation present in copper cables, DDJ is one of the main limiting factors on how far a recoverable signal may be sent.

To measure DDJ for a specific configuration, data patterns having specific characteristics need to be repeatedly launched into the cable. These patterns must present the worst-case transition characteristics based on the code used for sending data. This is usually described in terms of sequential combinations of long and short 0s and 1s.

A long 0 or 1 is specified as the longest continuous LOW or HIGH that can be sent. For the 8B/10B code this is five bits in length. The short 0 or 1 is the shortest LOW or HIGH that can be sent. For the 8B/10B code this is one bit in length. The sequences used for testing are diagrammed in *Figure 42*.

A design feature of the HOTLink Transmitter is that when neither data enable is active ($\overline{ENA}$ and $\overline{ENN}$ both HIGH), the part repeatedly sends out the K28.5 SYNC code. The 10-bit pattern of this code is 0011111010. Since the transmitter also tracks disparity, this pattern is inverted on every other byte. This alternating pattern contains the necessary combinations of long and short 0s and 1s for performing a proper eye pattern test.

The opening of the "eye" (see *Figure 43*) relative to the width of a bit cell is a good measure of link integrity. As this window gets smaller, it becomes more difficult for the HOTLink Receiver PLL to determine where to sample each bit cell (Reference 5).

The maximum variation, from early to late, of when the received signal crosses the receiver threshold is equal to the amount of jitter present. This jitter is usually expressed as a percentage relative to the width of a bit cell window. This relationship is shown in *Equation 18*.

$$Jitter = \frac{Bit_{TIME} - Th_{VAR}}{Bit_{TIME}} \times 100\% \qquad \text{Eq. 18}$$

The oscilloscope illustration in *Figure 44* is an actual DDJ measurement based on a 100 foot (30.4 m) segment of RG59 cable. The jitter measured in this configuration is approximately 600 ps.

Duty-Cycle Distortion Jitter Characteristics

In most cases duty-cycle distortion (DCD) is caused by the components used to make a link, rather than the data sent across the link. It manifests itself as either differences in the rise and fall times or differences in period for bits sent as a 0 compared to bits sent as a 1. This is measured by sending a pattern

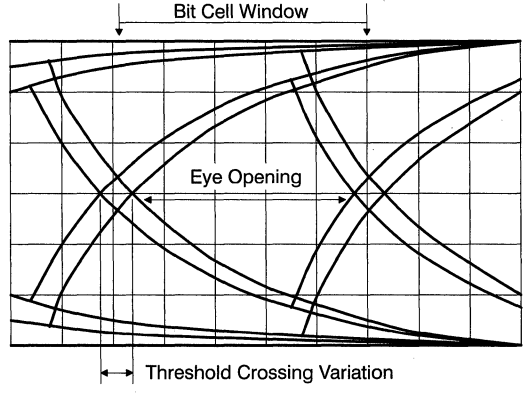


Figure 43. Eye Diagram

down a communications link that does not exhibit DDJ and using an averaging mode on the oscilloscope to filter out any random jitter (RJ) that may be present.

The HOTLink Transmitter has a built-in DCD pattern generator that is activated by placing the transmitter in BIST mode ($\overline{\text{BISTEN}}$ LOW) while both $\overline{\text{ENA}}$ and $\overline{\text{ENN}}$ remain HIGH. In this mode the transmitter sends out an alternating 1−0 pattern (D10.2 or D21.5). As all pulses in a square wave are the same, this pattern does not generate any DDJ. An example measurement of DCD for an optical link is shown in *Figure 45*.

When viewed from the receiver threshold (center horizontal line) in *Figure 45*, the timing for a logic 1 is seen to be slightly shorter than that of a logic 0. This difference in time is the DCD jitter present in the link.

Random Jitter Characteristics

Random jitter (RJ) is that portion of jitter that is not repetitive in nature and is caused by external or internal noise in a system (thermal noise, EMI, etc.). It is measured by using a data pattern free of DDJ (i.e., the same pattern used to measure DCD) relative to the transmitter clock. Now, averaging is turned off but infinite persistence is enabled. This captures the maximum variation of a transition relative to the clock. An example measurement of RJ for an optical link is illustrated in *Figure 46*.

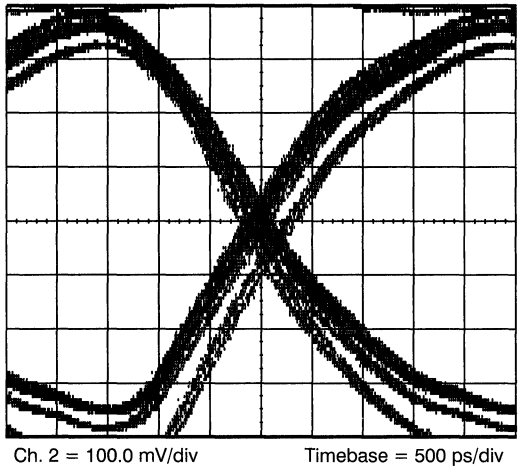


Figure 44. DDJ Measurement

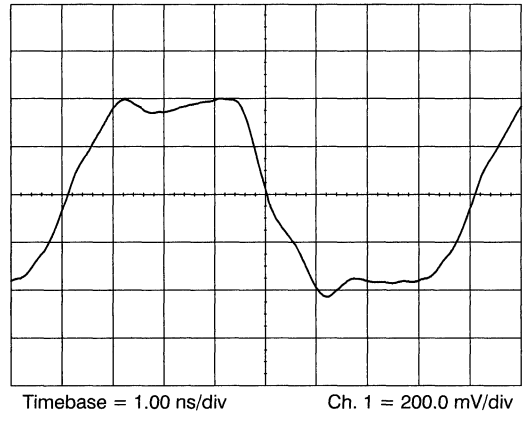


Figure 45. DCD Measurement

In this measurement the amount of random jitter present is measured by how wide the trace is as it crosses the threshold. This particular optical link has approximately 200 ps of random jitter present. This measurement was made using a 250-Mbit/second data pattern (4-ns/bit). *Equation 18* yields an RJ of 5% for this link example.

When making measurements of this kind, the tolerances of the signal sources and accuracy of the test equipment must also be taken into account. If the trigger source contains 50 ps of jitter, and the scope

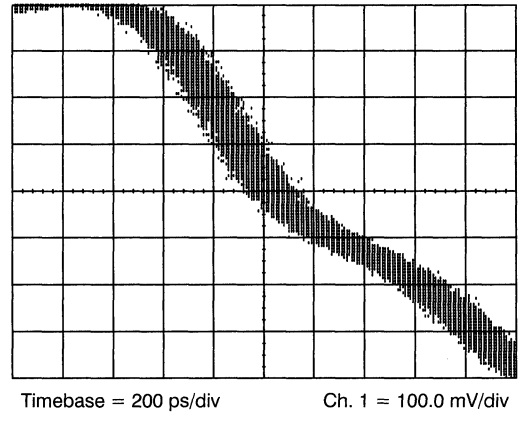


Figure 46. RJ Measurement

trigger accuracy is ±50 ps, then the actual jitter present may be substantially less than that measured.

Frequency Characteristics of 8B/10B Data

Most digital design engineers are used to viewing signals in the time domain using an oscilloscope. This instrument provides information about how a signal looks referenced to the passage of time. The waveforms in *Figure 47* illustrate the HOTLink Transmitter CKW clock on the upper trace and one of the ECL data output signals on the lower trace. The individual bit cells may be seen as the eye between the rising and falling output edges.

In the 8B/10B code, data is sent as a non-return-to-zero (NRZ) waveform. In this waveform the clocking information is contained in the edges, while the data is contained in the interval between the edges. While an oscilloscope-type display allows us to see what the output looks like in terms of voltage, rise time, period, etc., it does not present any frequency-specific information. To properly design filters, couplers, or transmission systems, it is necessary to know the frequency characteristics of the signals. This information can only be examined through use of a spectrum analyzer.

A spectrum analyzer could easily be called a frequency domain oscilloscope. A conventional spectrum analyzer operates as a swept frequency, superheterodyne receiver that displays a signal's amplitude versus its frequency. It operates by sweeping a narrow-band tuned filter across a specified section of the electromagnetic spectrum, and measuring (and displaying) the rms voltage of the signal at each frequency. This swept filter technique shows the specific frequency components that make up a complex signal, but does not provide any phase related information (Reference 22).

The spectrum analyzer output in *Figure 48* illustrates the spectral characteristics of the HOTLink Transmitter serial outputs when sending the 511-byte BIST pattern. The data patterns sent in the BIST loop are similar to those sent during normal communications traffic. This figure was made using a 30-MHz byte-rate clock (300-MHz bit-rate data). The envelope shows a relatively even distribution of power below the bit-rate of the data, and significant amounts of energy present in the information out to 1 GHz. This illustrates how necessary it is to have a true wideband transmission system to propagate the signals.

Figure 48 also shows a large dip in the energy distribution below 30 MHz. This confirms that the 8B/10B code used has no true DC component.

Figure 49 illustrates the spectral characteristics for the highest frequency data pattern that can be sent, a continuous 0101 (D21.5 character) pattern. With the 30-MHz byte-clock used here this pattern is equivalent to a 150-MHz square wave. Unlike *Figure 48*, most of the energy here is located at the fun-

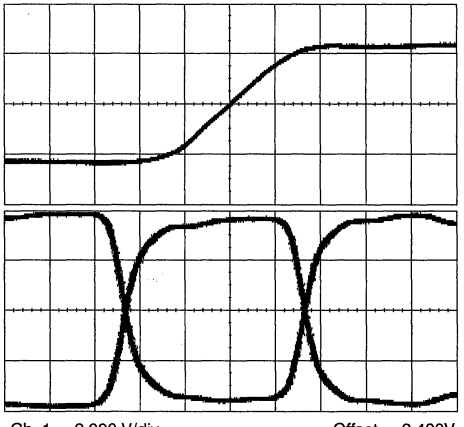


Figure 47. HOTLink Transmitter Serial Data

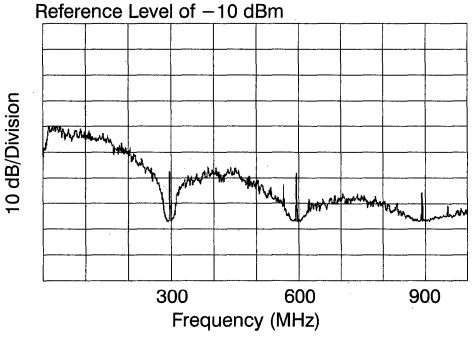


Figure 48. BIST Pattern Spectral Characteristics

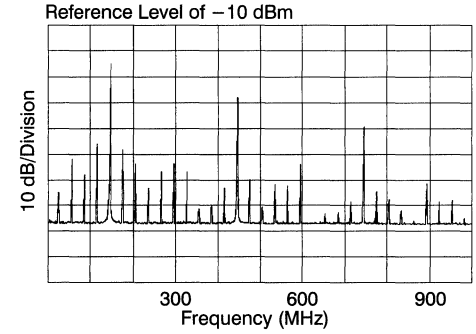


Figure 49. 0101 (D21.5) Pattern Spectral Characteristics

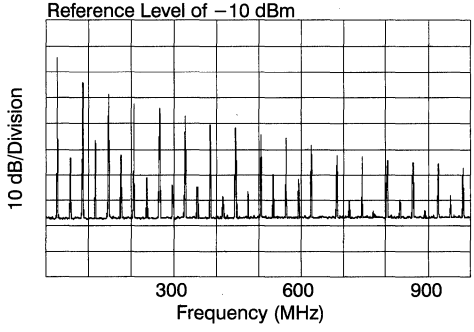


Figure 50. 0000011111 (K28.7) Pattern Spectral Characteristics

damental base frequency of 150 MHz, and at odd harmonics of that frequency. Other frequency components present in the signal are at least 30 dB down from the data being sent. These components are either generated by other parts of the HOTLink circuitry as it clocks, encodes, shifts, etc., the users data, or from external sources such as power-supply switching noise.

Figure 50 shows the spectral characteristics for the lowest legal frequency pattern that can be sent, a continuous 0000011111 (K28.7) pattern. This pattern ends up being an exact match in period to the source clock (30 MHz) with a fixed 50% duty cycle. Here, the largest amounts of energy are present at 30 MHz and all odd harmonics above that. The smaller frequency components present at the even harmonics are again due to the internal operation of the HOTLink Transmitter and external system noise. If this figure is compared to *Figure 49*, many of these even harmonic components can be seen to have almost exactly the same level in both figures.

To verify that these spectral characteristics have some resemblance to theory, these same two source waveforms were generated mathematically and analyzed using an FFT (fast Fourier transform) algorithm. This transform analyzes a source waveform and computes its frequency components.

Because the input waveforms are not true square waves, time constant curves based on a natural logarithm were used to synthesize the the rising and falling edges. These rising and falling edge equations are listed in *Equations 19* and *20* respectively.

$$T_R = 1 - e^{(-t/T)} \qquad \text{Eq. 19}$$

$$T_F = e^{(-t/T)} \qquad \text{Eq. 20}$$

In these equations, T represents the time constant for rise and fall time. For the waveforms generated here, a T of 400 ps was used. *Figure 51* illustrates the signal generated with these equations for a 150-MHz clock rate (300-Mbit/second bit-rate). This is equivalent to the data pattern generated by a D21.5 character.

Running a 4096 point FFT on this waveform yields the spectral components illustrated in *Figure 52*. The vertical axis here is plotted on a log scale to match up with the spectrum analyzer outputs. This plot illustrates that the energy of a square wave having a symmetrical rise and fall is located at the odd harmonics.

An FFT is based on numeric analysis rather than a physical measurement and will calculate signal components with an amplitude of zero. Because Log(0)

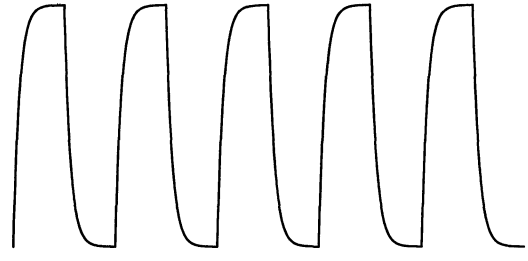


Figure 51. Synthesized D21.5 Waveform

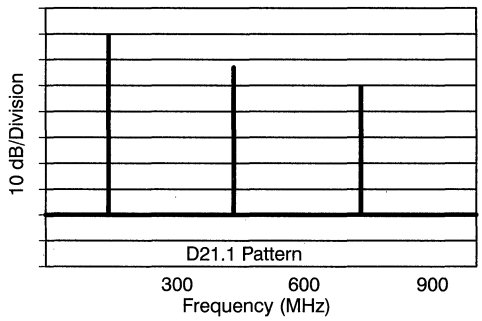


Figure 52. FFT Spectrum of Synthesized D21.5 Pattern

is equal to $-\infty$, a calculated FFT does not have a noise floor. To plot its results in a usable form requires the addition of an artificial noise floor to present the points of interest on a reasonable scale. To allow a better comparison, *Figures 52* and *54* use a noise floor similar to that measured in the spectrum analyzer charts.

Unlike a spectrum analyzer, which only displays the magnitude of the spectral components, an FFT of a waveform yields both magnitude and phase in rectangular form as a complex number. To plot this information for comparison with a spectrum analyzer plot requires conversion to polar notation of magnitude and phase. This calculation of the magnitude portion is done using *Equation 21* (Reference 24).

$$Magnitude = \sqrt{Re^2 + Im^2} \qquad \text{Eq. 21}$$

This same FFT analysis was performed on the synthesized K28.7 pattern illustrated in *Figure 53*. This waveform uses the same 400-ps time constant as *Figure 51*. The FFT based spectral plot for this waveform is illustrated in *Figure 54*. Because it uses the same value for a time constant, this waveform has the same rise and fall times as the D21.5 pattern in *Figure 51*. As with the plot for the D21.5 pattern, all of the energy is contained in the odd harmonics.

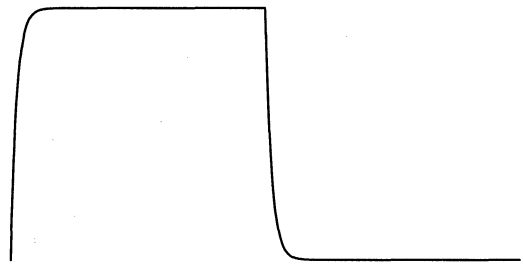


Figure 53. Synthesized K28.7 Waveform

The spectral plots for both the D21.5 and K28.7 synthesized patterns contain slightly more energy in the higher frequency harmonics than the actual measured signals. This is primarily due to the sharp knee present when the synthesized waveform changes between rising and falling. This knee is much rounder in the actual signal.

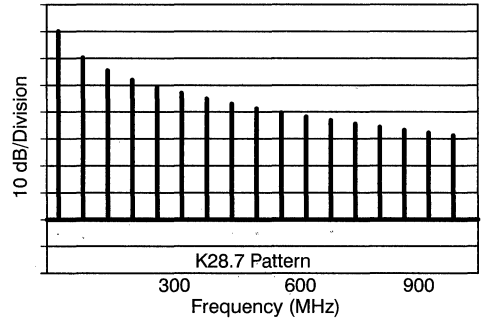


Figure 54. FFT Spectrum of Synthesized K28.7 Pattern

Components

The selection of support components for a HOTLink communications environment should not be taken lightly. The correct parts allow construction of a high-bandwidth, low error-rate system.

Several parts can be measured as key in a HOTLink system. These parts are

- Clock Oscillator
- Bypass/Coupling Capacitors
- Fiber-Optic Emitters
- Fiber-Optic Detectors
- Pulse Transformers
- Fiber-Optic Cable
- Copper Cables
- Circuit Board

Clock Oscillators

The HOTLink Transmitter and Receiver are designed to operate from a very stable clock source. To achieve the necessary frequency accuracy and stability it is necessary for this clock source to be based on a quartz crystal.

The current ANSI Fibre Channel standard calls out a frequency accuracy of ±100 ppm for both source and destination (ANSI FC-PH 4.1 Section 6.1.2 Table 8, and Section 8 Table 9) to allow reliable communications. Clock oscillators with this initial accuracy are available from multiple sources (Reference 3).

What must also be considered is lifetime stability. Most oscillator manufacturers can easily deliver product that meets the ±100-ppm rating right out of the box, but this limit must be met over the life of the product, and is affected by the operating environment. The two most critical parameters are referred to as *aging* and *temperature* stability.

Aging refers to how an oscillator's output frequency varies over time (assuming other environmental factors remain constant). This is usually expressed in ppm/year. For most common "AT" cut crystals, the typical aging is 5 ppm/year for the first year and 3 ppm/year thereafter (5 ppm=.0005%).

A crystal's resonant frequency also varies with temperature. How much it varies is based both on how the crystal is cut, and over how wide a temperature range it is used. The stability over temperature is a non-linear function and is usually expressed as some peak-to-peak frequency change over a temperature range. The process for measuring and specifying temperature stability is called out in MIL-O-55310. Temperature stability may easily exceed the initial accuracy specification. Ratings of ±100 ppm for temperature alone are not uncommon. *Figure 55* shows a typical transfer curve of crystal frequency vs. temperature.

This curve can be rotated on the +25°C axis point by cutting the crystal differently. This can be used to create an oscillator that is more stable over a narrow temperature range (say 0°C to +50°C), yet is much more unstable outside of this range.

Temperature stability and initial accuracy are often combined in a vendor's specification; i.e., ±100 ppm at 0°C to 70°C. These numbers do not take into account the aging characteristic of stability.

Modified oscillators are available that allow for a wider operating environment while maintaining a high stability. These are referred to as either TCXO (temperature compensated crystal oscillator) or OCXO (oven controlled crystal oscillator).

The TCXO is usually built by adding a varactor diode in series with the crystal. A special thermistor network across the diode causes the oscillator to maintain a very stable operating frequency. Because of the desired stability of a TCXO (±2 ppm), a better grade of crystal is used to provide better aging characteristics (±1 ppm/year). Oscillators of this type are usually larger in size (and

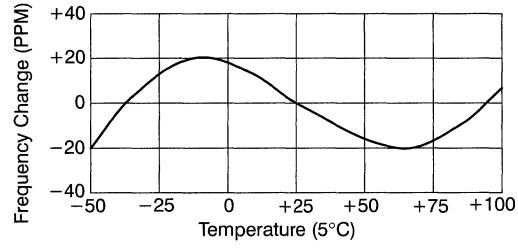


Figure 55. Oscillator Temperature Stability

higher in cost) than the standard 4/14-pin DIP footprint of standard clock oscillators.

The OCXO provides the highest-accuracy oscillators. These are built by placing a standard oscillator into a temperature-controlled environment. Rather than have to both heat and cool the crystal, the operating temperature is set to the upper end of the oscillator's range. Crystals are also cut such that a nearly flat area of temperature response is located at the operating temperature of the oven. The normal operating temperature of crystal ovens is in the 60°C to 100°C range.

Oven-controlled oscillators are generally quite large, expensive, and dissipate large amounts of power. They also have a significant warm-up period, requiring from 15 to 30 minutes after power on to achieve their specified stability (Reference 23).

HOTLink Oscillator Requirements

Unlike the ANSI requirement for ±100-ppm stability for end-to-end communication, the HOTLink family of parts will operate with a substantially wider range of reference frequencies between the HOTLink Transmitter and Receiver. The specification of 0.1% end-to-end frequency tolerance allows operation with oscillator sources operating at up to ±500-ppm tolerance. This allows even the lowest cost oscillators to be used with HOTLink.

Bypass Capacitors

At the frequencies that the HOTLink Transmitter and Receiver operate, the proper usage of power-supply bypassing becomes quite critical. Strategically sized and placed capacitors are used both to provide an AC path between V_{CC} and ground (V_{EE}), and to source current when the power supply cannot respond quickly enough due to the parasitics of the power distribution system.

The base of any power distributing system is the circuit board. Due to the very high frequencies developed in a HOTLink-based communications link, it is strongly advised to use full power and ground planes, rather than attempting to distribute power and ground on the same layers used for signal distribution. These power layers should be made with a minimum of 1-ounce copper.

To properly bypass the HOTLink Transmitter and Receiver it is necessary to know which V_{CC} pins are assigned to which portions of the logic inside the part.

HOTLink Transmitter Power Pins

The pin configuration for the HOTLink Transmitter is illustrated in *Figure 56*. The transmitter has three pins assigned as V_{CC} and two assigned as ground. All three of these V_{CC} power pins are connected internally and *must* be connected externally to the same power rail. The current flow from the slight voltage variations that would exist if different external V_{CC} supplies were used could damage the part.

Pin 4 of the HOTLink Transmitter is named V_{CCN} or Noisy V_{CC}. This pin provides power to the ECL emitter-follower output transistors. This pin is not usually a noise source if the ECL outputs are loaded in a balanced fashion. If these same outputs are operated single-ended with unbalanced loads, then a varying amount of current will flow through this pin as the outputs switch. To keep board noise to a minimum it is advised that, if an output is used, both outputs of the differential driver be loaded the same.

Pin 9 of the transmitter is named V_{CCQ} or Quiet V_{CC}. This pin provides power to the CMOS logic core of the part and the TTL compatible input buffers. This includes the 8B/10B encoder and the

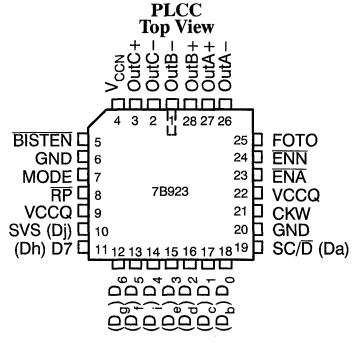


Figure 56. CY7B923 HOTLink Transmitter Pin Configuration

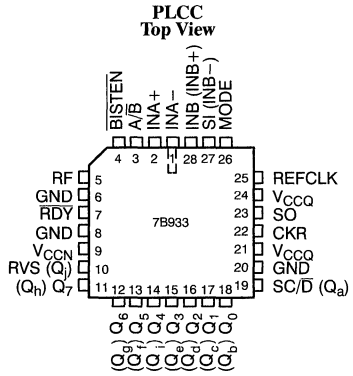


Figure 57. CY7B933 HOTLink Receiver Pin Configuration

counters and state machines used to control the flow of data through the part. Because the dynamic current draw through this pin should not be very large, the primary bypassing concern should be for higher frequency signal components present in the internal logic.

Pin 22 of the transmitter is also named V_{CCQ} or Quiet V_{CC}. This pin is probably the most critical of all the pins on the transmitter as it provides power to the analog core. This includes the charge pumps and comparators used with the PLL clock multiplier.

HOTLink Receiver Power Pins

The pin configuration for the HOTLink Receiver is illustrated in *Figure 57*. The receiver has three pins assigned as V_{CC} and three assigned as ground. All three of these power pins are connected internally and *must* be connected externally to the same power rail. The current flow from the slight voltage variations that would exist if different external V_{CC} supplies were used could damage the part.

Pin 9 of the HOTLink Receiver is named V_{CCN} or Noisy V_{CC}. This pin provides power to the TTL-compatible output buffers. Because there is no way to maintain a constant current load on these outputs (as can be done with the HOTLink Transmitter ECL outputs) there will always be significant dynamic current flow through this pin as the part operates.

Pin 21 of the receiver is named V_{CCQ} or Quiet V_{CC}. This pin provides power to the core CMOS logic in the receiver. This includes the 10B/8B decoder and the counters and state machines used to control the flow of data through the part. Because the dynamic current draw through this pin should not be very large, the primary bypassing concern should be for higher frequency signal components present in the internal logic.

Pin 24 of the receiver is also named V_{CCQ} or Quiet V_{CC}. This pin is probably the most critical of all the pins on the receiver as it provides power to the analog core. This includes the charge pumps and comparators used with the PLL and the input differential amplifiers for the high-speed serial data streams.

Bypass Capacitor Types

For the purposes of power supply bypassing, capacitors are used to store charge, and deliver that charge to a nearby device when necessary. While many still believe that charge is stored on the plates of a capacitor, it is not. Charge is stored in the dielectric (Reference 21).

There are two primary types of chip capacitors used for power supply bypassing; they are identified as either high-K or low-K capacitors. These capacitor types differ primarily in their dielectric material.

The K referred to here is the dielectric constant for the material used as a dielectric in the capacitor. High-K dielectrics for bypass-type capacitors are usually based on titanates of barium, calcium, strontium or magnesium. This material provides dielectric constants in the range of 1200 to 12,000. These high-K dielectrics allow construction of physically small capacitors that provide a large amount of capacitance per unit area. The generally available range of high-K capacitors is from 200 pF to 0.1 μF. These high-K capacitors have temperature characteristics of type X7R, Z5U, or Y5V.

Both high-K and low-K dielectrics are used for power supply bypassing. High-K dielectrics are usually not used for temperature-critical or high-frequency operations because of their thermal and frequency dependent characteristics.

One of the biggest problems with using these high-K dielectric capacitors is sensitivity to temperature. Per the graphs in *Figures 58* and *59*, these types of

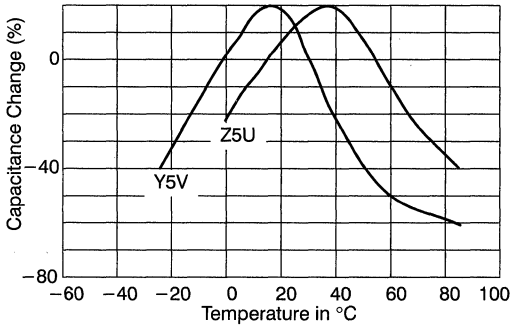


Figure 58. Capacitance vs. Temperature for Y5V and Z5U Dielectrics

parts can change their capacitance values by over 80% over the operating temperature range of most commercial or industrial applications. (The temperature characteristics for Y5V are similar to Z5U except that the peak capacitance occurs around 20°C lower in temperature.)

A second problem is that these titanate-based dielectrics exhibit ferroelectric properties; i.e., they do not respond linearly to an AC signal. The effect is similar to a hysteresis loop in magnetics. This makes these dielectrics a poor choice when a distortion-free analog response is required.

When used for high-frequency (RF) or communications-link type applications, high-K dielectrics have other drawbacks. Capacitors based

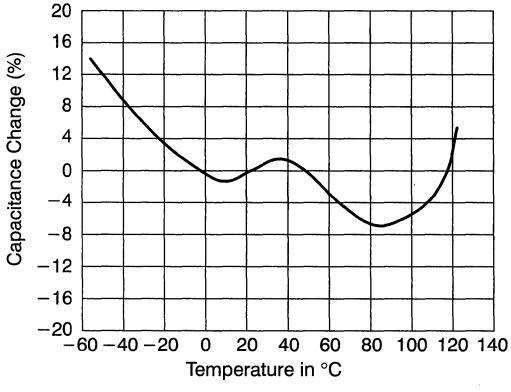


Figure 59. Capacitance vs. Temperature for X7R Dielectrics

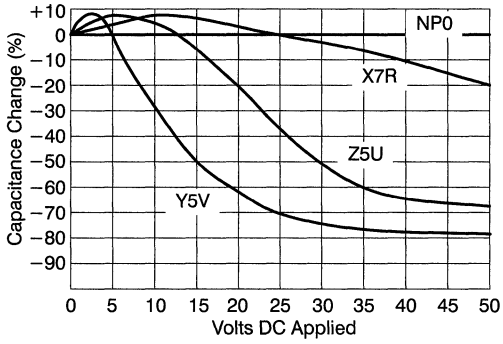


Figure 60. Capacitance vs. DC Voltage

on these dielectric types are also very sensitive to operating voltage and frequency.

Figure 60 illustrates the voltage sensitivity of high-K dielectrics. Here the capacitance loss can exceed 70% with as little as 25V applied to the part. This parameter may become critical if capacitors are used as part of a DC block in a communications link.

Figure 61 illustrates one of the effects of operating frequency on capacitance. As the operating frequency increases, the high-K dielectrics exhibit less and less capacitance. If these high-K dielectrics are to be used at an RF frequency, a capacitance correction factor must be applied to determine the actual capacitance present in the circuit (Reference 26).

Low-K dielectrics are generally based on either titanium-dioxide ceramic, alumina, or porcelain. These materials provide dielectric constants in the range of 9 to 30. Because of the low-K, these materials are only used for making small-valued capacitors

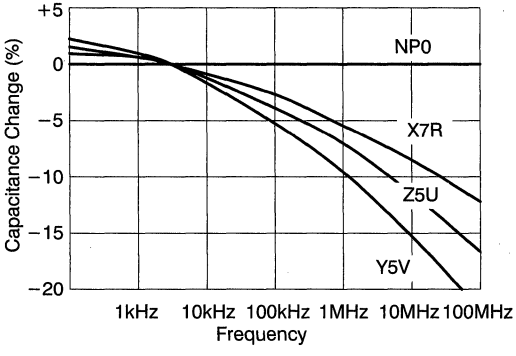


Figure 61. Capacitance vs. Frequency

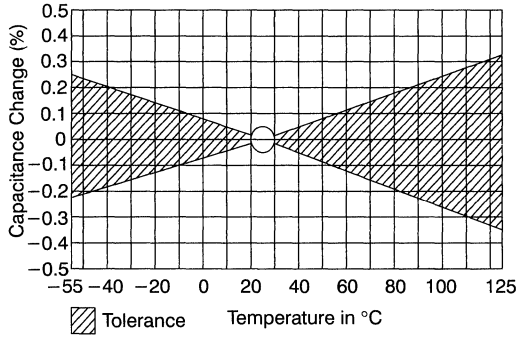


Figure 62. Capacitance vs. Temperature for NP0/C0G Dielectrics

in the range of 1 pF to 10,000 pF. These low-K capacitors are usually identified as having NP0 or C0G type temperature characteristics, and are often referred to as RF-grade capacitors because of their high-Q and low dissipation factors.

Low-K dielectric capacitors are very stable over temperature. Per *Figure 62*, these parts change in capacitance less than 0.5% over the full military temperature range of −55°C to 125°C. Because of this temperature stability, low-K capacitors are preferred for many analog applications where fixed time constants and resonant frequencies are necessary.

No capacitor provides a *pure* capacitance; i.e., there are other parasitic resistive and inductive components present in the complex impedance of a capacitor over frequency as illustrated in *Figure 63* (References 15, 16, 17, 21). These parasitic components of a capacitor are due to the materials used in, and mechanical construction of, the physical capacitor. Because of these parasitics, a capacitor cannot be treated as having ever-decreasing impedance with increasing frequency. At some frequency the capacitor passes through its series resonant point and must then be treated as an inductor.

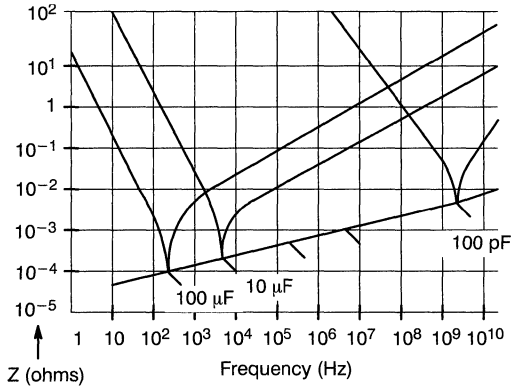


Figure 64. Capacitor Impedance vs. Frequency

A general rule of thumb is that as the capacitance decreases, the series resonant frequency increases. This relationship is illustrated in *Figure 64*. At this series resonant point, the capacitive and inductive reactance components cancel each other out, leaving only the Effective Series Resistance (ESR). For most common bypass capacitors, the ESR is well under 1Ω. When selecting parts for high-frequency operation, the smaller case sizes (0805 or 0603) are preferred because they have smaller inductive parasitics.

Resistors

Figure 65 shows a first order model of a real world resistor. Because of the parasitic L and C present, a resistor does not have a constant impedance over frequency. The actual amount of change in impedance from a pure resistance is based primarily on the construction of, materials in, and DC resistance value of the component.

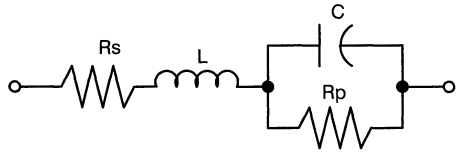


Figure 63. Capacitor Equivalent Model

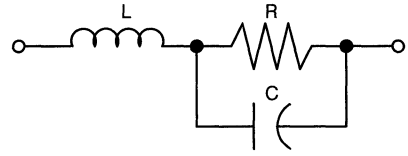


Figure 65. First Order Resistor Model

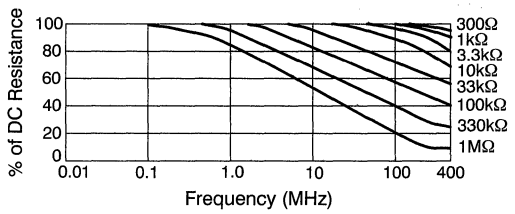


Figure 66. Carbon Film Resistor Frequency Characteristics

For high frequency or RF designs, most low-value (<1 kΩ) composite (non wire-wound) resistors may be assumed to operate at or near their DC resistance. As the DC resistance of the part increases, its impedance at higher frequencies decreases. *Figure 66* shows this relationship for typical carbon film resistors. This change in impedance is referred to as the Boella Effect and is caused by the distributed shunt capacitance present in the conducting carbon particles (Reference 21).

This shows that low-value carbon film resistors have reasonable impedance characteristics for RF applications, but for higher values a different type of resistor must be used.

For higher resistance values at RF frequencies, metal film resistors should be used. Because these types of resistors are not formed from particulate material, the distributed capacitance is reduced. These types of resistors are manufactured by vacuum sputtering of thin films of mixed metals onto a ceramic substrate. Because there are no individual particles of metal, the capacitance is much lower.

Care must also be used when selecting metal film resistors as some of these have significant inductive parasitics. These inductive parasitics are often caused by the method of laser trim used to adjust the value of the resistor. Those resistors created using two straight cuts, one from either side, are generally more inductive than those trimmed using a single straight or L shaped cut.

Metal film resistors should be used for resistors in the analog data path. This includes the transmission line termination and line bias resistors at both the source and destination ends of the serial link.

Fiber-Optic Emitters (Drivers)

A fiber-optic emitter is an electro-optical converter that changes an electrical stimulus into light. A simplified block diagram of a fiber-optic emitter is shown in *Figure 67*. The input buffer is an ECL differential line receiver. While some emitters do provide a V_{BB} output to allow single-ended operation, its use is strongly discouraged. The ECL receiver controls a high-current amplifier. The amplifier drives its current through an LED or semiconductor laser to generate a shaped optical output in response to the ECL signal input. A micro-lens assembly (usually a small sphere of glass) is used to couple and direct the light into a port for an optical fiber. Because of the small core size of the optical fiber, the lens and fiber receptacle are aligned by the fiber-optic emitter manufacturer (Reference 27).

Fiber-optic emitters are available in may different case styles, wavelengths, launch modes, data rates, etc. When selecting an emitter, the main concerns are

- Optical Receiver characteristics
- Operating data rate
- Cable plant characteristics

Most of these areas deal with interoperability of data communications links. If a shortwave laser is used as an emitter, the optical receiver must be designed to operate with the specific data rates and spectral properties of that shortwave laser. While it would be nice if a more mix-and-match combination of LED, shortwave laser, and longwave laser emitters could be used, existing receivers do not allow this. If a 1300-nm LED-driver is used, an optical receiver designed for 1300-nm LED reception must

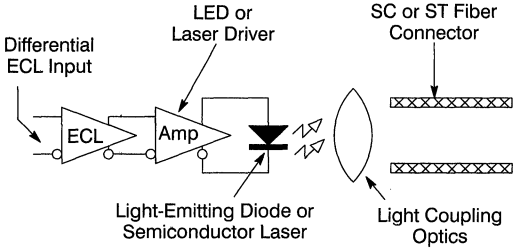


Figure 67. Fiber-Optic Emitter Module Block Diagram

be used to properly detect the signals. In addition the optical receiver must be designed to support the data rate used in the link.

Optical emitter assemblies are available from multiple sources, including AMP/Lytel, Siemens Optical, Hewlett-Packard, Sumitomo Electric, AT&T, and others.

ANSI Fibre Channel Requirements

The current ANSI Fibre Channel standard calls out four optical interface technology options for use at the 25-MByte/second data rate supported by HOTLink. The ANSI designators for these technology options are (Reference 3)

- 25−SM−LL−L
- 25−SM−LL−I
- 25−M5−SL−I
- 25−M6−LE−I

These designators are interpreted as four fields. The first field identifies the data rate used (25 MBytes/second).

The second field identifies the media used. SM specifies single-mode fiber, M5 specifies 50-μm core multimode fiber, and M6 specifies 62.5-μm core multimode fiber.

The third field identifies the transmitter type. LL specifies a 1300-nm longwave laser, SL specifies a 780-nm shortwave laser, and LE specifies a 1300-nm LED-driver.

The last field identifies the distance class of the link. L specifies long distance (2m−10 km), and I specifies intermediate distance (2m−1.5 km).

HOTLink will correctly operate with all these different link types. However, it is up to the user to select the proper combination of emitter and detector for each class.

For those users intending to implement laser-based optical links, there are a number of federal and international safety certifications required before any such link can be put into public use. These safety requirements (ANSI Z136.1 and Z136.2, F.D.A. regulation 21 CFR subchapter J, and IEC 825) are called out in the ANSI Fibre Channel standard (References 9, 10, 11, 12, 13). No such certification requirements are necessary for LED based links.

Power Distribution Requirements for Optical Drivers

The LED or laser used to drive the optical link is probably the largest noise generating item in an optical link. When the optical driver is turned on (sending 1s), currents of 50 mA to 100 mA are forced through the LED or laser. While current steering is often used to minimize dynamic current requirements, significant high-frequency noise is still generated. Most optical modules attempt to remedy part of this situation by providing multiple V_{CC} and V_{EE} pins on their package and including some power supply bypass capacitance inside the optical module. This does take care of some of the problem, but does not correct all of it.

While bypass capacitors are still necessary to provide dynamic current, additional power isolation and filtering is required to separate the high noise of the optical transmitter from the highly sensitive optical receiver, and from the serializer/deserializer operations of the HOTLink Transmitter and Receiver. Vendor's recommendations for this include a 10-μF solid Tantalum capacitor located near the optical transmitter, and a 0.1-μF decoupling capacitor directly connected to the optical transmitter V_{CC} pins (Reference 27).

Isolation is provided by separating the V_{CC} or power plane for the transmitter from the rest of the surrounding power plane, through an inductive path. This is done by placing a gap in the V_{CC} plane around most of the the transmitter V_{CC} pins with a single limited connecting point. If the transmitter package only has one or two V_{CC} pins, these may be treated individually by bringing in power through a small inductor or surface trace. For a low-noise environment this inductor may be constructed as part of the circuit board using a 15-mil-wide trace approximately 10 mm in length (approximately 5 nH). The specified bypassing should occur after this inductive trace, right next to the optical transmitter. The net result is to implement a π-filter using the circuit board and capacitors for the different filter elements. This is illustrated schematically in *Figure 68*.

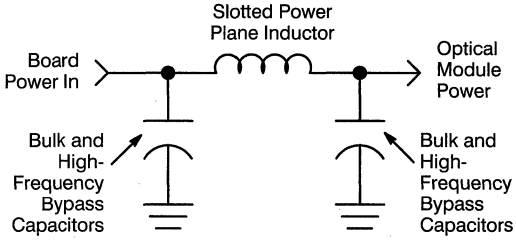


Figure 68. Optical Module Power π-Filter

An example slotted power plane used to implement the inductive element in the π-filter is shown in *Figure 69*. This illustration details an actual power plane layout for an optical module. The black areas indicate the absence of copper. The slot in the center of the figure is used to separate the power for the optical transmitter from the optical receiver. The shaded line on the right hand side indicates a surface layer trace (inductor) used to separate power for the optical module from the remainder of the design.

Fiber-Optic Detectors (Receivers)

A fiber-optic detector is an opto-electric converter that changes a light stimulus into an electrical signal. A simplified block diagram of a fiber-optic detector is illustrated in *Figure 70*. Light enters the module through an optical fiber and is guided by the connector housing. A coupling lens focuses all available optical energy onto the active region of a light sensitive diode. The presence or absence of light affects the amount of current flow through the diode. This small current flow is then amplified by a transimpedance amplifier which then feeds an ECL differential driver. Many fiber-optic detectors also contain additional circuitry such as signal-detect (Reference 27).

Fiber-optic receivers are generally available from the same vendors as fiber-optic emitters. As with fiber-optic emitters, the optical receiver must match the characteristics of the light driven into the optical fiber.

Unlike the optical emitter where there are multiple technologies used for light generation, all optical receivers are based on the response of a PIN (positive-intrinsic-negative) photodiode. These photodiodes are based on either silicon or gallium arsinide technology. The output of the PIN photodiode is a small (<1 μA) change in current in response to received light. A fiber-optic detector module feeds the output of this PIN photodiode into a transimpedance amplifier. The function of this amplifier is to convert this small change in current into a large change (ECL 100K-level) in voltage.

For many optical receivers, it is possible to operate them above their stated maximum data rate. What is given up is receiver sensitivity; i.e., many 200-Mbit/second optical modules will operate at the ANSI Fibre Channel data 266-Mbit/second data rate, but with a 3-dB or greater loss of sensitivity. This loss may be converted directly into a shorter usable distance on the fiber-optic media.

Because the optical receiver has ECL outputs, care should be taken to maintain a balanced load on any differential outputs to minimize current transients. While some optical receiver outputs (i.e., signal-detect on endfire modules) may be single-ended,

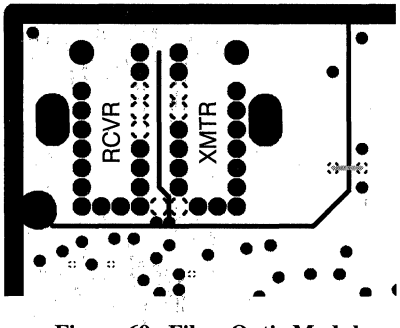


Figure 69. Fiber-Optic Module Slotted Power Plane

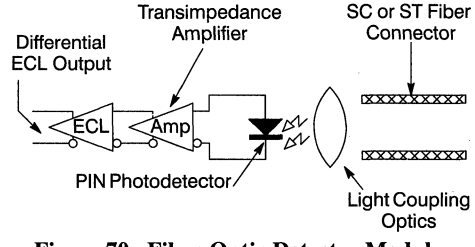


Figure 70. Fiber-Optic Detector Module Block Diagram

they usually do not change very often and should not affect data integrity when they do.

Power Distribution Requirements for Optical Receivers

The power filtering of the optical receiver is quite critical as the transimpedance amplifier must responding to very low current variations. This filtering problem is usually compounded by the placement of the high-noise generating optical transmitter, directly adjacent to the optical receiver.

Depending on the type of receiver, it may be implemented with one or many V_{CC} pins. For those made with a single V_{CC} connection, this pin should be isolated through a π-network or other network that implements an inductive leg to block RF on the power lead.

For those optical receiver modules that use multiple V_{CC} pins, these pins are usually kept separate internal to the module, and feed different sections of the logic. For those V_{CC} pins that supply power to the ECL output emitter-followers and the ECL differential amplifiers, all that is necessary is a good 0.1-µF decoupling capacitor next to the V_{CC} power pins. An inductive-based filter is recommended for the V_{CC} pin that provides power to bias the PIN photodiode and the transimpedance amplifier to limit the external noise input from the system supply.

Just as with the transmitter this inductive filter can be implemented either as a notched or slotted power plane, or by using a surface trace to act as an inductor. When implemented in this fashion the capacitor placed at the optical receiver end of the inductor should be 0.1 µF.

Optical Modules

Thanks to the efforts of a group of optical component manufacturers (AMP/Lytel, Siemens Optical, Hewlett-Packard, and Sumitomo Electric), a *de facto* standard footprint has been developed for optical modules. While originally developed for the FDDI market, optical modules with speeds suitable for Fibre Channel and ATM are also available. This footprint specifies the mechanical dimensions and signal names of two different package styles, yet allows a common board layout to accept both. The dimensions and pin numbering of this footprint are illustrated in *Figure 71*.

The two module types supported by this footprint are called DIP and endfire. The DIP modules utilize pins 1–32, while the endfire modules only use pins 33–41 (for signals) and pins 1 and 32 for package mounting. These two mounting pins are also larger in diameter than the other pins on the package.

These optical modules (DIP and endfire) share several signals. For compatibility with both module types, only the smaller set of signals present on the endfire module type should be used. A complete listing of the signals present in the standard footprint is found in *Table 5*. The signals present on the optical module are

- SD — Signal Detect
- TD — Transmit Data
- RD — Receive Data
- Case — Outer Case of Module
- V_{CC} — Positive Supply Voltage
- V_{EE} — Negative Supply Voltage
- V_{BB} — ECL Base Threshold Voltage

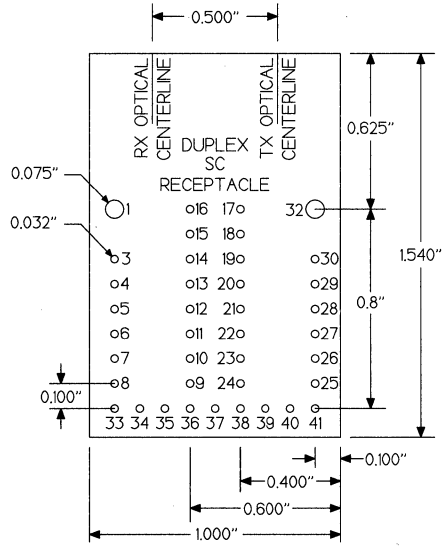


Figure 71. Standard Optical Module Footprint

The V_{BB} and SD− signals are only present on the DIP footprint package and thus should not be used in designs that wish to support interchangeable module types.

Table 5. Standard Optical Module Pinout

DIP Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
1	Case	2	No Pin
3	Case	4	V_{EE}
5	V_{EE}	6	+SD
7	−SD	8	Case
9	Case	10	RD
11	+RD	12	V_{CC}
13	V_{CC}	14	V_{CC}
15	Case	16	Case
17	Case	18	Case
19	V_{CC}	20	V_{CC}
21	Case	22	+TD
23	−TD	24	Case
25	Case	26	V_{BB}
27	Case	28	Case
29	V_{EE}	30	V_{EE}
31	No Pin	32	Case
Endfire Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
33	V_{EE}	34	+RD
35	−RD	36	+SD
37	V_{CC}	38	V_{CC}
39	−TD	40	+TD
41	V_{EE}		

Care must be used when connecting to the pins marked as Case. These pins are not specified as being isolated, tied to V_{EE}, or tied to V_{CC}. As such, each manufacturer is allowed to connect them as they wish.

Isolated Case pins may be connected either to V_{CC} or V_{EE}. Usually this connection is made to whichever power rail is identified as ground in the system. When used with the HOTLink Transmitter, these types of modules are usually operated in PECL mode with the Case pins connected to V_{EE}.

When the case is connected to the V_{CC} pins, the part is designed for operation in a standard ECL (negative-referenced) system. Modules of this type may still be used with HOTLink, but some care must be taken in how they are interfaced.

Pulse Transformers

A pulse transformer is a magnetic device used to couple electrical energy from one stage to another with minimal distortion. This coupling occurs through magnetic induction. How well this coupling occurs is based on the construction of the transformer and the materials used for the core and windings.

Core Materials

There are three basic types of core materials used for transformers: metal, powdered iron, and ferrites. Metal cores consist of pieces of low conductivity metal having some magnetic properties; usually soft iron or steel. This metal core is usually made from multiple strips or laminations of material to limit eddy currents in the core. Metal cores have a practical upper frequency limit of about 50 kHz.

Powdered iron cores use metal powder fused together by an insulating binder. Because of the smaller size of the magnetic particles, the upper frequency for powdered iron cores extends to near 1 MHz.

Ferrites are a magnetic form of ceramic. Depending on the type of ferrite and construction of the core, transformers with ferrite-based cores are available with operating frequencies of near 1 GHz. This is the core material that must be used for transformers used with HOTLink.

ANSI Fibre Channel Specifications

The current ANSI Fibre Channel standard, section 7.1, states that the recommended interface to all types of copper media is via transformer coupling. The primary benefits of transformer coupling are ground isolation, common-mode rejection, and the ability to drive both balanced and unbalanced transmission lines with the same interface (Reference 3).

Just as with optical interfaces, the ANSI standard calls out multiple copper technology options for use

at the 25-MByte/second data rate supported by HOTLink. The ANSI designators for these technology options are

- 25-TV-EL-S
- 25-MI-EL-S
- 25-TP-EL-S

These designators are interpreted as four fields. The first field identifies the data rate used (25-MBytes/second).

The second field identifies the media used. TV specifies 75Ω video grade coaxial cable, MI specifies a 75Ω miniature coaxial cable, and TP specifies shielded twisted-pair.

The third field identifies the transmitter type. The EL identifier is used for all electrical classes.

The last field identifies the distance class of the link. S specifies short distances (<75m).

While these are the only electrical classes that ANSI supports for Fibre Channel, many other impedances and distances will function with the HOTLink Transmitter and Receiver.

The typical transformer electrical characteristics to support these interface combinations are called out in the ANSI Fibre Channel standard in Section 7.1, Table 10 (Reference 3).

Pulse transformers suitable for coupling HOTLink to copper based cables are available from Pulse Engineering, Mini-Circuits, Premier Magnetics Inc., Valor, and others.

Fiber-Optic Cables

Optical media generally falls into two categories: multimode and single-mode. The usage of each type is dictated by the spectral characteristics and launch mode of the light into the fiber.

Single-Mode Fiber

Single-mode fiber is most often used with optical drivers that are both spectrally pure (i.e., a laser) and coherent in their output (well collimated, long-wave laser). Fibers of this type have a very small core section to limit the modes of propagation of the transmitted light, and an index of refraction designed to only allow light to remain in the core that strikes the cladding at a very low critical angle. Its main propagation of light is by refraction (bending) of light that travels down the center of the core. In addition, a small number of tight turns of the fiber are usually placed near the optical transmitter to act as a filter for any of the higher-order modes of propagation that may be launched into the fiber. These turns change the incidence angle of the higher-order modes between the core and the cladding of the fiber, causing light at these modes to leave the core. A diagram of a single-mode fiber is shown in *Figure 72* (Reference 18).

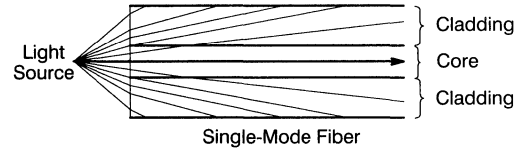


Figure 72. Single-Mode Fiber Propagation

Single-mode fibers are available in different core diameters for use with different optical sources. The fiber type called out for single-mode propagation in the ANSI Fibre Channel standard is 125-μm fiber diameter with a 9-μm core. With this core diameter, the fiber is limited to use with 1300-nm sources (Reference 3).

Multimode Fiber

Multimode fiber is usually used with optical drivers that that are not spectrally pure (i.e., LED) or not coherent in their output (i.e., shortwave lasers). The lensing system used to couple the optical driver's light output to the fiber is not designed for collimation, but to couple the maximum amount of light. This type of fiber allows propagation of light both by refraction and by reflection.

Two distinct classes of multimode fiber are in use today: step-index and graded-index. In a step-index fiber, the primary mode of light propagation is through total internal reflection. Light that enters the core on one end is continuously reflected at the core/cladding interface until it exits the cable at the other end. A diagram of multimode step-index fiber is shown in *Figure 73*.

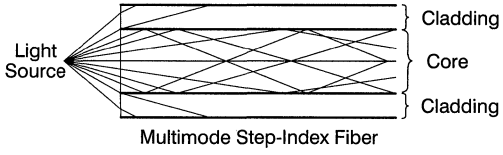


Figure 73. Multimode Step-Index Fiber Propagation

In a graded-index fiber, light is propagated through refraction rather than reflection. The fiber core is constructed of multiple concentric layers of glass. The index of refraction in each layer is slightly different, getting lower as you move out from the center of the core. Because light travels faster in a lower index of refraction, the higher-order modes of propagation that travel the farthest arrive in phase with the low-order modes that remain near the center of the core. A diagram of a multimode graded-index fiber is shown in *Figure 74* (Reference 18).

The step-index form of multimode fiber is not normally used for data communications because its propagation characteristics limit the usable distance of a link. The ANSI Fibre Channel standard currently only supports graded-index fibers with core diameters of 50 µm or 62.5 µm, both with a cladding diameter of 125 µm (Reference 3).

Optical Pulse Dispersion

In a step-index fiber, light that travels straight through the core covers a shorter distance and arrives at the end of the fiber before light that repeatedly bounces off the core/cladding interface. This difference in delay through the fiber causes a narrow pulse launched into the fiber to widen as it travels down the fiber. Because this pulse widening or dispersion is caused by the different modes of propagation, this phenomena is known as modal dispersion.

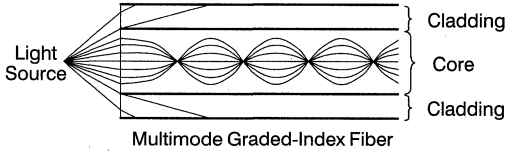


Figure 74. Multimode Graded-Index Fiber Propagation

When used with an LED driver, an additional source of dispersion comes into play. Unlike free space where all wavelengths of light propagate at the same rate, an optical fiber propagates different wavelengths at different rates. This causes any light pulse that is not spectrally pure (i.e., all the same wavelength) to widen as it travels down the fiber. Pulse widening caused by wavelength is called chromatic dispersion.

With multimode fiber one of the main limits to usable distance is the pulse spreading caused by light dispersion within the fiber. As the transmitted 1s (pulses of light) get wider through dispersion, they interact with adjacent transmitted 0s (absence of light). The effect of dispersion is illustrated in *Figure 75* (Reference 18).

With single-mode fiber, dispersion is usually not a limiting factor. Here the amount of attenuation over distance is the main limiting factor.

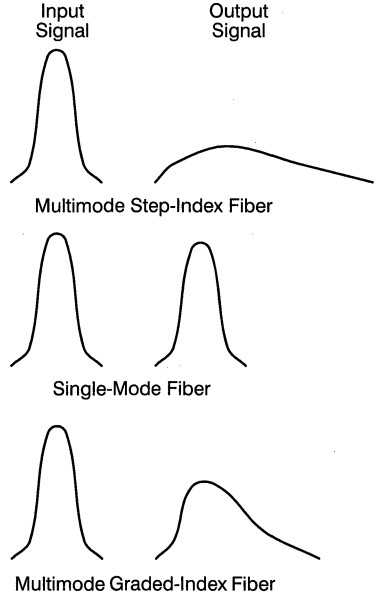


Figure 75. Pulse Dispersion

ANSI Fibre Channel Optical Fibre Requirements

Fiber-optic cables are available with many different optical and mechanical characteristics. International organizations have set standards for optical cable plants to allow manufactures to standardize on some cable types.

The standards body that created the standards used for optical cable plants is called EIA/TIA (Electronic Industry Association/Telecommunications Industry Association). The governing document for all optical fiber types is EIA/TIA 492BAAA. This includes single-mode and both core diameters of multimode fiber.

The ANSI Fibre Channel standard has also selected a common fiber-optic connector type for use with all types of optical fiber media. This connector type was developed by NTT in Japan and is known as an SC-type optical fiber connector. A diagram of a simplex SC connector is shown in *Figure 76*.

These simplex connectors may be joined together using a plastic clip to form a duplex connector. In the duplex configuration the center-line spacing of the optical fibers is 0.5 inch.

Simplex and duplex cable assemblies are available from AMP, FOCS Inc., Alcoa Fujikura Ltd., Belden, and many others.

Copper Cables

There are three primary types of copper media available for distance data transmission: shielded twisted-pair (STP), twinaxial cable, and coaxial cable. Each of these cable types has specific advantages and characteristics.

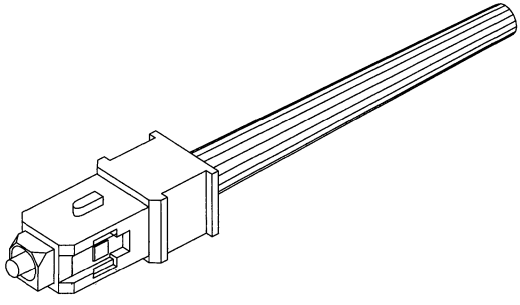


Figure 76. SC Simplex Fiber-Optic Connector

Shielded Twisted Pair

Shielded twisted-pair (STP) cables are used for many low-cost LAN installations. One of the most common of these is the IBM Type-1 and Type-6 cables used for IEEE 802.5 token ring networks. For use with the ANSI Fibre Channel, the standard calls out Type-1 and Type-2 150Ω STP cables as defined in EIA/TIA 568 (References 3, 11, 20).

STP cables are constructed of two insulated conductors twisted together at a specific number of twists per foot, with an overall shield and jacket. They are available with characteristic impedances of from 78Ω to 200Ω. With this type of cable the transmission remains fully differential from source to destination. The shield is only used to prevent radiation and control susceptibility. Cables of this type are effective for long distances at low data rates, and short distances for high data rates. The main limiting factor for cables of this type is their attenuation at high frequencies. In many cases, cables of this type are so poor above 50 MHz that attenuation is not even specified at these frequencies. In some vendors' data, shielded twisted-pair cables are also referred to as twinax (Reference 20).

Twinaxial Cable

Twinaxial cable is a shielded form of twin-lead. Twinaxial cables consist of two parallel insulated conductors, maintained at a fixed spacing with an overall shield. Cables of this construction are often used for television reception lead-in cable. As with STP cables, twinaxial cables maintain a fully differential transmission system from transmitter to receiver. Twinaxial cables can have lower attenuation of high frequency signals than STP cables and can be used for longer distances.

Unshielded twin-lead, while having excellent high-frequency characteristics, is not generally usable for data communications due to the radiated emissions of the cable, and the impedance changes that occur as the unshielded cable is routed near metallic objects.

Twinax cables are available in impedances from 125Ω to 300Ω and velocities of 70% to 80% (Reference 20).

Coaxial Cable

Coaxial cable is used for the longest distances. They consist of a single center conductor surrounded by a dielectric spacer, surrounded by a concentric shield. Unlike either STP or twinax, coaxial cables are an unbalanced transmission line; i.e., the signal is transmitted and received as a signal relative to a ground or shield, rather than a signal relative to another signal.

In a coaxial cable the outer conductor acts both as part of the transmission line to propagate the signal, and as a shield to prevent radiation of the transmitted signal and susceptibility from outside signals.

Coaxial cables are available in impedances from 50Ω to 125Ω and velocities of 66% to 90%. The main element that affects the velocity of propagation is the dielectric type used between the center conductor and the shield. Solid polyethylene is a common dielectric at the 66% velocity. The fastest speeds usually resort to foamed Teflon or partial air core. *Table 6* lists some common coaxial cable types and characteristics (Reference 20).

One thing that cannot be seen from this table are the cable's attenuation characteristics versus frequency. This is one of the characteristics that determines just how far a usable signal can be sent. The cables listed in *Table 6* are plotted for attenuation in *Figure 77*.

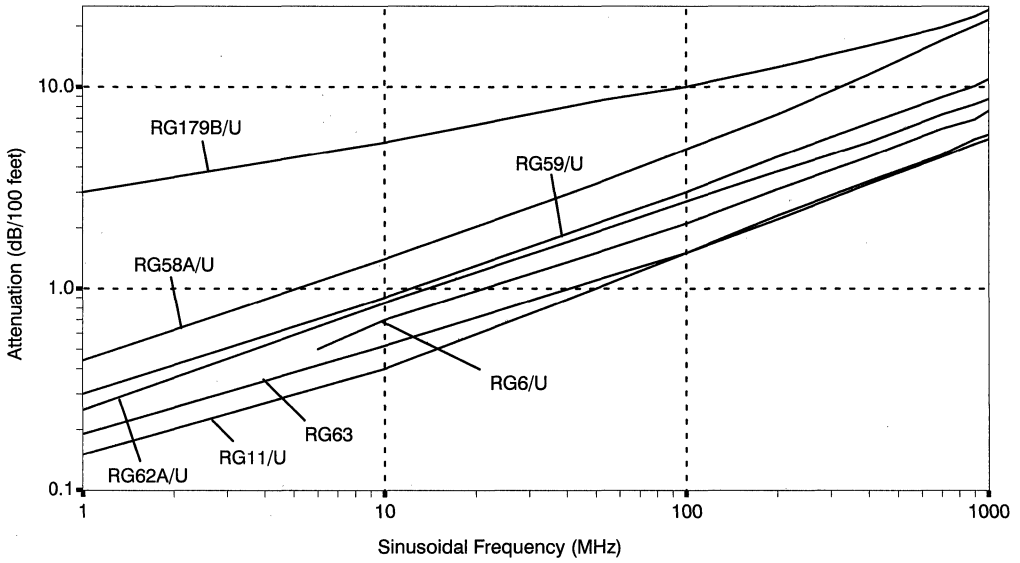


Figure 77. Coaxial Cable Attenuation Characteristics

Table 6. Common Coaxial Cable Types

RG/U Type	Belden Type	Z_O	Nominal O.D.	V_P
RG58A/U	8259	50	.193″	66%
RG179B/U	83264	75	.1″	70%
RG6/U	1223A	75	.290″	83%
RG59/U	9259	75	.242″	78%
RG11/U	87292	75	.348″	82%
RG62A/U	9268	93	.242″	84%
RG63	9857	125	.405″	84%

ANSI Fibre Channel Copper Cable Requirements

The ANSI cable plant requires copper cables with specific operating characteristics. These characteristics are called out in Section 9 and Annex F of the Fibre Channel PC-PH standard (Reference 3).

Realizing these requirements means that the cable must be made with specific construction. For coaxial cables the V_P of 70% to 82% requires a foam dielectric.

The minimum necessary shield coverage for braid is 95%. This is necessary because of the high frequencies carried by the cables. With shield coverage lower than this, the signal leakage through the braid can allow not only significant signal radiation, but an impedance mismatch due to signal propagation down the outer surface of the braid. For best effectiveness, a 100% foil shield should be used in addition to the braid shield.

To meet flammability requirements, the National Electrical Code now requires that almost all installations use either CL2 or CL2P (plenum rated) jacket material (Reference 25).

Cables meeting all of these requirements are available from multiple vendors.

The ANSI standard also allows use of shielded twisted pair or twinaxial type cables. These cables all require a shield to meet EMI/EMC requirements. Unshielded twisted pair (used for many networks) should not be used. This is primarily due to radiated emissions rather than susceptibility.

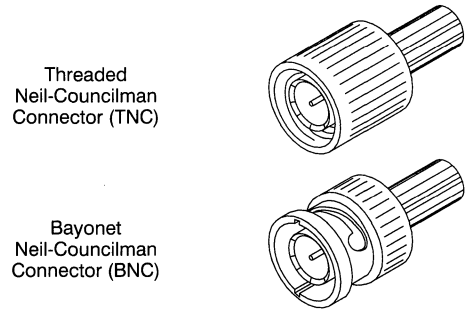


Figure 78. TNC/BNC Cable Connectors

Copper Cable Connectors

There are three primary connector types called out for use with copper cables: BNC and TNC for coaxial cables (illustrated in *Figure 78*) and a 9-pin D-sub (illustrated in *Figure 79*) for twisted-pair/twinax cables.

For coaxial cables, the BNC connectors are used on the transmitting end of the cable while the TNC connectors are used on the receiver end of the cable. This dual connector configuration allows a duplex cable to be connected without having to identify one cable from the other. With these connectors the male end is always on the cable while the female end is used at the board bulkhead.

For twisted-pair or twinaxial type cables a 9-pin D-sub connector is used. This connector is required to have a metal shell because the shields of both the transmit and receive pairs are terminated to the shell of the connector. As with the coaxial connec-

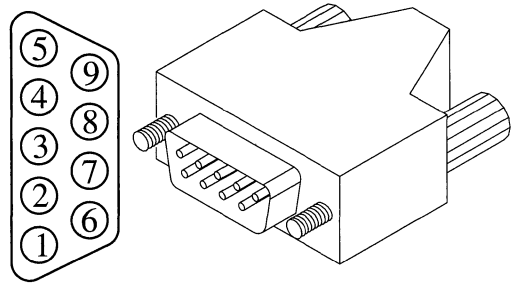


Figure 79. STP Cable Connector and Connector Pinout

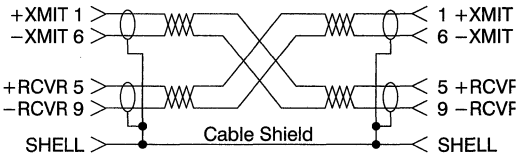


Figure 80. STP Cable Connections

tors the cable gets the male connector while the board or bulkhead gets the female connector.

The STP cable is wired in a crossover fashion where the transmit pins at one end of the cable (as illustrated in *Figure 80*) are connected to the receive pins at the other end of the cable. The cable shields for both pairs are tied together and connected to the D-sub connector shell at each end.

Because of the low current used in these cables, the connections are considered to be dry circuits. To prevent contact oxidation from degrading the link over time the contacts are required to be gold or palladium plated (Reference 28).

Conclusion

The HOTLink family of communications products provide designers with a simple yet elegant method of reliably moving large quantities of data at very high speeds from one place to another. These parts are capable of communicating over copper or optical media at distances well in excess of industry standards. Their BiCMOS implementation, along with their integrated power saving features, combine to offer one of the lowest-power, high-speed serial communications link standards available.

References

1. A. X. Widmer and P. A. Franazek, *A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code*, IBM Journal of Research and Development, 27, No. 5: 440-451, September 1983
2. U.S. Patent 4,486,739, Peter A. Franaszek and Albert X Widmer, *Byte Oriented DC Balanced (0,4) 8B/10B Partitioned Block Transmission Code*, December 4, 1984
3. *Fibre Channel Physical Standard, ANS X3.230-1994*, American National Standards Institiute, 1994
4. *Enterprise System Architecture/390 ESCON I/O*, SA22-7202, IBM Corporation, 1990
5. *F100K ECL Logic Databook and Design Guide*, National Semiconductor, 1990
6. Lawrence B. Levit and Marco L. Vincelli, *Characterize High-speed Digital Circuits: A Job For Wideband Scopes*, Lecroy Corp., EDN June 10, 1993
7. *The ABC's of Probes*, Tektronix Pub 60W-6053-3
8. *Product Overview*, Cascade Microtech, Product Overview, 1992
9. *Safe Use of Lasers ANS Z136.1-1993*, American National Standards Institute, 1993
10. *Laser Safety in Optical Communication Systems ANS Z136.2*, American National Standards Institute
11. *Commercial Building Telecommunications Wiring Standard EIA/TIA-568*, Electronics Industries Association/Telecommunications Industries Association
12. F.D.A. Regulation 21, Code of Federal Regulations
13. *IEC825*, International Electrotechnical Committee
14. Scott, Paul, Cypress Semiconductor, *Draft Paper — HOTLink On Wire*, May 1992
15. *1990–91 Resistor/Capacitor Data Book*, Philips Components, 1990
16. *System Design Considerations When Using Cypress CMOS Circuits*, Cypress Semiconductor Applications Handbook, 1993
17. White, Donald R.J., *Electrical Filters, Synthesis, Design and Applications*, Second Edition, 1980
18. Sterling, Donald J.Jr., *Technician's Guide To Fiber Optics*, Second Edition, 1993
19. Blood Jr., William R., *MECL System Design HandBook*, Fourth Edition, 1988
20. *Belden Wire and Cable*, Cooper Industries, 1990
21. Botos, Bob, Hewlett Packard, *Designers Guide to RCL Measurements*, June 1979
22. *Hewlett-Packard Test and Measurement Catalog*, Hewlett-Packard Corp., 1992
23. *Crystal Oscillator Handbook and Catalog*, Vectron Laboratories, Inc., 1992
24. Ramierez, Robert W., *The FFT, Fundamentals and Concepts*, Tektronix, Inc. 1985
25. *National Electrical Code*, National Fire Protection Association
26. *1990–91 Resistor/Capacitor Data Book*, Philips Components, 1990
27. *Application Note 65074, Fiber-Optic Transmitter and Receiver*, AMP Incorporated, 1992
28. J. H. Whitley, AMP Inc., *Contacts and Dry Circuits*, AMP Symposium paper, October 1963

Serializing High Speed Parallel Buses to Extend Their Operational Length

Introduction

Parallel buses are used in many designs for the purpose of moving data from one point to another. VME, ISA, EISA, VESA, PCI, SBus, and NuBus are some of the more familiar bus architectures. These buses are usually configured with a single bus master and multiple users, all communicating over a shared set of address and data lines. Some bus architectures, however, involve only two nodes on the bus, creating a point-to-point data link. Regardless of the architecture, the trend in bus design is for higher bandwidth achieved by increasing the width and transfer rate of the bus. When wide, high-speed, parallel buses are operated over distances of more than a couple of feet, problems can result. The source of these problems relates to the high-frequency signals interfering with each other over the long parallel conductors of the bus. This application note uses the UTOPIA bus as an example of how to serialize a high speed parallel point-to-point bus in order to allow the bus to operate over any distance.

The topics covered in this application note are as follows:

1. The UTOPIA Bus
2. UTOPIA Applications
3. Problems with Parallel Buses
4. The Serial Solution
5. Serial Links and HOTLink™
6. Serializing the UTOPIA Bus
7. Round Trip Latency
8. The UTOPIA Extender
9. Conclusions

The UTOPIA Bus

A good example of a high speed point-to-point parallel bus is the Universal Test and Operations Physical Interface for ATM (or UTOPIA). UTOPIA is used in ATM (or Asynchronous Transfer Mode) applications. ATM is a network protocol that has grown out of the need for a worldwide standard to allow interoperability of information, regardless of the "end-system" or type of information. With ATM, the goal is one international standard.

ATM is a method of communication which can be used as the basis for both LAN and WAN technologies. When information needs to be communicated, the sender negotiates a "requested path" with the network for a connection to the destination. When setting up this connection, the sender specifies the type, speed, and other attributes of the call, which determine the quality of service. Thus ATM is a switch-based technology (see *Figure 1*). By providing connectivity through a switch (instead of a shared bus) ATM delivers several benefits including dedicated bandwidth per connection, higher aggregate bandwidth, well-defined connection procedures, and flexible access speeds.

Using ATM, information to be sent is segmented into a fixed-length cell, transported to and reassembled at the destination. The ATM cell has a fixed length of 53 bytes. Being fixed-length allows different traffic types on the same network. The cell itself is broken into two main sections, the header and the payload. The payload (48 bytes) is the portion that

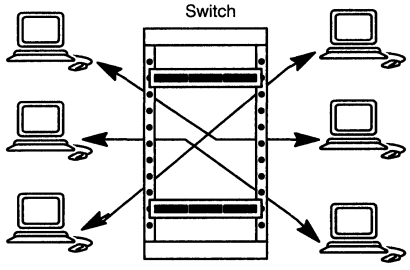


Figure 1. ATM Connections Through Switch

carries the actual information—either voice, data, or video. The Header (5 bytes) is the addressing mechanism (see *Figure 2*).

ATM closely follows the International Standards Organization's (ISO) Open Systems Interconnection (OSI) model for communication. This model breaks down any communication process into several sub processes arranged in a stack (see *Figure 3*).

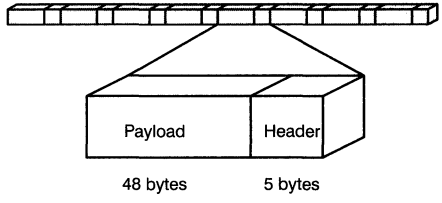


Figure 2. ATM Cell Format

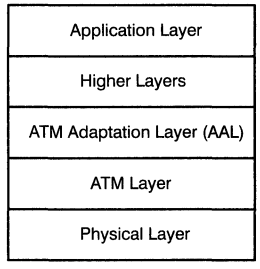


Figure 3. ATM Protocol Stack

Each layer of the "protocol stack" provides services to the layer above that allow the top most processes to communicate. The idea is that two different devices, using hardware and software from different vendors, but still conforming to the model, can communicate over an ATM network. The layers of the protocol stack can be thought of as modules in software code. Each layer performs a specific function and must provide data to other layers according to a specified interface. However, how that layer accomplishes its task is immaterial. Thus, layers in the stack can be updated without affecting the communication model.

The UTOPIA bus is a standard defined by the ATM forum for moving data between the physical (or PHY) and Asynchronous Transfer Mode (or ATM) layers in the ATM protocol stack. The PHY layer interfaces directly to the network media (i.e., fiber, twisted pair, etc.) and also handles "transmission convergence" (that is, extracting the ATM cells from the transport coding scheme). The ATM layer processes the cell headers and directs routing. The signals used by the UTOPIA bus are shown in *Figure 2* and described in *Table 1*.

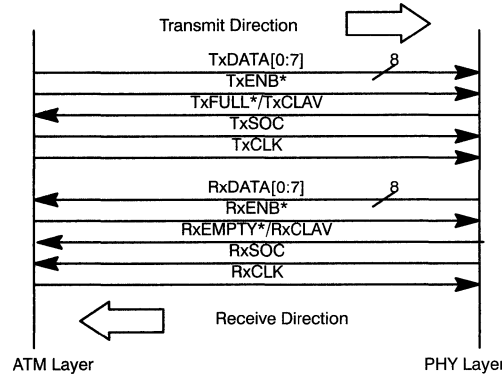


Figure 4. UTOPIA Signals

Table 1. UTOPIA Signals

Signal Name	Description
TxDATA[0:7]	Data lines for transmit (from ATM to PHY layer)
TxENB*	Indicates data on this cycle is valid
TxFULL*	Indicates Tx FIFO on PHY layer can only accept 4 more bytes (used only in Octet Level Handshaking)
TxCLAV	Indicates Tx FIFO on PHY layer is capable of storing an entire cell
TxSOC	Indicates data on this clock cycle is the start of a cell
TxCLK	Clock for Tx signals and data
RxDATA[0:7]	Data lines for receive (from PHY to ATM layer)
RxENB*	Indicates data on this cycle is valid
RxEMPTY*	Indicates Rx FIFO on PHY layer is empty (used only in Octet Level Handshaking)
RxCLAV	Indicates Rx FIFO on PHY layer is currently storing an entire cell
RxSOC	Indicates data on this clock cycle is the start of a cell
RxCLK	Clock for Rx signals and data

UTOPIA Applications

The UTOPIA bus is present in any ATM system that makes use of the ATM and PHY layers. Typical applications utilizing UTOPIA include Network Interface Cards and ATM switches. The ATM switch application for UTOPIA is of particular interest. Many switches are built using a rack mounted architecture as shown in *Figure 5*.

In this type of switch, individual shelves of the rack are dedicated to PHY layer circuits, and others to ATM layer circuits. Thus the UTOPIA bus is used to move the data between the different shelves of the switch. Usually, the interconnect between the

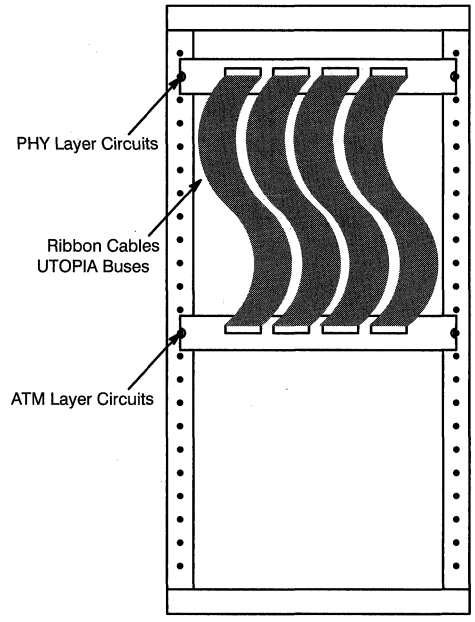


Figure 5. UTOPIA in a Rack Mount Switch

shelves is a simple multi-conductor ribbon cable. Since the shelves can be fairly far apart, the ribbon cable required to connect the shelves can be anywhere from 1 to 6 feet in length.

Problems with Parallel Buses

The difficulty with the use of ribbon cable for the UTOPIA switch application is related to the width and bandwidth requirements of the bus, combined with the uncontrolled impedance of the ribbon cable. These three characteristics can lead to skew across the signals of the UTOPIA bus as shown in *Figure 6*.

Note the skew shown in *Figure 6* has violated the set-up and/or hold times of the UTOPIA bus at the load end. Therefore, data communication over the bus will be corrupted. This effect is typical when high-speed parallel buses are driven over long distances. One possible solution is to drive each line of the bus differentially, but this also has the disadvantage of increasing the already bulky ribbon cable, and it is not guaranteed to solve the skew problem (skew can

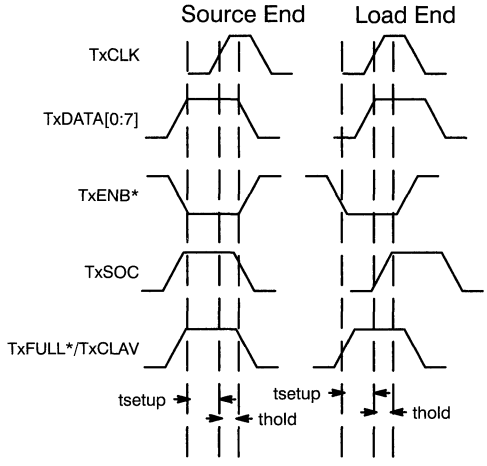


Figure 6. Effect of Skew on UTOPIA Bus

still result from differences in propagation delays for each signal through its respective differential driver/cable/receiver).

The Serial Solution

A good solution to the skew problems described above is to transmit the parallel bus data as a serial data stream. Transmitting the data serially requires a parallel-to-serial conversion of the UTOPIA data at the source end and a corresponding serial-to-parallel conversion at the load end. With such a scheme, the skew problems associated with operating a high-speed parallel bus over long distances are eliminated. In addition, the cable size is reduced from a multi-conductor ribbon cable to a two-conductor serial cable (such as coaxial cable).

The method by which a serial data transfer eliminates the skew problems associated with parallel buses is related to how serial links operate. Although some "serial" communication systems utilize more than one conductor (e.g., RS232), more serial links provide for transmission of only one signal. Note that to transmit one signal over copper media requires two conductors. This transmission can be either single-ended (requiring one conductor for the signal and one reference or ground) or differential (requiring two conductors for one signal). Both clock and data information must be included in this single signal. To accomplish this clock and data multiplexing function, serial links make use of special encoding schemes and use clock recovery circuits. The clock recovery circuits rely on the special characteristics of the data encoding scheme in order to recover or generate a clock of the same frequency and phase (with respect to the serial data) as the clock used to shift the data onto the serial link. The serial-to-parallel converter then uses this recovered clock to resample or retime the serial data before placing this data into a parallel word register. When this register is full, the serial-to-parallel converter presents the data in the register (in a parallel format) along with a parallel word clock (generated by dividing down the recovered serial clock). Thus, there is no skew between the clock and parallel data.

The main advantages of a serial link over a parallel bus are: (1) the clock is embedded with data, thus there is no skew between clock and data signals, (2) the distance over which the serial link is operated can be changed and the link will remain operational, (3) the transfer rate of the serial link can be scaled up and the link will remain operational, and (4) the cables required are smaller in size.

Serial Links and HOTLink™

The Cypress HOTLink™ chipset performs all of the functions shown in the simplified block diagram in *Figure 7*. The CY7B923 HOTLink Transmitter serves as the serializer while the CY7B933 HOTLink Receiver operates as a deserializer. In the HOTLink chipset, clock multiplication and clock recovery are accomplished using Phase Locked Loops (or PLLs). PLLs are closed loop control systems which align an output waveform in phase and frequency with an input waveform. Block diagrams of PLLs performing clock multiplication and clock recovery are shown in *Figure 8*.

PLLs operate by constantly comparing their output waveform with their input (or reference) waveform. Deviations in phase or frequency are then corrected at a rate governed by the Low Pass Filter (LPF). A wide bandwidth LPF allows a PLL to track high-frequency phase deviations between the reference and the output waveforms. A narrow bandwidth LPF dictates that the PLL rejects high-frequency phase

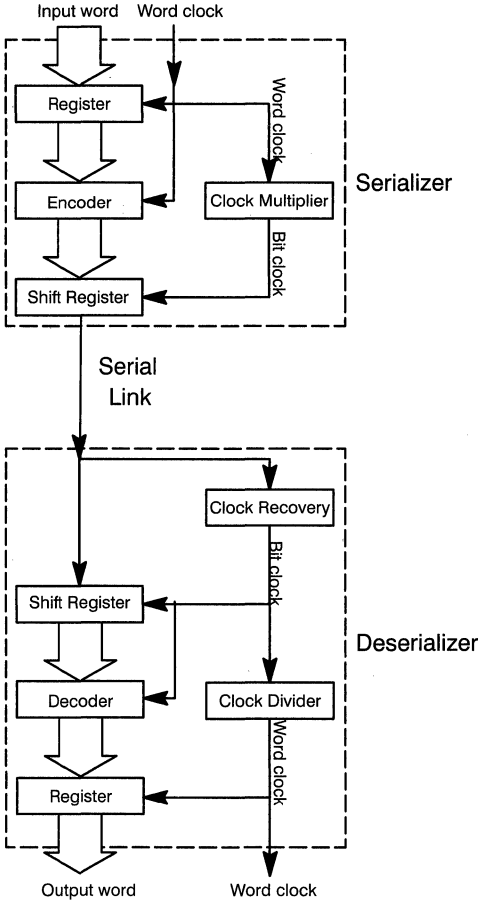


Figure 7. Architecture of a Serial Link

deviations between the reference and output waveform. Ideally, an input waveform would have a transition at a regular periodic rate, thus allowing the PLL to check its alignment constantly. However, such a signal would contain no information (essentially the link would be composed of one baseband frequency and its harmonics) and is not useful for data communication. Actual serial streams do not have data transitions at strictly periodic intervals. Instead, there are often "runs" of consecutive ones or zeros, which result in short periods where the serial stream has no transitions. The lack of transitions in the serial stream can cause the clock recovery PLL to fall out of phase lock, and eventually out

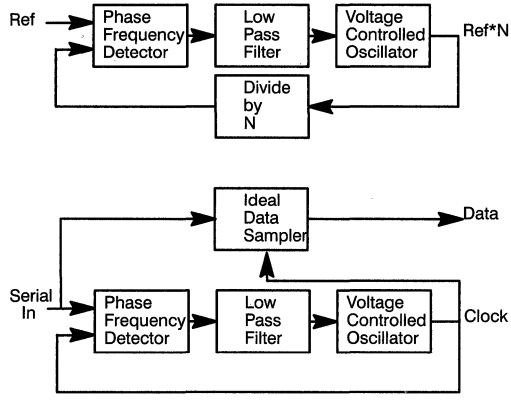


Figure 8. Multiplication and Clock/Data Recovery PLLs

of frequency lock. In order to reliably perform clock recovery with PLLs, the serial data needs to be encoded in such a way as to ensure there are frequent transitions (either from HIGH to LOW or LOW to HIGH) in the serial stream. These transitions cannot be ensured when sending unencoded data, since a user is free to send any data pattern. Some serial patterns like 00000000 contain no transitions and therefore could be transmited indefinitely resulting in a serial link without any transitions.

The HOTLink chipset utilizes an encoding scheme known as 8B/10B. This code takes in a 8-bit data word and converts it into a 10-bit transmission character. The transmission characters are chosen such that their run length is limited to 5 consecutive ones or zeros. With this encoding scheme, the HOTLink Receiver's clock recovery circuit can maintain lock and recover the clock from the serial data stream.

Serializing the UTOPIA Bus

Operating the UTOPIA bus over a serial link is accomplished using the architecture shown in *Figure 9*.

The basic block functions are as follows: On the ATM side, the serializer converts the parallel UTOPIA transmit data into a serial stream, embedding the UTOPIA transmit clock with the data. The deserializer converts the serial receive stream (from the PHY layer) back into parallel data and a receive clock. The First In First Out (FIFO) memory works

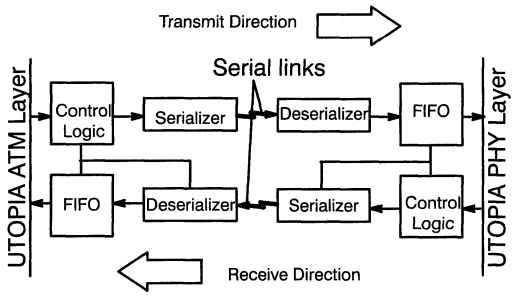


Figure 9. UTOPIA Serializer Block Diagram

as an elastic buffer, queuing the parallel receive data until the ATM layer parallel interface is ready to accept the data. The control logic provides control for all of the blocks. On the PHY side, the blocks perform similar functions. The serializer converts the parallel receive data into a serial stream, embedding the UTOPIA receive clock into the data. The deserializer converts the serial transmit stream (from the ATM layer) back into parallel data and a transmit clock. The FIFO provides buffering for the transmit interface, and the control logic manages all of the blocks.

Round Trip Latency

The purpose of the FIFO in the serialized UTOPIA architecture is to account for latency in the system. To understand the importance of the FIFO, consider a design which implemented a serialized UTOPIA bus. For UTOPIA transmits, there are two handshaking signals TX_FULL* (sourced at the PHY layer) and TX_ENB* (sourced at the ATM load). A transfer is initiated when TX_FULL* goes HIGH, followed by TX_ENB* going LOW and the UTOPIA data placed onto the bus. If TX_FULL* should go LOW at any time, the transfer must stop (according to the UTOPIA specification) within four write cycles. However, since TX_FULL* is sourced at the PHY layer and sampled at the ATM layer, there is a time delay for any change of state of TX_FULL* at the PHY layer to be recognized at the ATM layer. *Figure 10* shows an example of the timing relationships of the critical UTOPIA signals. This time delay is the latency through the serializer, serial media, and deserializer. There is a similar la-

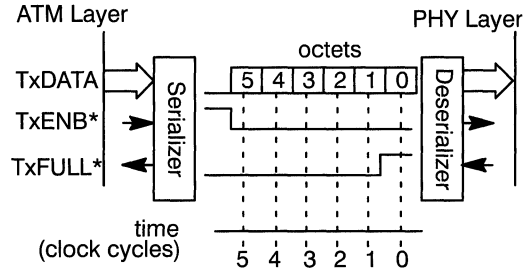


Figure 10. Round Trip Latency Example

tency with respect to the TX_ENB* and TX_DATA from the ATM layer to the PHY layer. A problem arises if a transfer is in progress and TX_FULL* goes LOW. The figure shows that the transfer began successfully and several octets were placed onto the serial link. However, at clock cycle 1, the TX_FULL* signal on the PHY side went LOW, indicating that the PHY layer is full. According to the UTOPIA specification, the transfer must stop (TX_ENB* must go HIGH) within four byte times of TX_FULL* going LOW. In order for TX_ENB* to go HIGH, the ATM layer must recognize the change in state of TX_FULL*, but there is a delay from the PHY layer to the ATM layer. During this delay, the ATM layer may have already sent out too many bytes (in *Figure 10* five bytes are shown as being transmitted before TX_FULL* is recognized at the ATM layer). Since it is possible to not recognize the change in state of TX_FULL* within the four byte specification, there is the potential for data loss at the PHY layer.

Note that the latency in the link that is the source of the problem in the above example is *not* entirely due to the serializer and deserializer. If the serial link itself is long enough, the mere time delay required for the electrical pulses to travel down the link may be enough to cause the problems described above.

The latency issue is solved by buffering the data coming out of the deserializer. A FIFO is an adequate buffer for this application. With the FIFO buffer, the effects of the link latency are corrected. When the PHY layer UTOPIA interface indicates it has no more room for data, the FIFO can store the octets that are sent by the ATM layer before it receives the TX_FULL* signal. The data can then be

read out of the FIFO when the PHY layer UTOPIA interface is ready.

The UTOPIA Extender

Following the block diagram shown in *Figure 9*, and the hierarchical schematics shown in Appendix A, a serialized UTOPIA bus can be implemented. With the bus serialized, it can essentially be extended to any length, thus the design results in a "UTOPIA extender." The major components required to implement such a design are shown in *Table 2*.

Table 2. Cypress UTOPIA Extender Components

Generic Part	Cypress Part
Serializer	CY7B923 HOTLink Tx
Deserializer	CY7B933 HOTLink Rx
FIFO	CY7B451 512x9 clocked FIFO
Control Logic	CY7C371 32-macrocell Flash PLD

The "Top Level" hierarchical schematic shows a generic breakdown of the entire design. The "ATM Layer UTOPIA Extender" block implements all of the functions at the ATM layer interface necessary to serialize the UTOPIA bus. Likewise, the "PHY Layer UTOPIA Extender" block implements all of the functions at the PHY layer interface. Between these two blocks are two serial links over which the serialized UTOPIA bus operates. A system level application of the UTOPIA Extender is shown in *Figure 11*.

Both the "ATM" and "PHY Layer UTOPIA Extender" blocks have additional hierarchical schematics associated with them. Within these lower-level hierarchical schematics are additional blocks that show more detail than the previous levels. Each block performs a specific function necessary for the operation of the entire design. Some functions are common to both the "ATM" and "PHY Layer UTOPIA Extender" blocks, such as the "Media Interface" block. The "Media Interface" block performs the function of interfacing the transmit and receive electrical signals (comprising the serial links carrying the serialized UTOPIA bus) to the specific media interface used in the design (in this case to co-

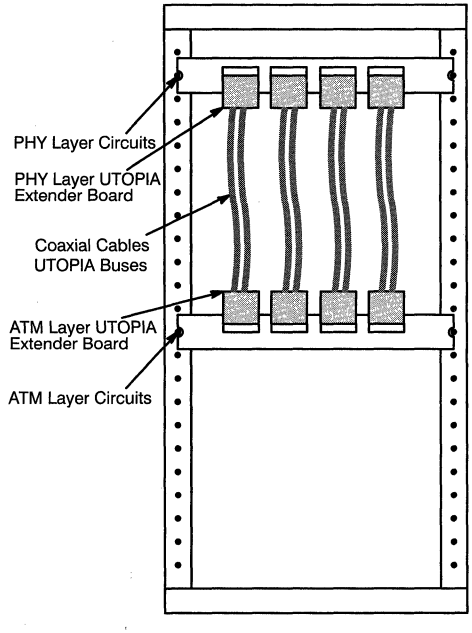


Figure 11. UTOPIA Extender in a Rack Mount Switch

axial cable). The "Media Interface" schematic contains termination networks and transformers used to interface the transmit and receive serial signals to the coaxial cable.

The "ATM" and "PHY UTOPIA Logic" blocks contain all of the circuits used to serialize the UTOPIA bus. These blocks contain the serializers, deserializers, FIFOs, and PLDs used to implement the logic for the UTOPIA extender.

The operation of the UTOPIA extender, implemented in the "ATM" and "PHY UTOPIA Logic" blocks, can be broken down into two modes. The first mode, or Steady State mode, moves the UTOPIA transmit and receive data between the ATM and PHY layers, and handles generation of the necessary control signals. The second mode, or FIFO State Update mode, handles the control of the buffering FIFOs assuring that no data is lost due to overfilling of these buffers. This mode also handles the case of the CLAV signals going inactive, indicating the UTOPIA interface cannot accept more

data. Regardless of the mode of operation, the basic link operation revolves around the Cell Level Handshaking (or CLH) protocol.

The main characteristic of CLH is that once a cell transmission begins, all 53 octets of the cell are sent in succession on consecutive clocks. In this mode, back to back cell transmissions are also possible. For this design, however, back to back cell transmissions will not be allowed (this is accomplished through special considerations in the UTOPIA control logic). A gap will be forced between all cells. This gap serves two purposes. The first is to allow for the communication of the CLAV control codes from the PHY layer to the ATM layer and also to update the status of the buffering FIFOs. The second reason for the gap is to allow for easy generation of the SOC signal at the load end of the serial link.

The Steady State mode of operation for the UTOPIA extender is defined as the condition when neither buffer FIFO is overfilled. When in this mode, there is a minimal amount of control logic necessary to implement the extender. As an example, consider a UTOPIA transmit (defined as data movement from the ATM to the PHY layer). When a 53-octet cell becomes available on the ATM layer side, it is immediately placed into the HOTLink transmitter and sent over to the PHY side. Following the first octet, the remaining 52 octets of the cell are sent consecutively. Following transmission of the 53rd byte, the link pauses to implement the forced cell gap. During this pause, the HOTLink Transmitter is disabled and sends idle characters (called K28.5 or "Commas") across the link. If there is another cell available from the ATM layer, it is sent across after the cell gap. If no data is available, the link remains disabled. The flow of data under the steady state mode is shown in *Figure 12.*

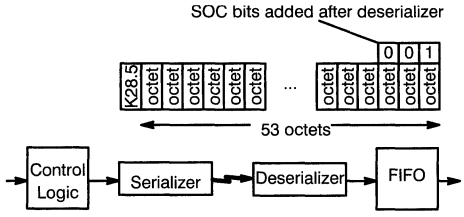


Figure 12. Transmission Data Flow

Upon receiving the octets from the ATM layer, the output of the HOTLink Receiver is immediately placed into the buffering FIFO. In addition, when the first octet out of the receiver is sensed (by taking advantage of the forced gap between cells), an additional bit, serving as the TX_SOC signal, is placed into the FIFO coincident with the first octet. The remaining 52 octets are also placed into the FIFO, but without the TX_SOC bit set. The TX_ENB* signal to the UTOPIA interface is then generated from the TX_CLAV signal and the FIFO status signals. The PHY UTOPIA interface directly reads the output of the buffering FIFO. Data movement in the UTOPIA receive direction is similar.

The other mode of operation is FIFO State Updating. This mode basically serves to handle the case when the CLAV signals change state. That is, if the TX_CLAV is deasserted, no data will be read out of the PHY side buffering FIFO. Eventually, this FIFO will fill beyond a check point and a code will be sent back to the ATM layer side indicating no more data should be sent until the FIFO is read beyond a certain level. The operation of this mode requires some additional control logic. Again, consider the case of UTOPIA transmission. A FIFO state update begins when the control logic on the PHY layer side detects that the buffering FIFO has filled beyond a predefined level. The control logic then waits for a pause in the data stream going back to the ATM layer side (remember a gap is forced between successive cells). During this pause, the control logic inserts a "FIFO Full" control code into the HOTLink transmitter in place of one of the comma characters (see *Figure 13*). This FIFO Full code travels across the link back to the ATM layer side. The ATM layer control logic then interprets the FIFO Full code and deasserts the TX_CLAV signal at the ATM layer UTOPIA interface, thus stopping transmission on the next cell boundary.

Eventually, the PHY layer FIFO will empty past another predefined level, thus indicating data transmission can begin again. The control logic on the PHY layer side then waits for a pause in the data stream back to the ATM layer side, and inserts a "FIFO Not Full" code in place of one of the comma characters (see *Figure 14*). This code travels down the link back to the ATM layer side where it is inter-

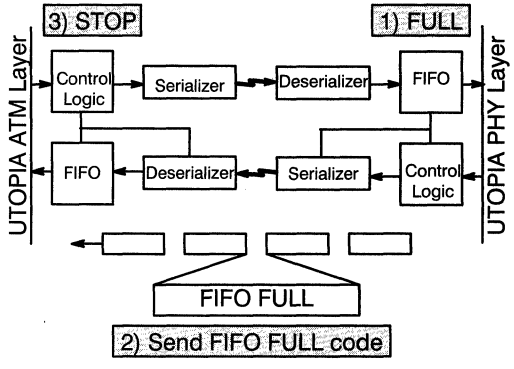


Figure 13. FIFO State Updating, FIFO Full

preted by the ATM layer control logic. The control logic then asserts the TX_CLAV signal to the ATM layer UTOPIA interface allowing data transmission to resume. Operation then reverts back to the Steady State mode.

The remaining blocks in the UTOPIA Extender ("ATM UTOPIA and Processor Interface," "PHY UTOPIA and Processor Interface," and "Framer Processor Interface") are used to interface the "ATM" and "PHY UTOPIA Logic" blocks to the UTOPIA bus of the ATM and PHY Layer Circuits as shown in *Figure 11*. In general, these remaining blocks contain connectors with pinouts specific to the particular ATM/PHY layer circuits used in the system. In addition, some ATM and/or PHY layer circuits require additional circuits to configure and/or monitor their operation. Thus the actual design of the "ATM UTOPIA and Processor Interface," "PHY UTOPIA and Processor Interface," and "Framer Processor Interface" blocks differs depending on the unique ATM and PHY layer circuits used in the system.

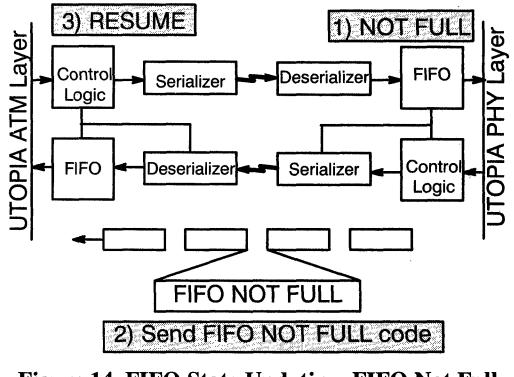


Figure 14. FIFO State Updating, FIFO Not Full

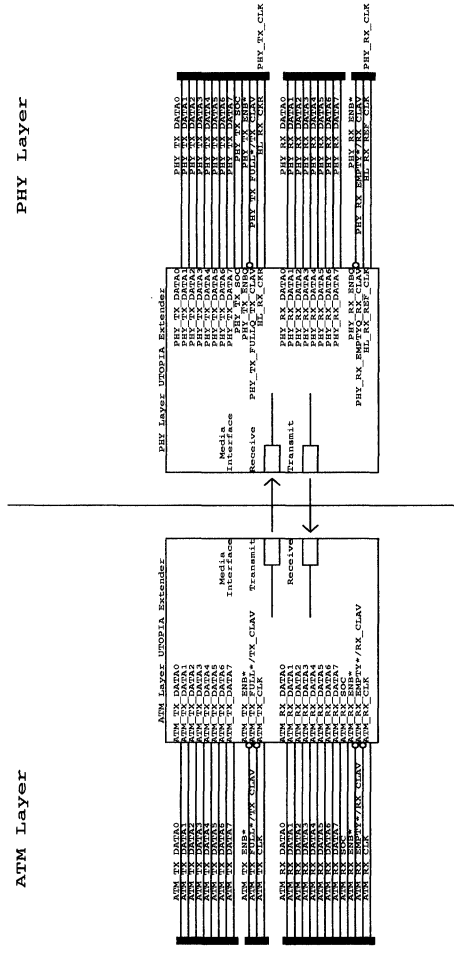
To exemplify a system using the UTOPIA Extender, a complete design of the PHY Layer is shown in the schematics (that is, only the "PHY Layer UTOPIA Extender" is shown fully implemented). The PHY Layer Circuit used was a Duke Communications DC-202® SONET/ATM UNI Transceiver Module. Thus the "PHY UTOPIA and Processor Interface" block was tailored to interface to the DC-202. In addition, the "Framer and Processor Interface" block was required to configure the DC-202 for proper operation. VHDL code for the "Framer and Processor Interface Block" is included in Appendix B. Also included in Appendix B is VHDL code implementing the algorithms for the "PHY UTOPIA Logic" PLD.

Conclusions

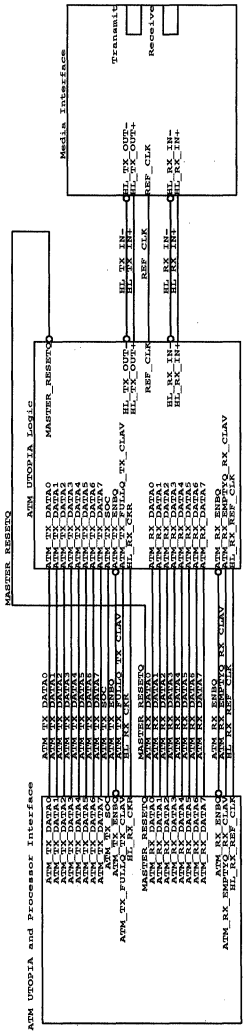
This application note has shown that signal skew across a ribbon cable can limit the operational distance of high-speed parallel buses such as UTOPIA. Serial links can operate over longer distances since they are not susceptible to the skew effects that limit parallel buses. This application note describes the design of a serialized parallel bus called the "UTOPIA Extender." Implementation of the UTOPIA Extender requires only a minimal amount of logic, with most of the work being performed by a high-speed serial-link chipset such as the Cypress HOTLink chipset.

HOTLink is a trademark of Cypress Semiconductor.
DC-202 is a registered trademark of Duke Communications.

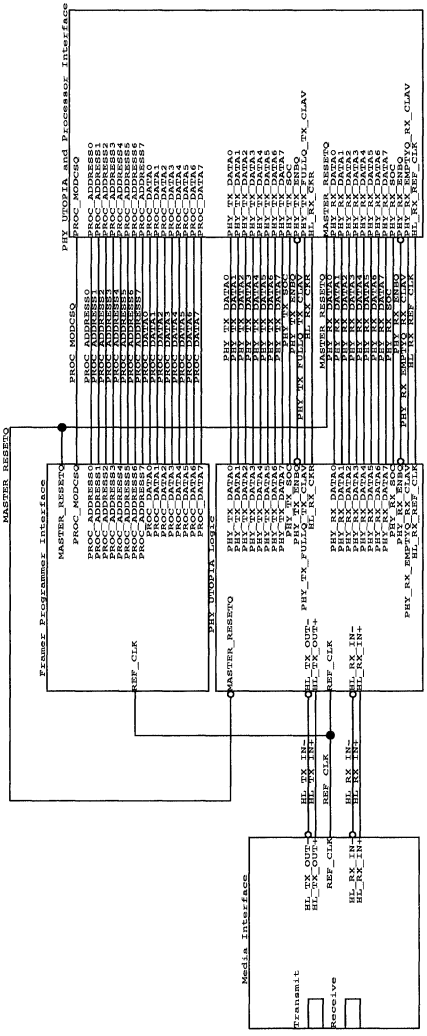
Appendix A. Hierarchical Schematics
Sheet 1 of 7: Top Level



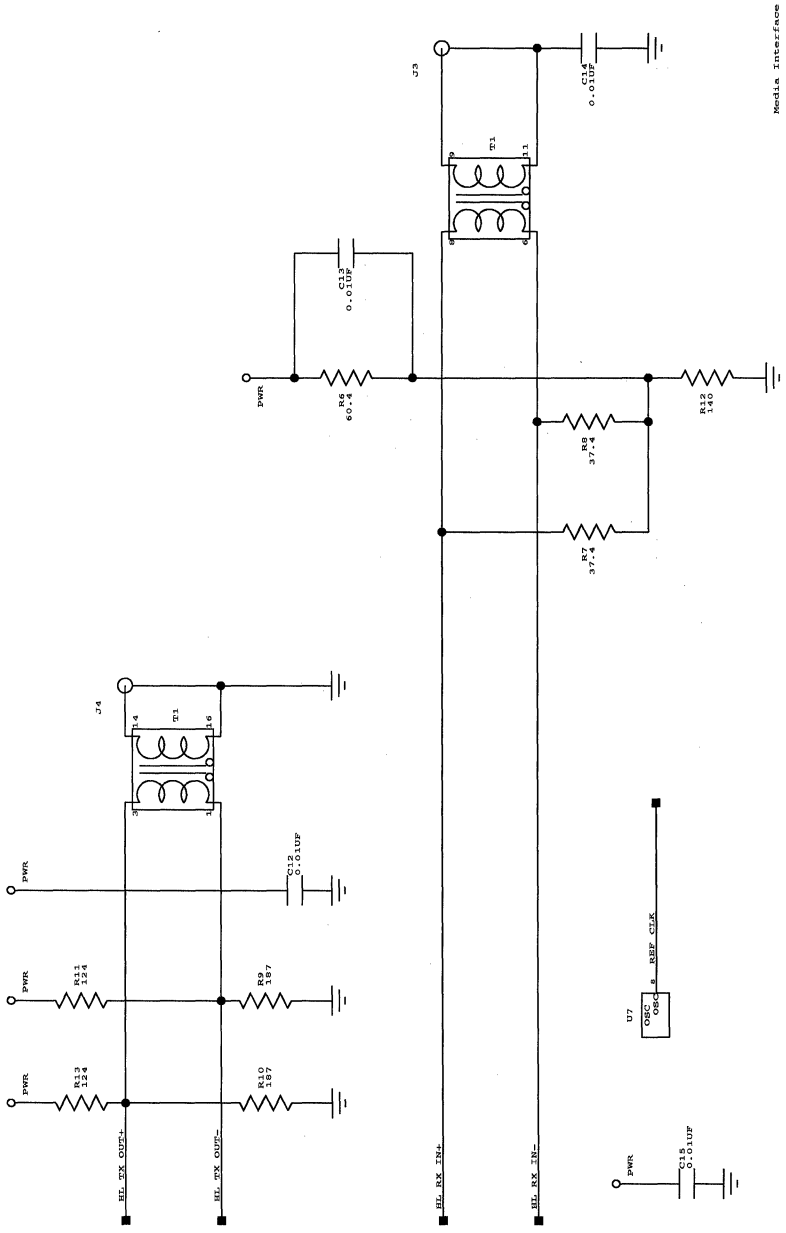
Appendix A. Hierarchical Schematics
Sheet 2 of 7: ATM Layer UTOPIA Extender



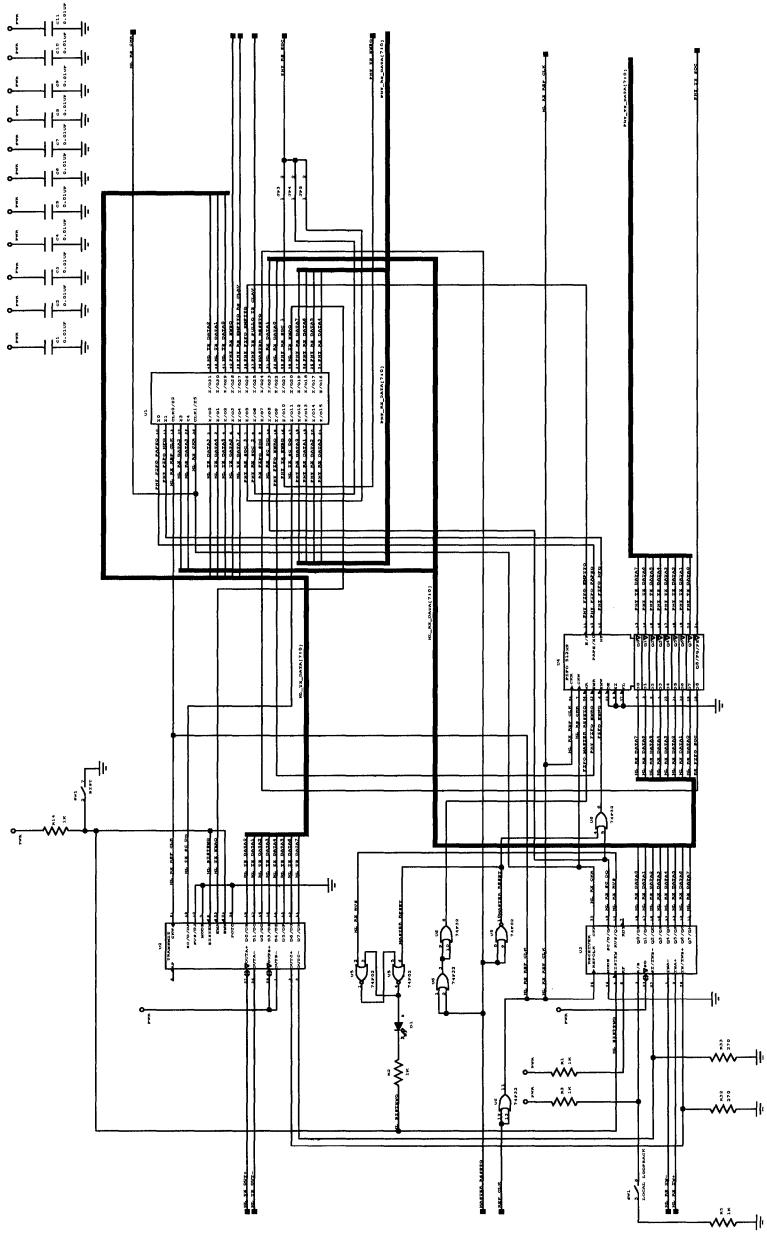
Appendix A. Hierarchical Schematics
Sheet 3 of 7: PHY Layer UTOPIA Extender



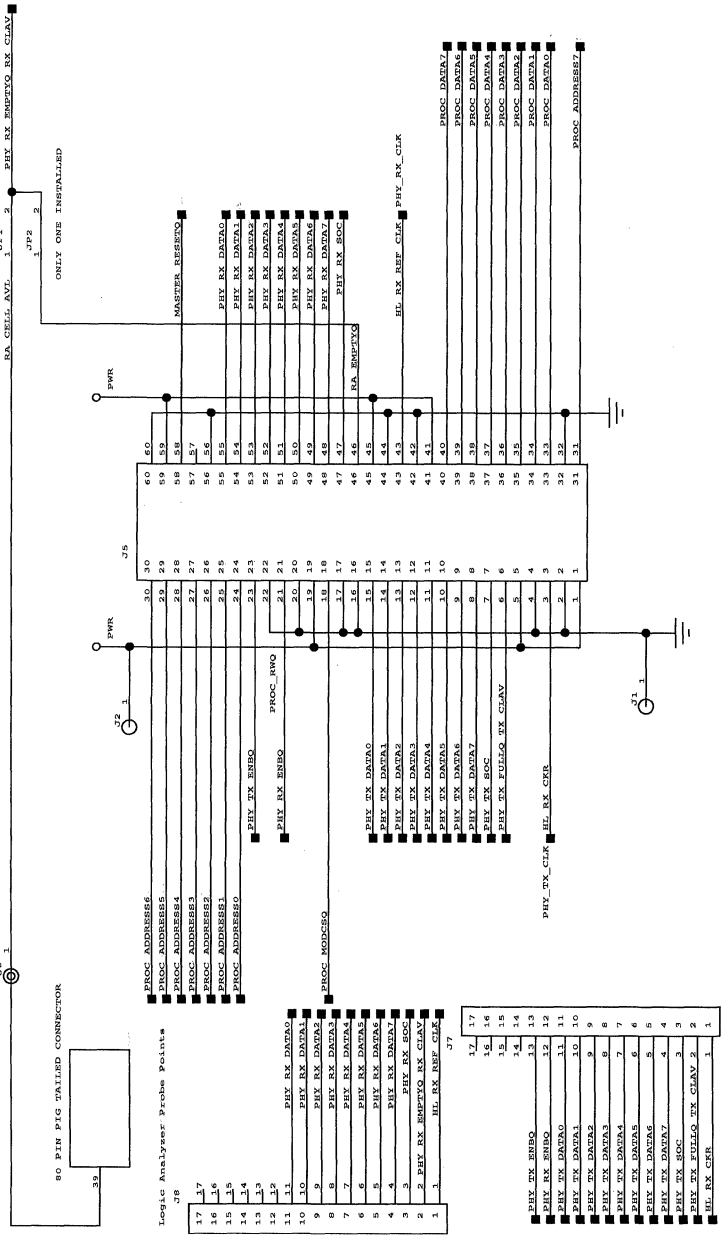
Appendix A. Hierarchical Schematics
Sheet 4 of 7: Media Interface



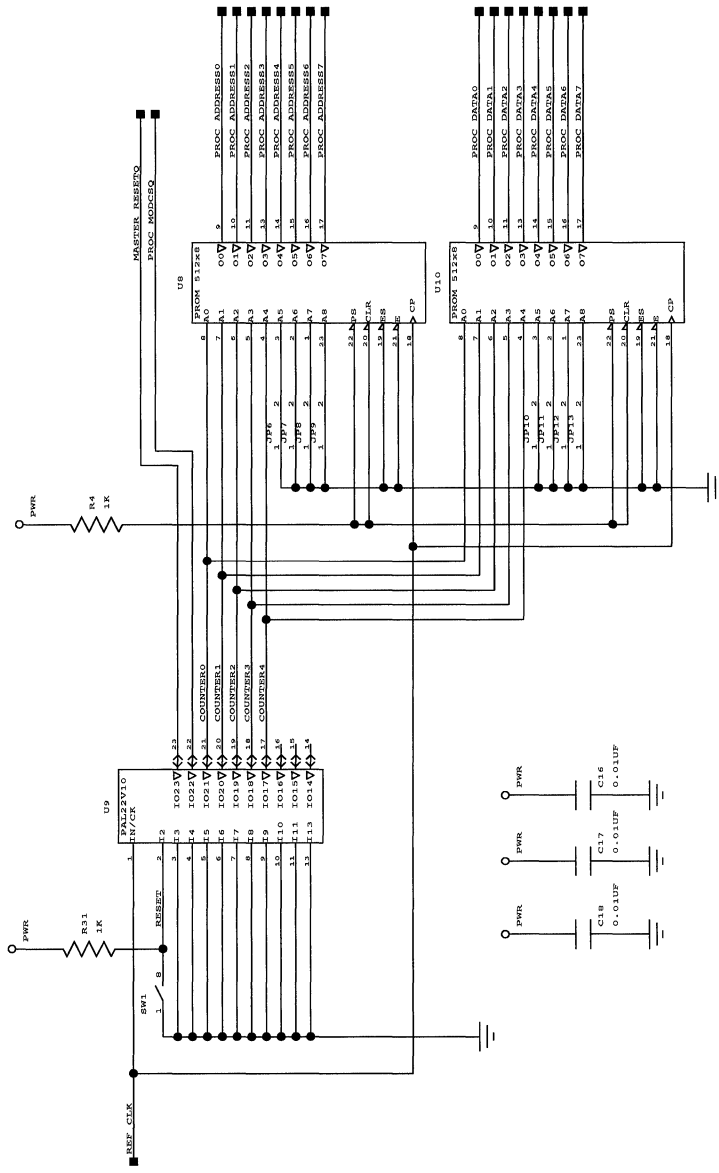
Appendix A. Hierarchical Schematics
Sheet 5 of 7: PHY UTOPIA Logic



Appendix A. Hierarchical Schematics
Sheet 6 of 7: PHY UTOPIA Processor Interface



Appendix A. Hierarchical Schematics
Sheet 7 of 7: Framer Programmer Interface



Appendix B: VHDL Code
UTOPIA Extender, PHY Layer

```
-- UTOPIA extender, PHY layer
--
--

USE WORK.phy_utopia_transmitter_package.ALL;
USE WORK.phy_utopia_receiver_package.ALL;

ENTITY phy_utopia IS
        PORT(   hl_rx_ckr, hl_rx_sc_d,
                master_reset,
                phy_tx_full_tx_clav,
                phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                rx_fifo_soc,
                phy_tx_enb, phy_fifo_enr        : INOUT BIT;

                phy_rx_clk,
                phy_rx_empty_rx_clav            : IN BIT;
                phy_rx_data                     : IN BIT_VECTOR(0 to 7);
                hl_tx_sc_d, hl_tx_ena,
                phy_rx_enb                      : INOUT BIT;
                hl_tx_data                      : INOUT BIT_VECTOR(0 to 7));

        ATTRIBUTE pin_numbers OF phy_utopia:ENTITY IS
                "hl_tx_data(3):2 " &
                "hl_tx_data(4):3 " &
                "hl_tx_data(5):4 " &
                "hl_tx_data(6):5 " &
                "hl_tx_data(7):6 " &
                "rx_fifo_soc:9 " &
                "phy_fifo_pafe:10 " &
                "phy_fifo_hf:11 " &
                "phy_rx_clk:13 " &
                "hl_rx_sc_d:14 " &
                "phy_fifo_enr:15 " &
                "phy_tx_enb:16 " &
                "hl_tx_sc_d:17 " &
                "phy_rx_data(0):18 " &
                "phy_rx_data(1):19 " &
                "phy_rx_data(2):20 " &
                "phy_rx_data(3):21 " &
                "phy_rx_data(4):24 " &
                "phy_rx_data(5):25 " &
                "phy_rx_data(6):26 " &
                "phy_rx_data(7):27 " &
                "hl_tx_ena:28 " &
                "hl_rx_data(0):30 " &
                "hl_rx_data(1):31 " &
                "hl_rx_data(2):32 " &
                "hl_rx_data(3):33 " &
                "hl_rx_ckr:35 " &
                "master_reset:36 " &
                "phy_tx_full_tx_clav:37 " &
                "phy_fifo_empty:38 " &
                "phy_rx_empty_rx_clav:39 " &
                "phy_rx_enb:40 " &
                "hl_tx_data(0):41 " &
                "hl_tx_data(1):42 " &
                "hl_tx_data(2):43 ";

END phy_utopia;

ARCHITECTURE netlist OF phy_utopia IS

        SIGNAL atm_fifo_hf_code                 : BIT;
        SIGNAL atm_fifo_not_hf_code             : BIT;
       SIGNAL phy_fifo_hf_state             : BIT;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer (continued)

```
BEGIN
        U1: phy_utopia_transmitter
                PORT MAP (hl_rx_ckr, hl_rx_sc_d, master_reset,
                            phy_tx_full_tx_clav, phy_fifo_hf,
                          phy_fifo_pafe, phy_fifo_empty, hl_rx_data,
                            phy_fifo_hf_state, rx_fifo_soc,
                          atm_fifo_hf_code, atm_fifo_not_hf_code,
                          phy_tx_enb, phy_fifo_enr);

        U2: phy_utopia_receiver
                PORT MAP (phy_rx_clk, phy_rx_empty_rx_clav, master_reset,
                          atm_fifo_hf_code, atm_fifo_not_hf_code,
                          phy_fifo_hf_state, phy_rx_data, hl_tx_sc_d,
                            hl_tx_ena, phy_rx_enb, hl_tx_data);
END netlist;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Transmitter Interface (PHY to ATM)

```
-- UTOPIA extender, PHY layer transmitter interface (PHY to ATM).
--

PACKAGE phy_utopia_transmitter_package IS

COMPONENT phy_utopia_transmitter

                -- Note, hl_rx_ckr = phy_tx_clk.

        PORT(   hl_rx_ckr, hl_rx_sc_d,
                master_reset,
                phy_tx_full_tx_clav,
               phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                phy_fifo_hf_state,
               rx_fifo_soc, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
                phy_tx_enb, phy_fifo_enr        : INOUT BIT);

END COMPONENT;
END phy_utopia_transmitter_package;

ENTITY phy_utopia_transmitter IS
        PORT(   hl_rx_ckr, hl_rx_sc_d,
                master_reset,
                phy_tx_full_tx_clav,
               phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                phy_fifo_hf_state,
               rx_fifo_soc, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
                phy_tx_enb, phy_fifo_enr        : INOUT BIT);

END phy_utopia_transmitter;

ARCHITECTURE behavior OF phy_utopia_transmitter IS

        -- Codes received from ATM side pertaining to the state
        --      of the ATM side FIFO.  Note, the 'fifo_hf_code'
        --      is a HOTLink K28.0 code, while the 'fifo_not_hf_code'
        --      is a HOTLink K28.2 code.

        CONSTANT fifo_hf_code : BIT_VECTOR := X"2";
        CONSTANT fifo_not_hf_code : BIT_VECTOR := X"0";

        SIGNAL phy_tx_enb_wait                  : BIT;

BEGIN

        -- Generate the FIFO read enable signal using the invert of
        -- phy_tx_full_tx_clav.  Also, want to disable when resetting.

        phy_fifo_enr <= NOT(phy_tx_full_tx_clav) OR NOT(master_reset);

        -- Note that data out of the FIFO is valid on the rising edge
        -- AFTER the data is read out.  So, want to delay the phy_tx_enb
        -- one clock from the FIFO read enable.

        PROCESS
        BEGIN
                WAIT UNTIL hl_rx_ckr = '1';
                phy_tx_enb_wait <= phy_fifo_empty AND phy_tx_full_tx_clav;

        END PROCESS;

        phy_tx_enb <= NOT(phy_tx_enb_wait) OR NOT(master_reset);

        -- Essentially, rx_fifo_soc is a one clock delay (w.r.t.
        -- hl_rx_ckr) of the hl_rx_sc_d pin.  This is then used to
        -- generate the input bit to the FIFO for the phy_tx_soc signal.
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Transmitter Interface (PHY to ATM) (continued)

```
    PROCESS
    BEGIN

          WAIT UNTIL hl_rx_ckr = '1';
          rx_fifo_soc <= hl_rx_sc_d;

    END PROCESS;

  PROCESS
  BEGIN

          WAIT UNTIL hl_rx_ckr = '1';

          IF ((hl_rx_data = fifo_hf_code) AND (hl_rx_sc_d = '1')) THEN
                  atm_fifo_hf_code <= '1';
          ELSIF ((hl_rx_data = fifo_not_hf_code) AND (hl_rx_sc_d = '1'))
                  THEN
                  atm_fifo_not_hf_code <= '1';
          ELSE
                  atm_fifo_hf_code <= '0';
                  atm_fifo_not_hf_code <= '0';
          END IF;
  END PROCESS;

    PROCESS (master_reset, phy_fifo_pafe, phy_fifo_hf)

    -- Hysterisis is added to the PHY FIFO half-full flag via the
    --      input 'phy_fifo_hf_state'.  Thus, the half-full state
    --      is set to TRUE (1) when 'phy_fifo_hf' = 0.  The half-full state
    --      is set to FALSE (0) when 'phy_fifo_pafe' = 0.

    BEGIN

    phy_fifo_hf_state <= (NOT(phy_fifo_hf) OR (phy_fifo_pafe AND
                          phy_fifo_hf_state)) AND (master_reset);

    END PROCESS;
END behavior;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM)

```
-- UTOPIA extender, PHY layer receiver interface (PHY to ATM).
--
--

PACKAGE phy_utopia_receiver_package IS

COMPONENT phy_utopia_receiver
        PORT(   phy_rx_clk, phy_rx_empty_rx_clav,
               master_reset, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
               phy_fifo_hf_state             : IN BIT;
                phy_rx_data                     : IN BIT_VECTOR(0 to 7);
                hl_tx_sc_d, hl_tx_ena,
                phy_rx_enb                      : INOUT BIT;
                hl_tx_data                      : INOUT BIT_VECTOR(0 to 7));

END COMPONENT;
END phy_utopia_receiver_package;

ENTITY phy_utopia_receiver IS

        PORT(   phy_rx_clk, phy_rx_empty_rx_clav,
                master_reset, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
               phy_fifo_hf_state             : IN BIT;
                phy_rx_data                     : IN BIT_VECTOR(0 to 7);
                hl_tx_sc_d, hl_tx_ena,
                phy_rx_enb                      : INOUT BIT;
                hl_tx_data                      : INOUT BIT_VECTOR(0 to 7));


END phy_utopia_receiver;

ARCHITECTURE behavior OF phy_utopia_receiver IS

        -- Codes received from ATM side pertaining to the state
        --      of the PHY side FIFO.  Note, the 'fifo_hf_code'
        --      is a HOTLink K28.0 code, while the 'fifo_not_hf_code'
        --      is a HOTLink K28.2 code.
        -- 'packet_size' is the number of bytes in a packet (i.e. 53 bytes)
        -- 'packet_gap' is the minimum number clocks allowed between
        --      packets.
        -- 'packet_start_delay' is the number of clocks from when 'phy_rx_enb'
        --      is valid to when data appears at the PHY UTOPIA receiver
        --      interface.  Currently, this is defined by the UTOPIA spec.
        --      as 1 clock.

        CONSTANT fifo_hf_code                   : BIT_VECTOR := X"02";
        CONSTANT fifo_not_hf_code               : BIT_VECTOR := X"00";
        CONSTANT packet_size                    : INTEGER := 53;
        CONSTANT packet_gap                     : INTEGER := 1;
        CONSTANT packet_start_delay             : INTEGER := 0;

        -- State of ATM side FIFO maintained on PHY side as 'atm_fifo_hf'.
        -- State of PHY side FIFO as known on ATM side is
        --      'phy_fifo_hf_on_atm'.

        SIGNAL atm_fifo_hf                      : BIT:='0';
        SIGNAL phy_fifo_hf_on_atm               : BIT:='0';

        -- The 'counter' signal is used to establish the length of
        --      the packet from the PHY UTOPIA receiver interface.  It
        --      is also used to assure that there are a sufficient number
        --      of clocks in between packets as defined by 'packet_gap'.
       --      The 'hotlink_idle' signal is used to indicate no data
       --      is being transmitted by the HOTLink Tx and thus the
       --      Tx could be used to send FIFO update codes.

        SIGNAL counter                          : INTEGER(0 to packet_size):=0;
       SIGNAL hotlink_idle                     : BIT:='0';
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM) (continued)

```
        TYPE    state_type IS (wait_here, start_delay, count, cell_gap);
        SIGNAL  present_state, next_state     : state_type := wait_here;

BEGIN

        PROCESS (master_reset, atm_fifo_hf_code, atm_fifo_not_hf_code)
        BEGIN

                IF (master_reset = '0' OR atm_fifo_not_hf_code = '1') THEN
                        atm_fifo_hf <= '0';
                ELSIF (atm_fifo_hf_code = '1') THEN
                        atm_fifo_hf <= '1';

                        -- Set 'atm_fifo_hf' to 1 when receive
                        --      'atm_fifo_hf_code' and clear when receive
                        --      'atm_fifo_not_hf_code'.

                END IF;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL phy_rx_clk = '1';

                IF (present_state /= next_state)
                THEN
                        counter <= 1;
                ELSE
                        counter <= counter +1;
                END IF;

        END PROCESS;

        PROCESS(present_state, counter, phy_rx_empty_rx_clav, atm_fifo_hf,
                master_reset)

                -- 'phy_rx_empty_rx_clav' is 1 when the PHY side has
                --      a full cell (53 bytes).  So, if the ATM side
                --      FIFO is not half-full, then set 'phy_rx_enb'
                --      to 0 and start transmitting cells back to the
                --      ATM side.  Stop (i.e. set 'phy_rx_enb' to 1)
                --      after 53 bytes to prevent back to back cell
                --      transfers from the PHY UTOPIA receiver interface.
                --      Wait an additional 'packet_gap' number of clocks
                --      before reenabling the receiver via 'phy_rx_enb'.
                --      We must assure that there are at least packet_gap
                --      bytes between packets in order to recreate the
                --      rx_soc signal on the ATM side.  This gap will
                --      also be used to send PHY FIFO state codes to
                --      the ATM side.
        BEGIN

                CASE present_state IS

                WHEN wait_here =>

                        phy_rx_enb <= '1';
                        hotlink_idle <= '1';
                        IF (phy_rx_empty_rx_clav = '1' AND atm_fifo_hf = '0'
                                AND master_reset = '1')
                        THEN
                                IF (counter < packet_start_delay)
                                THEN
                                        next_state <= start_delay;
                                ELSE
                                        next_state <= count;
                                END IF;
                        ELSE
                                next_state <= wait_here;
                        END IF;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM) (continued)

```
        WHEN start_delay =>
                phy_rx_enb <= '0';
                hotlink_idle <= '1';
                IF ((counter < packet_start_delay)
                        AND master_reset = '1')
                THEN
                        next_state <= start_delay;
                ELSIF (master_reset = '0')
                THEN
                        next_state <= wait_here;
                ELSE
                        next_state <= count;
                END IF;
        WHEN count =>
                 phy_rx_enb <= '0';
                hotlink_idle <= '0';
                IF ((counter < packet_size)
                        AND master_reset = '1')
                THEN
                        next_state <= count;
                ELSIF (master_reset = '0')
                THEN
                        next_state <= wait_here;
                ELSE
                        next_state <= cell_gap;
                END IF;
        WHEN cell_gap =>
                 phy_rx_enb <= '1';
                hotlink_idle <= '1';
                IF (counter < packet_gap)
                THEN
                        next_state <= cell_gap;
                ELSIF (phy_rx_empty_rx_clav = '1'
                        AND atm_fifo_hf = '0' AND master_reset = '1')
                THEN
                        IF (packet_start_delay < packet_gap)
                        THEN
                                next_state <= count;
                        ELSE
                                next_state <= start_delay;
                        END IF;
                ELSE
                        next_state <= wait_here;
                END IF;
        END CASE;
END PROCESS;

PROCESS
BEGIN
        WAIT UNTIL phy_rx_clk = '1';
        present_state <= next_state;
END PROCESS;

PROCESS (phy_fifo_hf_state, phy_fifo_hf_on_atm, hotlink_idle,
                phy_rx_clk)
BEGIN
        -- If hotlink_idle = '0' send data.
        IF (hotlink_idle = '0')
        THEN
                hl_tx_ena <= '0';
                hl_tx_sc_d <= '0';
                hl_tx_data <= phy_rx_data;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM) (continued)

```
                -- If the HOTLink is idle (no data being sent) and
                --      the FIFO state needs updating, send the code.

                ELSE
                        hl_tx_sc_d <= '1';
                        IF (phy_fifo_hf_state /= phy_fifo_hf_on_atm)
                        THEN
                                hl_tx_ena <= '0';
                                IF (phy_fifo_hf_state = '1') THEN
                                        hl_tx_data <= fifo_hf_code;
                                ELSE
                                        hl_tx_data <= fifo_not_hf_code;
                                END IF;
                        ELSE
                                hl_tx_ena <= '1';
                        END IF;
                END IF;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL phy_rx_clk = '1';

                IF hotlink_idle = '1'
                THEN
                        phy_fifo_hf_on_atm <= phy_fifo_hf_state;
                END IF;
        END PROCESS;
END behavior;
```

Appendix B: VHDL Code
UTOPIA Extender, Duke PHY Board Programmer

```
-- UTOPIA extender, Duke PHY board programmer
--

PACKAGE duke_programmer_package IS

COMPONENT duke_programmer
        PORT(   ref_clk, reset                          : IN BIT;
                proc_modcs, master_reset        : INOUT BIT;
                counter                         : INOUT INTEGER(0 to 24));

END COMPONENT;
END duke_programmer_package;

ENTITY duke_programmer IS
        PORT(   ref_clk, reset                          : IN BIT;
                proc_modcs, master_reset        : INOUT BIT;
                counter                         : INOUT INTEGER(0 to 24));

       ATTRIBUTE pin_numbers OF duke_programmer:ENTITY IS
                "reset:2 " &
                "ref_clk:1 " &
                "counter(0):21 " &
                "counter(1):20 " &
                "counter(2):19 " &
                "counter(3):18 " &
                "counter(4):17 " &
                "proc_modcs:22 " &
                "master_reset:23 ";

END duke_programmer;

ARCHITECTURE behavior OF duke_programmer IS

       CONSTANT num_values                      : INTEGER :=24;

       TYPE state_type IS (wait_here, do_reset, count1, count2, count3);

       TYPE addrdata IS ARRAY(0 to num_values - 1) OF BIT_VECTOR(0 to 7);

       CONSTANT addresses : addrdata :=
       (       X"81",
               X"81",
               X"8D",
               X"8D",
               X"20",
               X"80",
               X"82",
               X"83",
               X"84",
               X"85",
               X"86",
               X"87",
               X"88",
               X"89",
               X"8A",
               X"8B",
               X"8C",
               X"8E",
               X"8F",
               X"90",
               X"91",
               X"92",
               X"9E",
               X"9F");
```

Appendix B: VHDL Code
UTOPIA Extender, Duke PHY Board Programmer (continued)

```
        CONSTANT data : addrdata :=
        (       X"01",
                X"00",
                X"01",
                X"00",
                X"0A",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00");

        SIGNAL  present_state, next_state     : state_type := wait_here;

BEGIN

        PROCESS(present_state, reset, ref_clk)
        BEGIN

                CASE present_state IS

                WHEN wait_here =>

                        master_reset <= '1';
                        proc_modcs <= '1';
                        IF (reset = '0')
                        THEN
                                next_state <= do_reset;
                        ELSE
                                next_state <= wait_here;
                        END IF;

                WHEN do_reset =>

                        master_reset <= '0';
                        proc_modcs <= '1';
                        next_state <= count1;

                WHEN count1 =>

                        master_reset <= '1';
                        proc_modcs <= '1';
                        next_state <= count2;

                WHEN count2 =>

                        master_reset <= '1';
                        proc_modcs <= '0';
                        next_state <= count3;
```

Appendix B: VHDL Code
UTOPIA Extender, Duke PHY Board Programmer (continued)

```
                WHEN count3 =>

                        master_reset <= '1';
                        proc_modcs <= '1';
                        next_state <= count2;
                        IF (counter < num_values - 1)
                        THEN
                                next_state <= count1;
                        ELSE
                                next_state <= wait_here;
                        END IF;

                END CASE;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL ref_clk = '1';

                present_state <= next_state;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL ref_clk = '1';

                IF (present_state = count3)
                THEN
                        counter <= counter + 1;
                END IF;
        END PROCESS;

END behavior;
```

Using High-Speed Serial Links to Supplement Parallel Data Buses

Today's designers face a multitude of problems when trying to move data within their systems. These problems range from overtaxed parallel-bus bandwidth to a lack of pins at the card edge connector. Even routing parallel buses around today's dense circuit boards is very difficult. This application note discusses using high-performance serial links as a solution to some of these bottlenecks. A serial approach provides three immediate benefits: first, bandwidth may be offloaded from the backplane bus; second, connector pins are saved; and, third, circuit board routing is made much easier since only two traces have to be routed for the data path (versus one for each data bus bit).

The ideal serial interface building block would be a chip set consisting of high-speed parallel-to-serial and serial-to-parallel converters (also referred to as transmitter and receiver). Additionally, this chip set would make the serial interface transparent to the user, i.e., parallel data would flow in one side and out the other. It would be able to use a variety of serial media directly such as coaxial cable, twisted-pair cable or even fiberoptic cable (when connected to the proper optical driver). It would also be easily adaptable to user-defined protocols for applications involving Direct Memory Access (DMA).

HOTLink™

Cypress's serial interface building blocks are the CY7B923 HOTLink Transmitter and CY7B933 HOTLink Receiver. These devices provide data rates of 160 to 330 Megabits/sec (16 to 33 Megabytes/sec) and conform to several communications standards.

This application note focuses on utilizing HOTLink to move data using a simple protocol. A block diagram of a typical HOTLink interface is shown in *Figure 1*.

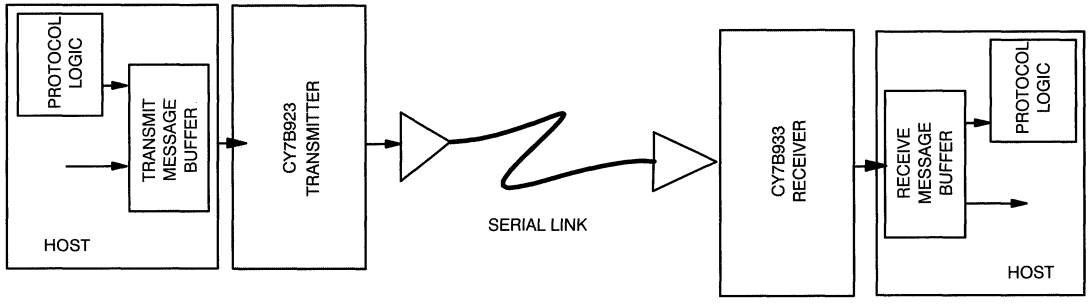


Figure 1. HOTLink System Connections

Preliminaries

For this application we will assume that the serial links in question will not exceed three or four feet. This length is adequate for most intra-board and board-to-board communications situations; and limiting ourselves to these distances removes several communications system issues from those that must be considered. Let's now discuss the general features of the HOTLinks.

In Encoded Mode, the HOTLink has an 8-bit parallel interface. Data bytes are encoded into 10-bit transmission words using 8B/10B encoding. In Bypass mode, HOTLink uses a 10-bit interface. 10-bit words bypass the encoder and go directly to the serializer. The 8B/10B code provides enough signal transitions on the serial interface to ensure proper PLL operation. It is also DC balanced, which prevents development of DC offset in the link over time. DC offset can result from more 1s being transmitted than 0s, so the encoder maps the 8-bit input word to multiple 10-bit output values to keep the number of 1's and 0's constant over time. Ideally, the time-averaged DC component should be zero, since DC offset over a long cable can cause increased noise susceptibility and power dissipation. In fiber systems excessive DC offset can burn out the LEDs used to drive the fiber.

The applications in this note use 8-bit Encoded Mode. In this mode HOTLink provides two control pins. A pin called SC/$\overline{D}$ indicates whether the byte on the parallel I/O pins is a special character or data. Another pin available in 8-bit mode, SVS (Send Violation Symbol), allows the data provider to force a violation symbol to be encoded and sent. The SC/$\overline{D}$ pins will be used to signify command words in the DMA protocol, which will be specified later. The SVS (RVS in the receiver) pin could be used for system testing and error checking, but will not be part of the design.

Parallel Interface

For details of the HOTLink parallel interface, please refer to the FIFO-HOTLink application notes located in this book.

Transmitter

The signals needed for the transmitter parallel interface are the 8-bit parallel inputs D(0..7), the SC/$\overline{D}$ bit, the $\overline{ENA}$ pin, the $\overline{RP}$ pin, and the CKW pin. Refer to the CY7B923/CY7B933 HOTLink datasheet for additional details.

When no data is enabled into the transmitter, it should be noted that the HOTLink Transmitter inserts a special character called SYNC. This SYNC character provides sufficient transitions to keep the PLLs locked to the bit stream.

Receiver

The signals needed for the receiver parallel interface are the eight parallel data outputs D(0..7) and the SC/$\overline{D}$, $\overline{RDY}$, and CKR pins. Again, refer to the device datasheet for additional details.

When the transmitter is sending SYNC characters, the receiver detects these and does not output this character until the last SYNC character is received. Then the receiver outputs a single SYNC character.

Serial Interfaces

Figure 2 shows the multiple serial outputs of the transmitter and the dual serial inputs of the receiver. OUTC is always on and in full duplex implementations, can be "looped back" to a receiver input for system diagnostics or used as another output. The other pair of outputs, OUTA and OUTB are enabled with the FOTO pin. This output pair makes it easy to transmit from one source to multiple destinations, making point-to-multi-point DMA architectures possible. The receiver, with its pair of inputs, can use one input channel for data and the second to implement local loopback. The input selection is accomplished with the A/$\overline{B}$ pin. Note that the INB channel does not have to be used for diagnostics, but can be used as another data stream input. However, switching input channels requires the PLL to reacquire lock with the incoming data stream.

Implementing a Data Link

The discussion that follows deals with issues confronting a designer trying to move data from point

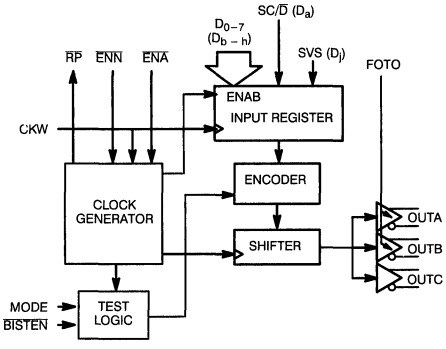


Figure 2a. CY7B923 Transmitter Logic Block Diagram

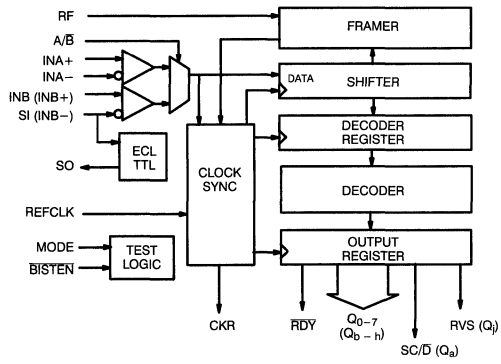


Figure 2b. CY7B933 Receiver Logic Block Diagram

A to point B using HOTLink. *Table 1* shows the three implementations discussed.

I/O Space Model

The first example is simple. It assumes that the receive FIFO resides in the destination's I/O space. The receive controller (a microprocessor, perhaps) merely reads and interprets the data stream out of the Rx FIFO. Data does not get placed in local memory before being used. There are two issues to consider with this example: What if the receive logic cannot keep up with the received data? This is known as receiver overflow or transmitter overrun. A FIFO with a programmable almost full/empty flag can be used together with a PLD to generate a receive or transmit inhibit to the transmit controller. This is known as "flow control". *Figure 3* shows a receiver block diagram with a flow control signal labeled TXINH. If the FIFO becomes too full, this signal tells the transmitter to stop transmitting until the receiver catches up. Since we are limiting our links to three or four feet, this may be a viable approach. However, using this form of flow control wastes a lot of bandwidth. Correct sizing of the FIFO, after careful analysis of the communications requirements, can give a deterministic system that never overflows or underflows. Communications links of hundreds of feet, or even miles cannot afford this type of flow control, since the channel itself may be several hundreds or even thousands of bytes long and a large enough FIFO may not be available. The channel is like a pipeline, and once something enters the pipeline, it must come out the other end.

Table 1. Data Link Implementations

I/O Model	Transmitter	Receiver	Features
I/O Space	FIFO+HOTLink	HOTLink+FIFO+ Microprocessor	Rx FIFO in I/O Space Microprocessor Accesses Data
Direct Memory Access	FIFO+HOTLink	HOTLink+FIFO+ DMA Logic	Rx Controller Decodes DMA Info in Rx FIFO DMA Moves Data Microprocessor Free Local Bus Used
Shared Memory Space	FIFO+HOTLink	HOTLink+DUALPORT+ DMA Logic	Rx Controller Uses Semaphores to Move Data Directly to Shared Memory Microprocessor Free/Local Bus Free

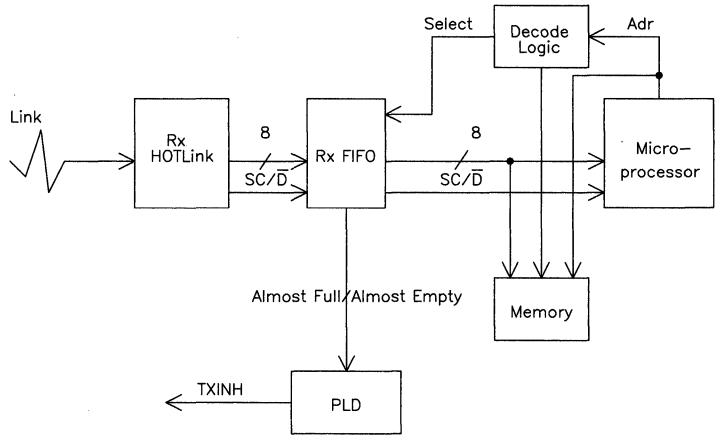


Figure 3. Receiver Flow Control

The second issue is that the microprocessor must be interrupted when data needs to be read from FIFO. This wastes microprocessor cycles and required lots of latency.

Direct Memory Access Model

So far we have discussed moving data from Point A to Point B using a microprocessor. Direct Memory Access (DMA) uses additional hardware, called a DMA controller or DMA Logic, to move the data from the FIFO to the memory. This frees the microprocessor of this task. Before proceeding, let's look at the bandwidth supported by HOTLink. This will determine the speed at which our DMA logic must operate. Refer to *Table 2.*

Table 2. Bandwidth Requirements

Part Number	Bandwidth (MByte/s)	Clock Period (ns)
CY7B923/933	16 – 33	63 – 30

Table 2 shows that DMA is probably a better solution than the I/O space model. The DMA Logic contains several basic functions. These are:

- control state machine
- address counter
- address (and sometimes data) latches and drivers

The control state machine detects when the receive FIFO contains data and issues a DMA request to the microprocessor. The DMA request asks the microprocessor to relinquish the memory address and data buses. The state machine also detects when the microprocessor has freed the buses. It then starts the actual transfer by loading the address counter with the initial address and placing the initial address and data word on the memory address and data buses. The control state machine then strobes the data into the memory, increments the address counter, reads the next data from the receive FIFO onto the memory data bus and strobes this data into the memory. The process then repeats until all the data has been placed in memory. When all the data has been placed in memory, the memory address and data buses are returned to the microprocessor's control. A block diagram is shown in *Figure 4.* Questions at this point might include, "Where did the starting address come from?", or "Where did the ending address come from?" To answer these questions, a protocol needs to be defined.

DMA Protocol Definition

Let's now discuss some concepts being introduced with our DMA protocol definition. First, we are now embedding command and control information in the data stream. Previously the information consisted of pure data (as far as our control logic was

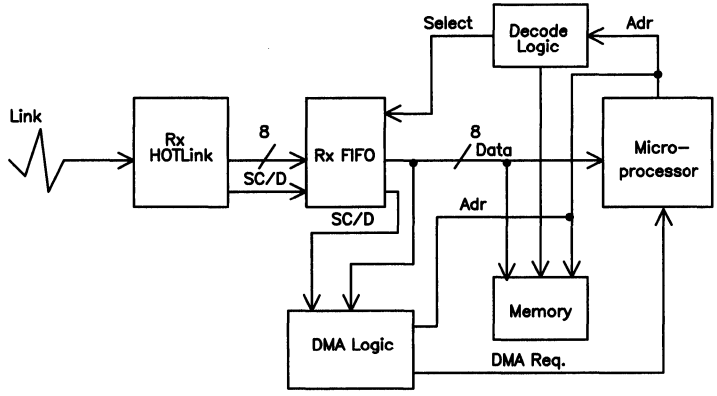


Figure 4. DMA Configuration

concerned). Second, we are now using dedicated hardware (a PLD) to move the data from the rx buffer to the location where it will be used, thus off-loading the main processor. Since the same protocol definition will be used for the shared memory implementation, let's define a protocol. First, the design will assume a fixed length DMA of 256 words. (The user is free to implement any length required, or to provide for variable length transfers.)

Table 3 defines a DMA Write message. The protocol will consist of a special character or message delimiter signifying a DMA write request, followed by characters defined as a broadcast address, and a starting address. A broadcast address can be thought of as a card or processor ID. The starting address indicates the first address to be written. This is followed by N-bytes of data, where N is equal to 256 in the example.

DMA reads will be identified by a unique message delimiter as shown in *Table 4*. Again, it is assumed that 256 bytes of data will be sent. The message defined in *Table 4* tells the recipient to send the 256 bytes of data beginning at the address indicated, and it also provides a destination address that can be used to create the DMA write message.

Finally, to assure proper initialization of the DMA hardware, *Table 5* defines a DMA Reset message.

The column labeled "Bits" in *Tables 3*, *4*, and *5* deserves further explanation. The labels HGF EDCBA are the 8B/10B designations for bits on the HOTLink parallel interface. Conventional notation for these bits is Q7..Q0 on the receiver outputs and D7..D0 on the transmitter inputs, with bit 7 being the most significant bit. In fact these signals correspond to the pins labeled identically on the HOTLink devices. The message delimiter characters are named according to Fibre Channel convention. Refer to the CY7B92X/CY7B93X datasheet for additional information.

The receiver DMA Logic in *Figure 4* needs to contain a state machine to detect the appropriate message delimiters and decode the broadcast address. If the broadcast address is for the module and the message is a DMA write, another state machine needs to issue a DMA request to the microprocessor and obtain the bus. After obtaining the bus, the starting address is read from the FIFO and loaded into an address counter. Since the address is defined as 32 bits, and the message length is defined as 256 bytes, the address must be loaded into 3 latches and an 8-bit counter. Then the state machine to reads the next 256 bytes out of the receive FIFO and writes it to memory. Then the bus is relinquished to the microprocessor and the counters and state machine are reset.

When a the message is a DMA read, the state machine is similar to that for a DMA write, but the ad-

dress counter is loaded with the address to read from, and the destination address is read out to be placed into a DMA write message. Creation of DMA write messages can be accomplished with additional resources in the DMA Logic. Suggested devices are the Cypress CY7C375 CPLD or the Cypress CY7C385 FPGA.

Table 3. DMA Write Message

SC/$\overline{D}$ Pin	Byte Name	Bits (HGF EDCBA)	Definition
1	K28.1	000 00001	Msg. Delimiter
0	8-bit address	–	Broadcast Address
0	Address byte 0	–	Most significant
0	Address byte 1	–	Next most signif.
0	Address byte 2	–	Next least signif.
0	Address byte 3	–	Least significant
0	Data byte 0	–	1st data byte
0	Data byte 1	–	2nd data byte
0	:	:	:
0	Data byte N	–	Last data byte
1	K28.1	000 00001	Msg. Delimiter

Table 4. DMA Read Message

SC/$\overline{D}$ Pin	Byte Name	Bits (HGF EDCBA)	Definition
1	K28.0	000 00000	Msg. Delimiter
0	8-bit address	–	Broadcast Address
0	Source Address byte 0	–	Most significant
0	Source Address byte 1	–	Next most signif.
0	Source Address byte 2	–	Next least signif.
0	Source Address byte 3	–	Least significant
0	Dest. Address byte 0	–	Most significant
0	Dest. Address byte 1	–	Next most signif.
0	Dest. Address byte 2	–	Next least significant
0	Dest. Address byte 3	–	Least significant
1	K28.0	000 00000	Msg. Delimiter

Table 5. DMA Reset Message

SC/$\overline{D}$ Pin	Code Name	Bits (HGF EDCBA)	Definition
1	K28.3	000 00011	Msg. Delimiter
0	8-bit address	–	Broadcast Address
1	K28.3	000 00011	Msg. Delimiter

The DMA model offloads the data moving task from the microprocessor. However, DMA has one major disadvantage; it requires the bus, which means the microprocessor must be idle during the DMA. There is another approach and it is the next topic.

Shared Memory I/O Model

If a shared memory area (dual-ported) is implemented, then data can be made available to the local logic without grabbing the microprocessor bus to perform a DMA. Simultaneous accesses must be prevented, but dual-ported memory opens up several options. These options include dividing the dual-ported memory into segments and alternating the segments between DMA write and local side read. This is known as "ping-ponging" and prevents simultaneous access of a dual-port SRAM location. So, dual-ported memory is attractive for those cases where the local bus cannot be tied up with a true DMA. *Figure 5* shows a diagram of a HOTLink communications link implemented with dual-ported SRAM. The DMA Logic is virtually identical to that of the prior section.

Summary

This application note has presented the basic concepts for employing HOTLink high-speed serial communications devices to replace parallel data paths. It has also defined a simple protocol and described the logic necessary to implement the protocol. Finally, the advantages and disadvantages of three different approaches have been presented to allow the designer to choose the one that best fits their needs. The simplest is Memory Mapped I/O, the highest performing is the Shared Memory I/O model employing dual-ported memory.

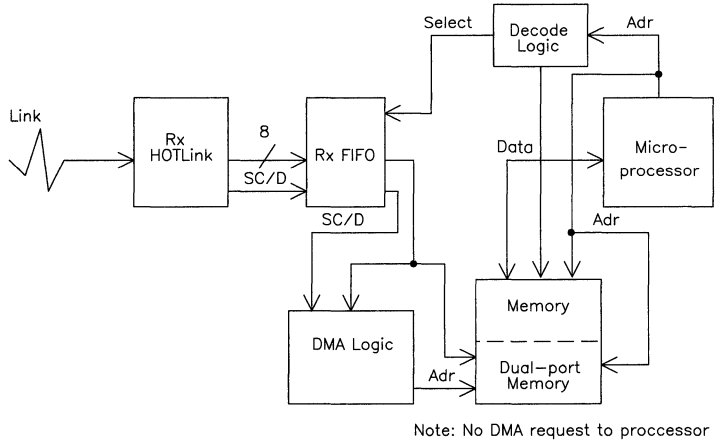


Figure 5. Dual-Ported Configuration

HOTLink is a trademark of Cypress Semiconductor Corporation.

Drive ESCON™ With HOTLink™

Introduction

The IBM® ESCON™ (Enterprise System CONnection) interface is presently experiencing rapid growth. Originally designed as a replacement for the older block-mux channel, it is also finding use as a high-performance system interface. This once IBM-proprietary interface is presently being processed to become an ANSI standard interface (known as SBCON) for computer to peripheral interconnect.

This application note contains an overview of ESCON operation and a design example of an ESCON physical interface, including a number of the low-level ESCON state machines (including the VHDL source code), implemented using HOTLink™ and a pASIC™ field programmable gate array.

Channels

The term *channel*, when referring to mainframes, carries a specific meaning. Rather than representing the connection between pieces of equipment, here it also represents a significant piece of equipment as well. The channel is, in effect, a sophisticated and intelligent DMA engine whose purpose is to move information between I/O devices and main storage. This channel function removes the burden of handling I/O activities from the main CPU.

Block-Multiplexer Channel

The original block-multiplexer channel dates back to the System 360/370 family of IBM mainframe CPUs. It uses a pair of parallel-bus copper cables (referred to as Bus and Tag cables) to move data between the host CPU and the I/O and storage peripherals as shown in *Figure 1*. These bus and tag cables were daisy-chained from the host channel adapter through multiple storage and I/O directors.

While quite powerful in its day, the block-mux channel shows both its heritage and its age. The bus and tag cables are quite bulky (around 1.5″ in diameter), heavy, *and* costly. The maximum length of the link between the host CPU channel adapter and the cable terminator is 400 feet, and operates at a maximum transfer rate of 4.5 MBytes/second. While originally designed to simultaneously support a larger number of peripherals, its is now possible to saturate the full I/O bandwidth capability of a block-mux channel with a single disk drive.

ESCON Channel

The ESCON channel was introduced in 1990 along with the ESA390 series of mainframe computers. It uses high-speed serial, point-to-point fiber-optic links to replace the daisy-chained parallel-bus copper cables of a block-mux channel. By maintaining the same host CPU software structures used with the block-mux channel, it was possible to dramatically change the architecture (and performance) of the I/O subsystem without effecting the major I/O routines present in the host CPU and channel microcode.

This new interconnect media was also merged with a dynamic switched connection scheme to improve both availability and access to the I/O peripherals. The use of switches (known as *directors*) allows many more paths to each peripheral, with multiple paths being active through each director at the same time. This new interconnect structure is shown in *Figure 2*. This switched I/O structure is now finding popular use in many other data communications in-

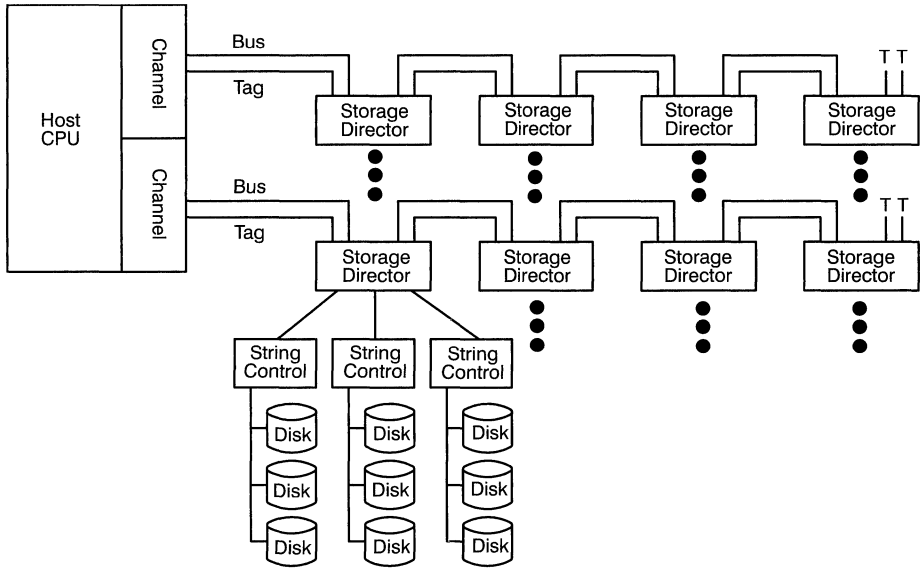


Figure 1. Block-Multiplexer Channel Subsystem

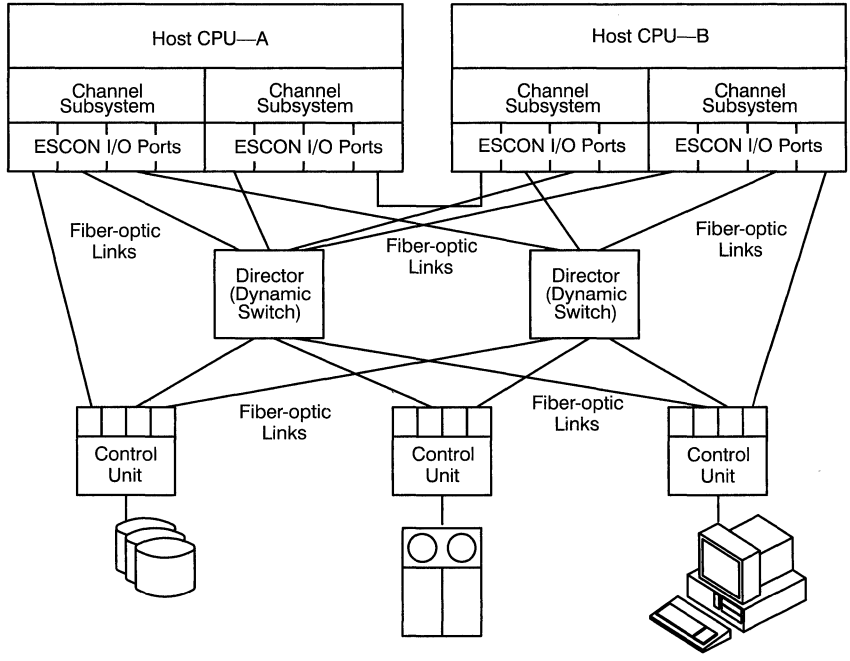


Figure 2. ESCON Channel Subsystem

terfaces like switched Ethernet, ATM, and Fibre Channel.

The ESCON interface provides numerous improvements over the older block-mux channel. A few of these are

- Improved transfer rate to 20 MBytes/second
- Longer distances—up to 3 km for each link and up to three links (two switches) between Channel and Control Unit
- Immunity from EMI/EMC concerns
- Improved access, redundancy, and availability through use of dynamic switches

ESCON Physical

The physical-level interconnections of ESCON are all made with 1300-nm LED-based optical links. These links operate through either 62.5 μm or 50 μm core multi-mode optical fibers at a fixed bit rate of 200 Mbits/second. This bit rate represents the encoded bit rate for the data being sent.

The data sent across ESCON links is encoded using the 8B/10B code built into HOTLink. (See the CY7B923/933 datasheet for a detailed description of the 8B/10B code.) This code converts normal 8-bit bytes into 10-bit transmission characters. While this encoding does have a 25% overhead, the benefits of using it far outweigh the data-rate penalty.

Part of the reason for the two extra bits in each character is to guarantee a minimum transition density for the receive PLL. Since no clock is present in the serial data, the HOTLink receiver PLL is used to extract a bit-rate clock from the data steam

Another benefit from this code is its DC-balance characteristic. This means that, over time, the net difference of all 1-bits versus 0-bits sent is at or near zero. This DC-balance characteristic allows the optical receiver circuits to be much simpler and lower in cost by reducing the complexity of the AGC (automatic gain control) in the receiver preamplifier.

With a transmission character being ten bits in length, there are actually 1024 possible transmission characters. Of these possible codes, only a fraction of them meet all the run length and DC-balance coding rules. The remainder are illegal codes and are detected as errors at the receive end of the link. Most of the valid codes are used to represent the 256 possible data bytes, with a few remaining legal transmission characters used for synchronization and in-band signaling.

The term *in-band* means that all delimiters, protocol, clocking, etc., are handled through the same serial interface as the data; i.e., there are no other control lines or interfaces used for this information. The 8B/10B code provides twelve transmission characters for these in-band functions. Of these twelve characters (referred to as special characters), only six are defined for use by ESCON.

Synchronization

With any serial interface some form of synchronization is necessary at the receiver-end of a link. The function of synchronization is to line up the receiver bit and byte clocks with the serial data stream.

Bit Synchronization

Bit synchronization is performed (for the most part) automatically by the receiver PLL. As transitions are detected, the phase detector in the receiver uses the position of the transition (relative to its internal bit-clock) to adjust the phase and frequency of the local bit-clock. This local bit-clock is optimally adjusted to allow the serial data stream to be sampled at the center of each bit. However, bit synchronization alone is not sufficient to recover and decode the transmitted information. This requires knowledge of which bit in the serial stream is the start of a character.

Framing

Proper detection of character boundaries is referred to as framing. Unlike bit synchronization, which occurs primarily in the analog domain, framing is a full-digital operation.

Framing is performed by examining the serial bit stream for a specific pattern (called a comma). This

test occurs on every bit-clock until an exact match is found. At this point the receiver byte-clock is reset to line up with the character boundary. Following this, all characters output from the receiver should remain properly synchronized, until some external event causes a significant disruption in the data stream.

The comma in the 8B/10B code is the seven bit pattern 0011111 (or its alternate 1100000). This bit pattern is part of the K28.5 special character. It cannot appear in any other location in any 8B/10B encoded character, and cannot be generated across the boundaries of any pair of characters.

While the detection of individual bits is controlled automatically by the PLL, the detection of framing for ESCON must be under the control of a separate state machine. This machine determines under what conditions the receiver is allowed to perform its framing function.

ESCON Synchronization

An ESCON interface is normally considered to be in one of two states regarding synchronization; either Synchronization_Acquired or Loss_Of_Synchronization (LOS). The transitions between these two primary states actually involve a number of substates that track error conditions and special characters on the interface. This state machine is shown in *Figure 2.*

In addition to its five states (four Sync Acquired and one Loss Of Sync), it operates with a 4-bit counter to track both valid characters and K28.5 characters. Since in any specific state of the machine only one thing is being counted (valid characters or K28.5 characters), only a single counter is needed.

Loss Of Synchronization

The ESCON interface automatically enters the LOS state following power-on. In this state (if a valid signal is present) the serial data receiver is enabled not only to received data, but also to frame on any received K28.5 character (RF=1).

While the receiver will frame on the first K28.5 received, this is not sufficient to leave the LOS state.

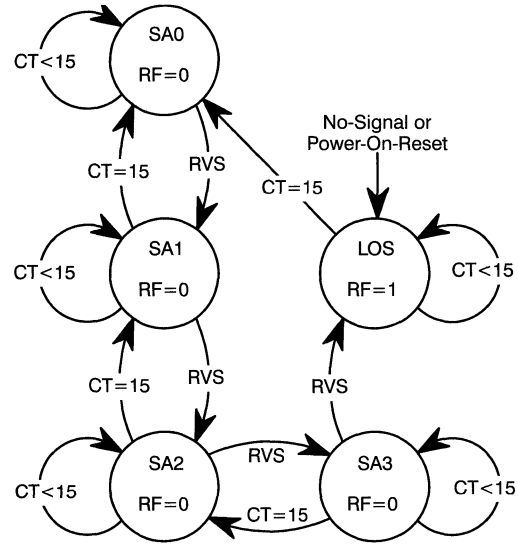


Figure 3. Synchronization State Machine

This requires reception of a minimum of fifteen K28.5 characters with no intervening code violations between any of the received characters. These K28.5 characters may be directly adjacent or more likely will have other characters interspersed. Once this string of K28.5 characters has been received, the receiver enters the Synchronization_Acquired state.

Synchronization Acquired

Exit from the LOS state also removes the reframe signal from the receiver (RF=0). In this condition the receiver will ignore (for framing purposes) all K28.5 characters embedded in the data stream. These characters are still properly received and decoded for use as part of the link protocol.

In the Sync Acquired state the state machine now tracks any code violations (RVS). If a code violation occurs the state machine changes from the basic Sync Acquired state (SA0) to SA1. In this state the machine has now detected a single error. It then enables the separate 4-bit counter to check for consecutive valid characters. If the following fifteen characters are received without error, the machine reverts back to the basic Sync_Acquired state.

If, however, additional character errors are detected, the state machine will advance through the SA1, SA2, and SA3 states—one change for each character received in error. At each of these states the machine will again check for valid characters and will revert to the previous state if fifteen are received without any errors. This would allow an interface receiving exactly one error every sixteen characters to remain in the SA0 and SA1 states, while a similar interface receiving one error every fifteen characters would quickly move to the LOS state and remain there.

Link-Level Operations

The actual functionality of an ESCON link is defined in terms of various ordered sets of special characters and data bytes. These ordered sets are used to define frame boundaries, control dynamic connections, and control synchronization between the transmitter and receiver circuits. All valid ESCON ordered sets are listed in *Table 1*.

Table 1. ESCON Ordered Sets

Ordered Set	Characters
Idle function	K28.5
Connect-start-of-frame delimiter	K28.1 K28.7
Passive-start-of-frame delimiter	K28.5 K28.7
Abort delimiter	K28.6 K28.4 K28.4
Disconnect-end-of-frame delimiter	K28.6 K28.1 K28.1
Passive-end-of-frame delimiter	K28.6, K28.2 K28.5
Not-operational	K28.5 D0.2
Unconditional-disconnect sequence	K28.5 D15.2
Unconditional-disconnect-response sequence	K28.5 D16.2
Off-line Sequence	K28.5 D24.2

Idle Function

The K28.5 character in ESCON is used for multiple purposes. It is

- the first character of many ordered sets
- used to provide byte framing of the serial data stream
- used as a fill or *Idle* character between frames and sequences

Because the K28.5 character is contained in many of the other ordered sets, a single K28.5 cannot be conferred to be an Idle function until the following character is detected. If the following character is also an K28.5, then the previous K28.5 is part of an Idle Function. If the following character is anything else, then the K28.5 character is part of a delimiter or sequence (or an error).

Delimiters

Delimiters are used to mark the start and end of frames. Frames are the real workhorse of the interface because they carry data. All frames have a start-of-frame delimiter (SOF) and an end-of-frame delimiter (EOF). (An Abort delimiter is considered to be a type of EOF.) These delimiters are only sent once per frame. Each frame must be separated by a minimum of four Idle characters.

Sequences

Sequences are used to indicate specific equipment conditions or states that cannot be identified through the use of frames. Unlike a delimiter, the ordered set defined for a specific sequence is sent repeatedly until the machine state changes or a specific response is received. At the receiver, a sequence is only detected as being valid if the defined ordered set is received a specific minimum number of times in succession.

Frames

Frames are used to carry information between the channel, switches, and the peripherals. Two generic types of frames exist; Link-Control and Device Level.

All frames follow the same three-field format:

- a 7-byte fixed-length link header
- a variable-length information field (may have a length of zero for some Link-Control frames)

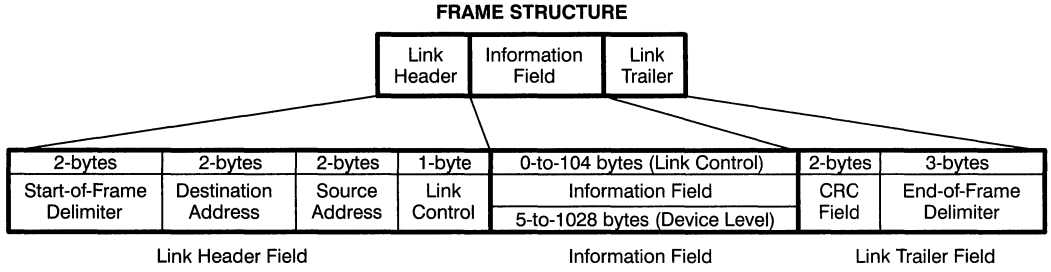


Figure 4. ESCON Frame Format

- and a 5-byte fixed-length link-trailer field

The structure of an ESCON frame is shown in *Figure 4*. The low-order bit of the Link Control field in the Link Header identifies the type of frame. When set to a one, the frame is a Link Control frame. When set to a zero, the frame is Device Level frame.

Link-Control frames are use to manage, configure, and maintain the link itself, and range in length from 12 to 116 bytes. Device Level frames carry data between the channel and the peripheral and range in size from 17 to 1040 bytes.

Frame Validation

Before a frame can be processed, it must be validated as a properly received frame. This involves making sure that there are no special characters or idles in the middle of the frame, no decoding errors are detected in the serial stream, and that the CRC Field (Cyclic Redundancy Check) shows no errors.

Cyclic Redundancy Check Field

The CRC field contains a 16-bit redundancy check code, used to insure that the received frame contents are the same as those sent. This field is generated at the transmitting end of a link and sent as the first two bytes of the Link Trailer field. It is calculated on all bytes between the start-of-frame delimiter and the Link Trailer field.

At the receiving end of the link the CRC is again generated using the received data stream. Now the CRC is generated on all bytes between the start-of-frame delimiter and the end-of-frame delimiter.

The CRC code used with ESCON is that defined by the ITU V.41 standard (previously known as CCITT). The polynomial for this CRC is listed in *Equation 1*.

$$x^{16} + x^{12} + x^5 + 1 \qquad \text{Eq. 1}$$

Normally with a code of this type the CRC remainder register is preset to an all 1s condition prior to the first bit of information being clocked through the polynomial. This is done to ensure that the polynomial will change state no matter what the data stream contains. At the end of the generation, the two bytes comprising the CRC remainder are sent as part of the data stream. At the receiving end the same process occurs, but the two CRC bytes are also clocked into the CRC register. If no errors exist in the serial stream then the contents of the CRC check register should be zero.

To increase the level of protection, the CRC is handled slightly differently in an ESCON interface. Here the CRC remainder generated at the transmitter is inverted prior to sending it across the link. When it is received (correctly) the CRC check register is no longer cleared, but must be set to exactly 1D0F (hexadecimal). Any other value indicates a transmission or reception error.

ESCON Design Example

The following design was originally done to replace an existing ESCON protocol component that was no longer available. All VHDL source code listed here has been both simulated and tested in a functioning ESCON system.

This design example covers

- an ESCON-compatible optical media interface
- ESCON-certified HOTLink serializer/deserializer components
- a pASIC383 protocol chip containing
  - — transmit and receive CRC circuits
  - — parity check and generate circuits
  - — synchronization state machine
  - — command code translation capability
  - — input/output pipeline registers
  - — miscellaneous flip-flops, muxes, and gates

The design is partitioned into transmit and receive data paths, and is implemented in four active devices:

- a pASIC383 containing both transmit and receive protocol functions
- a CY7B923 HOTLink transmitter for serialization and 8B/10B encode
- a CY7B933 HOTLink receiver for deserialization and 10B/8B decode
- a Siemens V23806–A1–M16 ESCON fiber-optic transceiver

The structure of how these components connect and major data paths are shown in *Figure 5*, with a complete schematic shown in *Figure 6*.

Fiber-optic Transceiver

The fiber-optic transceiver is an optoelectric device that both converts electrical signals to light (transmitter) and light into electrical signals (receiver).

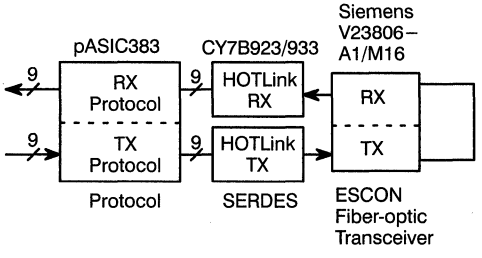


Figure 5. Design Example Structure

To operate with the ESCON interface the transceiver must meet a number of specific characteristics:

- operate at 200 Mbaud
- operate at 1300 nm wavelength
- use 62.5-μm or 50-μm core optical fiber
- meet the 0.7″ ferrule spacing and other dimensions of an ESCON optical connector

In addition to these criteria, compliant transceivers must meet numerous power level, receive sensitivity, and electrical interface criteria to properly operate in an ESCON environment. Manufacturers of ESCON compatible fiber-optic transceivers include Siemens, AMP, IBM, and others.

SERDES

The next section in an ESCON link is the serializer/deserializer block, also known as the SERDES. This section converts parallel bytes of information into an 8B/10B encoded serial data stream for transmission, and also converts a received 8B/10B encoded serial data stream back into parallel data bytes.

The Cypress CY7B923/933 HOTLink components are designed to perform this SERDES function. These components are specifically optimized to support the ESCON interface, as well as Fibre Channel, ATM (Asynchronous Transfer Mode), and proprietary communications links.

These HOTLink parts are especially well suited to the ESCON market because of their built-in 8B/10B encoders and decoders. This encode/decode function is required for ESCON operation. By building the encode/decode into the SERDES block, the complexity of this part of the interface design is removed from the design process. Its presence in the SERDES block also means that hardware resources are not required elsewhere to implement the encode/decode function.

The 8B/10B code used in the HOTLink components is licensed by Cypress Semiconductor from IBM. Any user of these parts is fully licensed to use the 8B/10B encoders and decoders contained in them at no cost and no royalties. For those applications that already have 8B/10B encoder/decoder circuits pres-

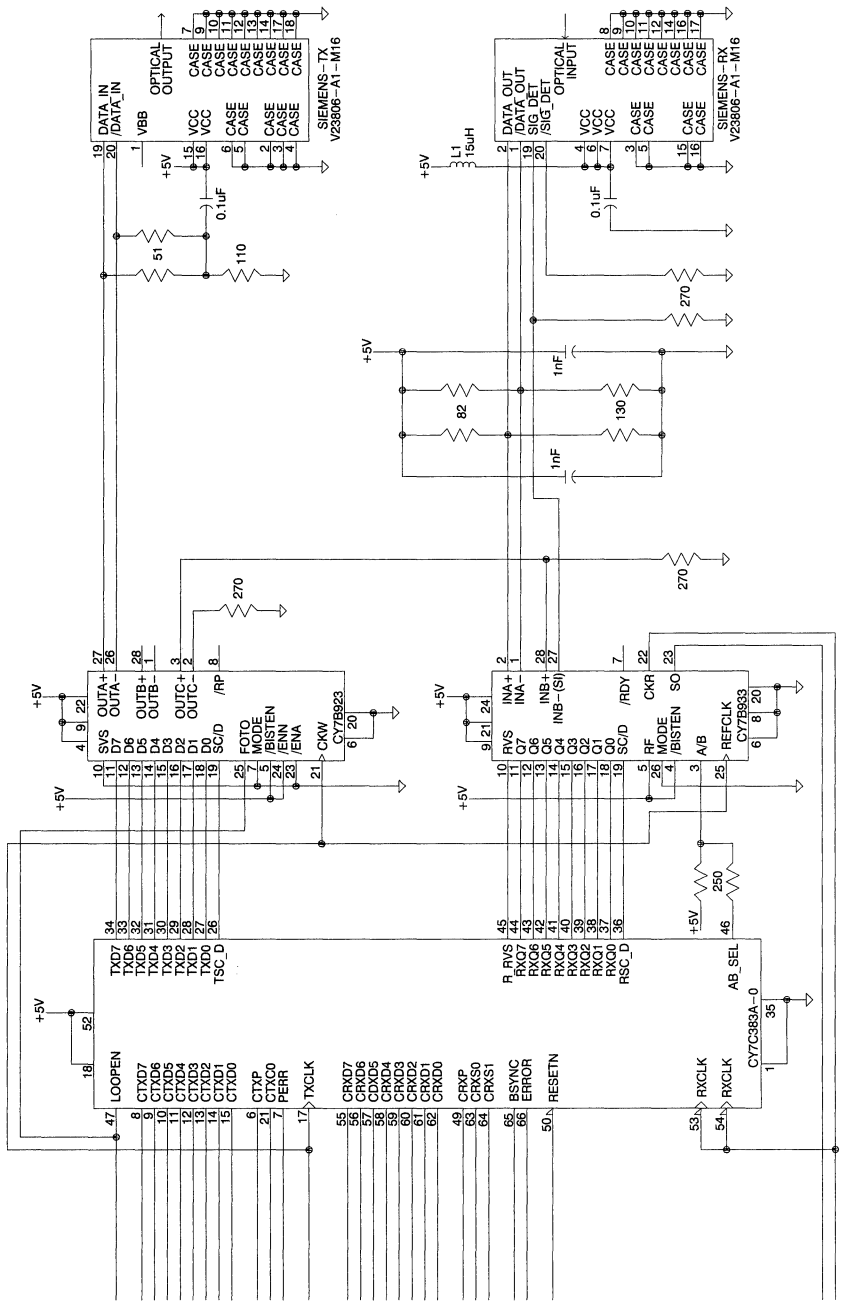


Figure 6. ESCON Physical I/O Interface Schematic

ent in their system, the encoder/decoder present in HOTLink can be bypassed through use of the MODE pin on each part.

An in-depth explanation of the operation and usage of the HOTLink components may be found in the CY7B923/933 datasheet and the *HOTLink User's Guide*.

Serial I/O Electrical Interface

The interface between the fiber-optic transceiver and the HOTLink SERDES operates at 200 Mbits/second. This interface is implemented with ECL (Emitter-Coupled-Logic) signaling to provide a low-noise, high-speed connection. Unlike standard ECL, which is normally operated below ground, both the fiber-optic transceiver and the HOTLink SERDES components are operated above ground. This allows the ECL portion of the design to use the same +5V supply as the surrounding logic. When ECL is operated from a positive supply it is referred to as Positive-ECL or PECL.

The source for the serial data stream is the CY7B923 HOTLink transmitter shown in *Figure 6*. A simplified schematic showing just the interconnect for the serial transmit path is shown in *Figure 7*.

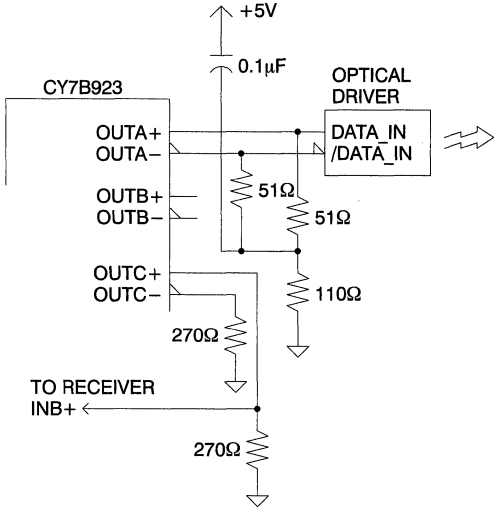


Figure 7. HOTLink Transmitter-to-Optical Serial Interface

The serial data is connected to the fiber-optic transmitter using a differential connection from the OUTA± differential output of the HOTLink transmitter. Because these are ECL/PECL signals, they require a pull-down bias to allow the outputs to switch.

With a transmission rate of 200 Mbits/second, the interconnect used for these signals should (in most cases) be constructed as a controlled-impedance transmission line. The bias network used on the OUTA± signals is referred to as a Y-bias network. It is designed to provide an equivalent transmission line termination impedance of 50Ω while providing a bias level of $V_{CC}-2V$.

The received serial data stream is output from the fiber-optic receiver as a differential signal, as shown in *Figure 6*, and is sent to the CY7B933 HOTLink receiver INA± inputs. A simplified schematic showing just the interconnect of the serial receive path is shown in *Figure 8*. Because this is also a PECL signal, it should be treated in a manner similar to the transmit serial path. This means controlled impedance transmission lines and a proper bias/termination network.

While the receive-path bias/termination network may be implemented using the same Y-bias network used with the transmit serial path, a Thévenin network is shown here. These two bias networks, when used with differential signals, are effectively interchangeable. For single-ended signals requiring the

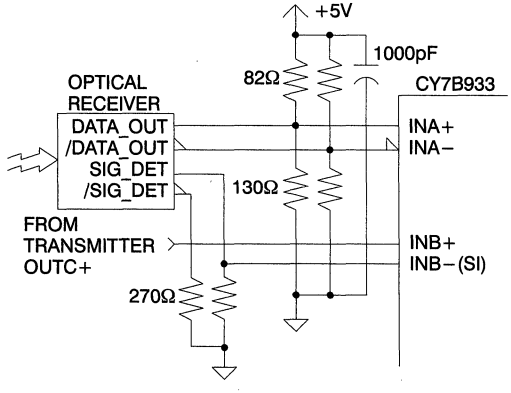


Figure 8. Optical-to-HOTLink Receiver Serial Interface

same electrical characteristics, the Thévenin network must be used. For additional information on terminating and biasing PECL signals, please see the application note "HOTLink Design Considerations" in the *HOTlink User's Guide*.

Serial I/O Support Interface

In addition to the transmit and receive serial data streams, two other PECL signals are normally present in an ESCON interface: signal-detect and local-loopback. The signal-detect function is performed by the fiber-optic receiver. It outputs a PECL logic signal to inform the upstream hardware if a valid signal is present or not. This signal is monitored to determine the synchronization state of the interface.

Because this is a PECL-level signal, it is necessary to convert it to a TTL-level signal for use by upstream logic. While there are components available that explicitly perform this level translation, they are not necessary for this application. Instead it is possible to use one of the design features of the HOTLink receiver INB± inputs to perform this signal-level conversion.

The INB± input can be configured as either a differential PECL receiver (like INA±), or as a single-ended serial PECL receiver and a PECL-to-TTL converter. To use INB± as a differential receiver it is necessary to pull the SO (Status Out) pin to V_{CC}. This disables the PECL-to-TTL converter and maintains both inputs as a differential pair.

To use INB± as two separate inputs requires that the SO pin be loaded as a normal TTL-level output. When configured this way the INB− pin is the input for the PECL-to-TTL converter, with SO being the TTL output. This is the configuration used in *Figures 6* and *8*.

Most ESCON interfaces are also equipped with numerous self-diagnostic capabilities. At the physical interface the most common is a selectable loopback of the serial data stream. This allows all components (with the exception of the fiber-optic transceiver) of the interface to be tested by transmitting data and verifying that it can be properly received. This loopback function is normally implemented using the OUTC+ output of the HOTLink transmitter and the INB+ input on the HOTLink receiver in a single-ended PECL connection, as shown in *Figures 6, 7,* and *8*.

While the best PECL connection is always a differential connection (like that used on INA±), the usage of INB+ in a single-ended mode is fine under these conditions. Because the HOTLink transmitter and receiver are close together in the system and operate from a common power supply, the normal noise-margin concerns of single-ended connections do not apply.

This local loopback functionality is selected through the LOOPBACK signal on the pASIC FPGA. When active (HIGH), this signal drives the HOTLink receiver A/$\overline{B}$ select input LOW to selected the INB+ input for the deserializer, and drives the FOTO input to the HOTLink transmitter HIGH. This FOTO pin is used to disable the OUTA± and OUTB± outputs of the transmitter. This is normally done during loopback diagnostics to prevent the diagnostic data from being interpreted at the other end of the fiber-optic link.

ESCON Protocol Controller

The control of the serial data stream is performed using a pASIC383 FPGA. This part has been programmed to manage both the transmit and receive serial data streams. The programming and verification were done using VHDL (VHSIC Hardware Description Language) using Cypress's *Warp3*™ logic synthesis and simulation tools. Complete source code of the design VHDL modules is listed in Appendixes A through H of this application note, and is available for download from the Cypress Bulletin Board system.

The design shown in this application note is effectively a logic replacement for a Triquint GA9104 ESCON protocol chip. Due to the flexibility of the pASIC family of parts, it is possible to add, replace, or remove logic that is not optimal for a specific application. In this design, the 8B/10B encoders present in the normal GA9104 were not implemented in the pASIC383 because they are already present in the HOTLink CY7B923/933. This allowed the entire functionality to be duplicated in a 2K-equivalent gate FPGA. The functions present in this design are

- Transmit Path
  - — input and output pipeline registers
  - — parity checker and status bit
  - — CRC generator and control state machine
  - — Command/data mux
  - — Command translator
- Receive Path
  - — input and output pipeline registers
  - — CRC checker, control state machine, and status bit
  - — parity generator
  - — Command/data mux
  - — Command translator
- Byte-Sync State Machine

Transmit Path

A block diagram of the transmit path is shown in *Figure 9*. Data is captured into a 10-bit register on each rising edge of the transmit clock (CKW). The data consists of an 8-bit data byte, a single control line (CTXC0), and a parity bit. The CTXC0 line is used to identify whether the data on the inputs is a command code (HIGH) or a data byte (LOW). If the latched character is a data byte, the data is simultaneously presented to the CRC register, the parity checker, and the output multiplexers. At the next rising edge of the transmit clock, this data byte is clocked into the CRC register, checked for proper parity, and loaded into the output register along with TSC_D set LOW.

The detection of a parity error is only a reported event, and occurs one cycle after the data (or command) is latched into the input register. Recovery from detected parity errors would normally require abnormal termination of the current frame using the Abort delimiter.

The CRC/MUX Control block is the heart of the transmit path logic. It monitors the CTXC0 line to determine when to

- preset the CRC register
- accumulate a CRC
- output the CRC bytes
- translate/send command codes

This block is implemented as a simple shift register that tracks the current and previous three states of CTXC0. These sixteen possible combinations (with don't care states removed) and their resulting outputs are listed in *Table 2*. The VHDL source code for this block is listed in Appendix C.

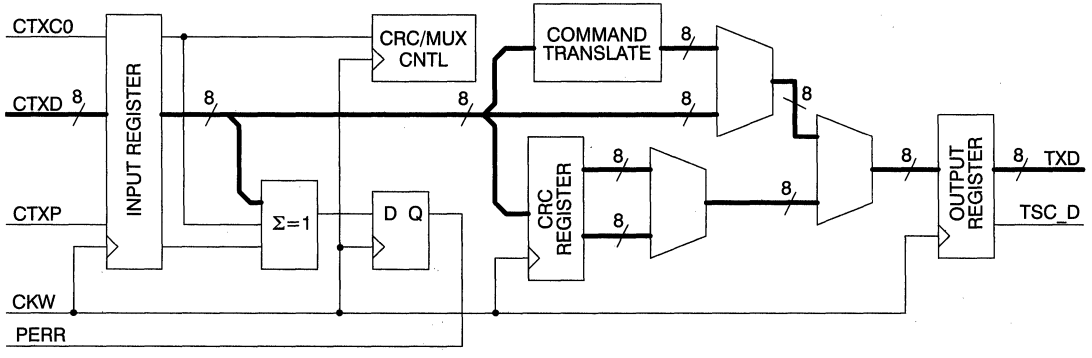


Figure 9. pASIC Transmit Path Block Diagram

Table 2. Transmit Path Control

CTXC0				Mux Select/ CRC Control
t+3	t+2	t+1	t+0	
X	X	X	0	Data
X	0	0	1	CRC High Byte
0	0	1	1	CRC Low Byte
X	X	1	1	Preset CRC
X	1	0	1	Command
1	0	1	1	Command

The CRC block implements the CRC-16 function in a byte-parallel fashion. This allows a full byte to be accumulated in a single clock cycle. While this does require a much larger number of XOR gates to implement than a serial CRC function, it allows the design to be constructed from much slower logic. Here the main CRC register is clocked at 20 MHz, rather than having to operate at a 200-MHz bit-clock rate. The VHDL source code for this function is listed in Appendix B.

The command-translate block is not normally needed for new designs. For this specific design it was necessary to translate an existing set of command codes to the native HOTLink command set. This translation is quite simple with the logic reduction performed manually for the transmit path. Here an 8-bit input command is decoded into a 4-bit command field (with the upper four bits of the byte set to zero).

The translation block actually implements circuitry to translate all twelve command codes in the 8B/10B character set. For ESCON implementations this logic could be simplified because only half of these (six) are actually allowed for use in ESCON ordered sets. The VHDL source code for this function is listed in Appendix D.

The last section in the transmit path is the output pipeline register. This block receives the multiplexed output of either the input pipeline register, the high-CRC byte, the low-CRC byte, or the translated command. It serves to keep the data presented to the HOTLink transmitter synchronous with the transmit clock.

Receive Path

A block diagram of the receive path is shown in *Figure 10.* Data is captured from the HOTLink receiver into the input register on each falling edge of the HOTLink recovered receive clock (CKR). Note that this could also be implemented using a rising edge clock, but that a falling edge clock was used for compatibility with the implementation being replaced.

All received data characters are clocked into the CRC register. Like the transmit path, this function is implemented in a byte-parallel form. The CRC register is synchronously preset if any command code is present in the input register. For all data codes it accumulates the CRC remainder.

The CRC register is constantly compared for the x'1D0F' pattern. The output of this compare is clocked into the output register. It is forced to a LOW for all clocks except the first command character received following a data character. This CRC status remains valid for only one clock cycle. The

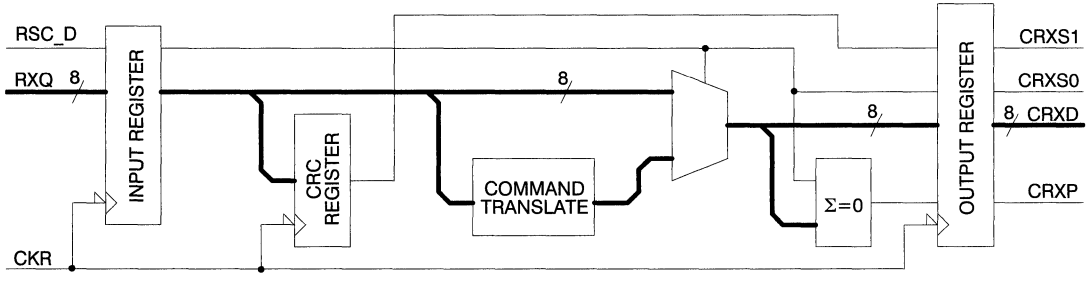


Figure 10. pASIC Receive Path Block Diagram

VHDL source code for this function is listed in Appendix E.

Just as in the transmit path, a command translation block is present in the design. This command translate block is not normally needed for new designs. For this specific design it was necessary to translate an existing set of command codes from the native HOTLink command set to a different set of command codes embedded in upstream logic. This block allows the HOTLink command codes to be translated to any host command set.

The translation block actually implements circuitry to translate all twelve command codes in the 8B/10B character set. For ESCON implementations this logic could be simplified because only half of these (six) are actually allowed for use in ESCON ordered sets. The VHDL source code for this function is listed in Appendix D.

Odd parity is generated on the output data byte and the CRXS0 status bit. This allows upstream logic to validate that the byte received is the same as that generated by the pASIC FPGA.

The last block in the receive section is the output pipeline register. This block receives the multiplexed output of either the input pipeline register or the translated command. It serves to keep the data presented to the upstream logic synchronous with the receive clock.

Byte-Sync State Machine

A block diagram of the byte-sync state machine is shown in *Figure 11*. The two primary structures in the machine are a 4-bit counter and a controlling state machine. The controlling state machine is programmed to follow the state diagram shown in *Figure 2*. It tracks the state of the RVS signal from the receiver and a decode from the input register of all C5.0 command codes (Idle characters). The four-bit counter is used to alternately count either valid characters (the absence of RVS) or valid Idle characters, based on the state of the machine.

The present form of this state machine was designed to duplicate the functionality of a previous implementation. Because of this it does not take into account the the additional condition of Signal Detected that is generated by the fiber-optic receiver. Sufficient I/O and logic resources are still available in the FPGA to add this into the state machine equations.

Design Summary

The small size of the FPGA design is made possible by the enhanced functionality present in the HOTLink transmitter and receiver. This removes the need to design and implement the 8B/10B encoders and decoders, and provides full received character validation. The embedded PECL-to-TTL converter also allows a small footprint by removing the need for an external conversion circuit.

The VHDL design both auto-routes and auto-places into a pASIC383 FPGA. Because of the high-speed operation of the pASIC cells and interconnect, this design meets or exceeds all design performance parameters, over worst case temperature and voltage, using the slow −0 speed bin of the pASIC383.

The 100% routability of the pASIC family allows the circuit board signal routing to be improved by selecting pins that best match the system interconnect. The pinouts listed in the top-level VHDL file were selected to allow straight-through routing (no crossovers) of the signals between the FPGA and the HOTLink transmitter and receiver. In addition, the

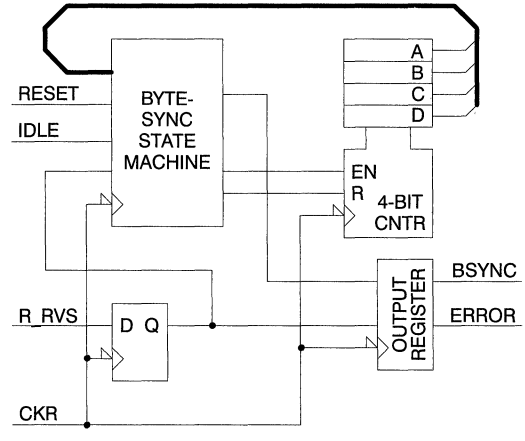


Figure 11. Byte-Sync State Machine Block Diagram

placement of the HOTLink transmitter and receiver were selected to line up with the transmit and receive halves of the fiber-optic transceiver. This pinout selection and interconnect are shown in *Figure 12*.

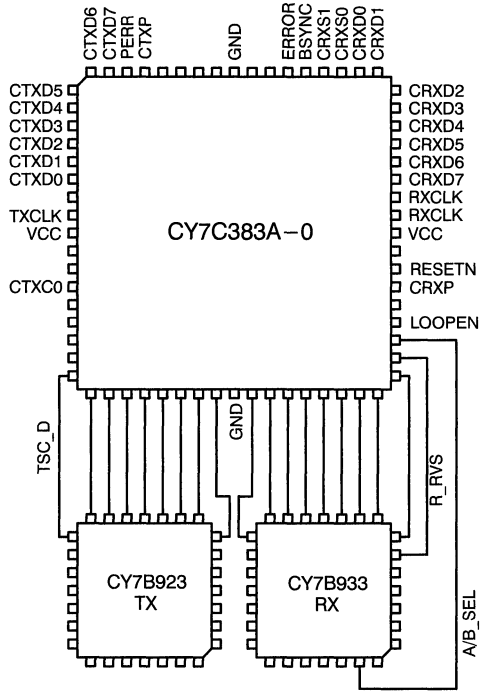


Figure 12. HOTLink/pASIC Pinout and Interconnect

Conclusions

The ESCON interface is both an elegant and powerful replacement for the older block-mux channels. The use of the HOTLink serializer/deserializer components to implement an ESCON interface guarantees both compliance with the 8B/10B coding rules and all jitter and timing specifications of the ESCON interface.

Due to the high-speed operation of the ESCON interface, the byte-level control is best implemented in hardware. The flexibility of the VHDL language and the unlimited routing of the Cypress pASIC family of FPGAs make them a perfect choice for building the control state machines. While only the lower level of the ESCON protocol is controlled in the design documented here, much of the higher level control may also be implemented through the use of either larger or additional FPGA components.

References

1. *ESCON I/O Interface*, IBM, 1990, 1991
2. *HOTLink User's Guide,* Cypress Semiconductor, 1995
3. *GA9104 Datasheet,* Triquint Semiconductor, Inc, 1992

Appendix A. Top-Level pASIC Code

```
-- ESC_TOP.VHD
--
-- ESCON Interface Control PLD
-- Equivalent to the Triquint GA9104 but designed for operation
-- with the Cypress Semiconductor HOTLink chipset
ENTITY esc_top IS PORT (
  txclk: IN BIT;                    -- transmit path byte clock
  rxclkA: IN BIT;                   -- receiver path byte clock
  rxclkB: IN BIT;                   -- receiver path byte clock
  resetn: IN BIT;                   -- active low reset
  rxq: INOUT X01Z_VECTOR(0 TO 7);   -- HOTLink receiver data in
  rsc_d: INOUT X01Z;                -- HOTLink receiver SC/D
  r_rvs: INOUT X01Z;                -- HOTLink receiver RVS
  txd: INOUT X01Z_VECTOR(0 TO 7);   -- HOTLink transmitter data out
  tsc_d: INOUT X01Z;                -- HOTLink transmitter SC/D
  crxd: INOUT X01Z_VECTOR(0 TO 7); -- receive path data output
  ctxd: INOUT X01Z_VECTOR(0 TO 7); -- transmit path data input
  crxs0: INOUT X01Z;                -- receive status 0 (command/data)
  crxs1: INOUT X01Z;                -- receive status 1 (CRC)
  ctxc0: INOUT X01Z;                -- transmit control 0 (command/data)
  bsync: INOUT X01Z;                -- byte sync acquired
  error: INOUT X01Z;                -- receive bad character error
  perr: INOUT X01Z;                 -- transmit-in parity error
  crxp: INOUT X01Z;                 -- odd parity output
  ctxp: INOUT X01Z;                 -- odd parity input
  loopen: INOUT X01Z;               -- local loopback enable
  ab_sel: INOUT X01Z);              -- receiver A/B select

ATTRIBUTE part_name OF esc_top:ENTITY IS "C383A";
ATTRIBUTE pin_numbers OF esc_top:ENTITY IS
    "txclk:17 rxclkA:53 rxclkB:54 resetn:50 rxq(7):44 rxq(6):43 "
  & "rxq(5):42 rxq(4):41 rxq(3):40 rxq(2):39 rxq(1):38 rxq(0):37 "
  & "rsc_d:36 r_rvs:45 txd(7):34 txd(6):33 txd(5):32 txd(4):31 "
  & "txd(3):30 txd(2):29 txd(1):28 txd(0):27 tsc_d:26 crxd(0):62 "
  & "crxd(1):61 crxd(2):60 crxd(3):59 crxd(4):58 crxd(5):57 "
  & "crxd(6):56 crxd(7):55 ctxd(0):15 ctxd(1):14 ctxd(2):13 "
  & "ctxd(3):12 ctxd(4):11 ctxd(5):10 ctxd(6):9 ctxd(7):8 "
  & "crxs0:63 crxs1:64 ctxc0:21 bsync:65 error:66 perr:7 "
  & "crxp:49 ctxp:6 loopen:47 ab_sel:46";

END esc_top;

USE work.cypress.all;
USE work.rtlpkg.all;
USE work.memorypkg.all;
USE work.ttlpkg.all;
USE work.registerpkg.all;
```

Appendix A. Top-Level pASIC Code (continued)

```
USE work.iopkg.all;
USE work.mcpartspkg.all;
USE work.gatespkg.all;
USE work.resolutionpkg.all;      -- used to double-buffer
USE work.bv_math.all;            -- allow use of INV function
USE work.crc_t.all;              -- add in CRC transmit function
USE work.crc_r.all;              -- add in CRC receive function
USE work.crc_ctl.all;            -- add in transmit CRC control machine
USE work.sync_det.all;           -- add in SYNC detect state machine
USE work.triq_code.all;          -- add in command decoder section
USE work.iopluspkg.all;          -- add in enhanced I/O buffers

ARCHITECTURE escon_top OF esc_top IS
-- add internal signal equivalents of signals after I/O pads
SIGNAL tclk : BIT;                    -- transmit clock
SIGNAL rclk : BIT;                    -- negative edge receiver clock
SIGNAL reset : BIT;                   -- reset controller
SIGNAL HL_rx : BIT_VECTOR(0 to 7); -- HOTLink receiver data bus
SIGNAL HL_rsc_d : BIT;                -- HOTLink receiver SC/D
SIGNAL HL_r_rvs : BIT;                -- HOTlink receiver RVS
SIGNAL HL_tx : BIT_VECTOR(0 to 7); -- HOTLink transmitter data bus
SIGNAL HL_tsc_d : BIT;                -- HOTLink transmitter SC/D
SIGNAL HL_tsc_q : BIT;                -- clocked HOTLink transmitter SC/D
SIGNAL sync_r : BIT;                  -- receiver byte sync
SIGNAL c_rxd : BIT_VECTOR(0 to 7); -- controller receive path data out
SIGNAL c_txd : BIT_VECTOR(0 to 7); -- controller transmit path dataout
SIGNAL c_rxs0 : BIT;                  -- receive status 0 (command/data)
SIGNAL c_rxs1 : BIT;                  -- receive status 1 (CRC)
SIGNAL c_txc0 : BIT;                  -- transmit control 0 (command/data)
SIGNAL b_sync : BIT;                  -- byte sync acquired
SIGNAL r_error : BIT;                 -- receive bad character error
SIGNAL p_err : BIT;                   -- parity error
SIGNAL c_rxp : BIT;                   -- odd parity output
SIGNAL c_txp : BIT;                   -- odd parity input
SIGNAL b_loopen : BIT;                -- buffered loop enable

-- transmit internal signals
SIGNAL t_data : BIT_VECTOR(0 TO 7);   -- transmit data bus
SIGNAL t_mux : BIT_VECTOR(0 TO 7);    -- muxed transmit data path
SIGNAL t_comm : BIT_VECTOR(0 TO 7);   -- re-encoded transmit commands
SIGNAL tp_odd : BIT;                  -- transmit data parity input
SIGNAL t_parity : BIT;                -- transmit parity checker output
```

Appendix A. Top-Level pASIC Code (continued)

```
SIGNAL t_CRC : BIT_VECTOR(0 TO 7);    -- transmit CRC vector
SIGNAL c_txc_0 : BIT;                 -- transmit command/data
SIGNAL mux_hi : BIT;                  -- enable HI/LOW transmit CRC byte
SIGNAL mux_low : BIT;                 -- enable LOW transmit CRC byte
sIGNAL ctxc3 : BIT;                   -- 3x registered c_txc_0
SIGNAL t_CRC_reset : BIT;             -- preset transmit CRC register
-- receive internal signals
SIGNAL r_data : BIT_VECTOR(0 TO 7);   -- registered receiver data bus
SIGNAL r_mux : BIT_VECTOR(0 TO 7);    -- muxed data and translated commands
SIGNAL rp_odd : BIT;                  -- receive data parity output
SIGNAL rcom_data : BIT;               -- registered SC/D pin
SIGNAL r_com_data: multi_buffer BIT;  -- double buffered registerd SC/D pin
SIGNAL r_crc_err : BIT;               -- un-registered CRC status
SIGNAL r_CRC_d : BIT;                 -- CRC check D-input
SIGNAL rvs : BIT;                     -- registered RVS signal
SIGNAL sync : BIT;                    -- decoded K28.5 signal
SIGNAL t_code : BIT_VECTOR(0 to 7);   -- Triquint pattern for K-codes

-------------------------------------------------------------------------
-------------------------------------------------------------------------
BEGIN
-- instantiate pASIC buffers/drivers on I/O signals
-- clocks
p1: CKPAD PORT MAP (txclk, tclk);       -- transmit path clock
p2: HDI2PAD PORT MAP (rxclkA, rxclkB, rclk); -- receive path clock on
                                             -- on negative edge
-- high drive pads
p3: HDIPAD PORT MAP (resetn ,reset);    -- active HIGH system reset
-- data buses
-- HOTLink receiver data bus (input)
p4:  INPAD PORT MAP (rxq(0), HL_rx(0));
p5:  INPAD PORT MAP (rxq(1), HL_rx(1));
p6:  INPAD PORT MAP (rxq(2), HL_rx(2));
p7:  INPAD PORT MAP (rxq(3), HL_rx(3));
p8:  INPAD PORT MAP (rxq(4), HL_rx(4));
p9:  INPAD PORT MAP (rxq(5), HL_rx(5));
p10: INPAD PORT MAP (rxq(6), HL_rx(6));
p11: INPAD PORT MAP (rxq(7), HL_rx(7));
p12: INPAD PORT MAP (rsc_d,  HL_rsc_d);       -- receive SC/D
p13: INPAD PORT MAP (r_rvs,  HL_r_rvs);       -- RVS
-- HOTLink transmitter data bus (output)
p14: OUTPAD PORT MAP (HL_tx(0), txd(0));
p15: OUTPAD PORT MAP (HL_tx(1), txd(1));
p16: OUTPAD PORT MAP (HL_tx(2), txd(2));
p17: OUTPAD PORT MAP (HL_tx(3), txd(3));
p18: OUTPAD PORT MAP (HL_tx(4), txd(4));
p19: OUTPAD PORT MAP (HL_tx(5), txd(5));
```

Appendix A. Top-Level pASIC Code (continued)

```
p20: OUTPAD PORT MAP (HL_tx(6), txd(6));
p21: OUTPAD PORT MAP (HL_tx(7), txd(7));
p22: OUTPAD PORT MAP (HL_tsc_q, tsc_d);
-- controller transmit data bus (input)
p24: INPAD PORT MAP (ctxd(0), c_txd(0));
p25: INPAD PORT MAP (ctxd(1), c_txd(1));
p26: INPAD PORT MAP (ctxd(2), c_txd(2));
p27: INPAD PORT MAP (ctxd(3), c_txd(3));
p28: INPAD PORT MAP (ctxd(4), c_txd(4));
p29: INPAD PORT MAP (ctxd(5), c_txd(5));
p30: INPAD PORT MAP (ctxd(6), c_txd(6));
p31: INPAD PORT MAP (ctxd(7), c_txd(7));
-- controller receiver data bus (output)
p34: OUTPAD PORT MAP (c_rxd(0), crxd(0));
p35: OUTPAD PORT MAP (c_rxd(1), crxd(1));
p36: OUTPAD PORT MAP (c_rxd(2), crxd(2));
p37: OUTPAD PORT MAP (c_rxd(3), crxd(3));
p38: OUTPAD PORT MAP (c_rxd(4), crxd(4));
p39: OUTPAD PORT MAP (c_rxd(5), crxd(5));
p40: OUTPAD PORT MAP (c_rxd(6), crxd(6));
p41: OUTPAD PORT MAP (c_rxd(7), crxd(7));
-- misc input pads
p44: INPAD PORT MAP (loopen, b_loopen); -- loopback enable
p45: INPAD PORT MAP (ctxc0, c_txc0);    -- transmit control 0
p49: INPAD PORT MAP (ctxp, c_txp);      -- odd parity input
-- misc output pads
p50: OUTPAD PORT MAP (c_rxs0, crxs0);   -- receiver status 0 output
p51: OUTPAD PORT MAP (c_rxs1, crxs1);   -- receiver status 1 output
p53: OUTPAD PORT MAP (b_sync, bsync);   -- byte sync acquired
p54: OUTPAD PORT MAP (r_error, error);  -- received bad character
p55: OUTPAD PORT MAP (p_err, perr);     -- parity error
p56: OUTPAD PORT MAP (c_rxp, crxp);     -- odd parity output
p57: OUTPAD PORT MAP (INV(b_loopen),ab_sel); -- HOTLink receiver A/B select
-------------------------------------------------------------------------
------------- TRANSMIT PATH ---------------------------------------------
-------------------------------------------------------------------------
-- add in transmit path input data pipeline register
t1a: DFF PORT MAP (c_txd(0), tclk, t_data(0));
t1b: DFF PORT MAP (c_txd(1), tclk, t_data(1));
t1c: DFF PORT MAP (c_txd(2), tclk, t_data(2));
t1d: DFF PORT MAP (c_txd(3), tclk, t_data(3));
t1e: DFF PORT MAP (c_txd(4), tclk, t_data(4));
t1f: DFF PORT MAP (c_txd(5), tclk, t_data(5));
t1g: DFF PORT MAP (c_txd(6), tclk, t_data(6));
t1h: DFF PORT MAP (c_txd(7), tclk, t_data(7));
```

Appendix A. Top-Level pASIC Code (continued)

```
-- add parity and control bits
t1j: DFF PORT MAP (c_txp, tclk, tp_odd);
t1k: DFF PORT MAP (c_txc0, tclk, c_txc_0);
-------------------------------------------------------------------------
-- add transmit data parity checker (10 bit parity tree)
t_parity <= NOT(t_data(0) XOR t_data(1) XOR t_data(2) XOR t_data(3)
  XOR t_data(4) XOR t_data(5) XOR t_data(6) XOR t_data(7)
  XOR tp_odd XOR c_txc_0);
-------------------------------------------------------------------------
-- add parity check F-F
t2: DFF PORT MAP (
  t_parity,                          -- parity of inputs
  tclk,                              -- transmit clock
  p_err);                            -- output parity status
-------------------------------------------------------------------------
-- add transmitter CRC generator
t3: crc_tx PORT MAP (
  tclk,                          -- transmit clock
  t_CRC_reset,                   -- from tx CRC control state machine
  c_txc_0,                       -- from tx input register
  mux_hi,                        -- enable low byte onto bus
  t_data,                        -- transmit data bus
  t_CRC);                        -- 8-bit transmit CRC output vector
t_CRC_reset <= '0' WHEN (c_txc_0 = '0' OR mux_hi = '0') ELSE '1';
-------------------------------------------------------------------------
-- add transmit output register
t5a: DFF PORT MAP (t_mux(0), tclk, HL_tx(0));
t5b: DFF PORT MAP (t_mux(1), tclk, HL_tx(1));
t5c: DFF PORT MAP (t_mux(2), tclk, HL_tx(2));
t5d: DFF PORT MAP (t_mux(3), tclk, HL_tx(3));
t5e: DFF PORT MAP (t_mux(4), tclk, HL_tx(4));
t5f: DFF PORT MAP (t_mux(5), tclk, HL_tx(5));
t5g: DFF PORT MAP (t_mux(6), tclk, HL_tx(6));
t5h: DFF PORT MAP (t_mux(7), tclk, HL_tx(7));
HL_tsc_d <= (mux_low AND c_txc_0) OR
            (c_txc_0 AND mux_hi AND ctxc3);
-- add in SC/D output bit
t5j: DFF PORT MAP (HL_tsc_d, tclk, HL_tsc_q);

-------------------------------------------------------------------------
-- add in transmit CRC supervisor machine
-- contains the double pipelined C/D bit
t6: tx_ctl_crc PORT MAP (
  tclk,                    -- transmit clock
  c_txc_0,                 -- registerd command/data control bit
  mux_hi,                  -- registered c_txc_0
  mux_low);                -- 2x registered c_txc_0
```

Appendix A. Top-Level pASIC Code (continued)

```
-------------------------------------------------------------------------
-- transmit path data/command/CRC mux
t8: PROCESS (c_txc_0, mux_low, mux_hi)
BEGIN
  IF (c_txc_0 = '0') THEN
    t_mux <= t_data;
  ELSIF (c_txc_0 = '1' AND ((mux_low = '0' AND mux_hi='0') OR
        (ctxc3 = '0' AND mux_low = '0' AND mux_hi = '1'))) THEN
    -- output CRC bytes
    t_mux <= t_CRC;
  ELSE
    -- output re-encoded command codes
    t_mux <= t_comm;
  END IF;
END PROCESS t8;

-- Add in transmit command decoder
t9: t_decode PORT MAP (t_data, t_comm); -- translate to HOTLink commands

-------------------------------------------------------------------------
-------------- RECEIVE PATH ---------------------------------------------
-------------------------------------------------------------------------
-- add in receive path input data pipeline register
r1a: DFF PORT MAP (HL_rx(0), rclk, r_data(0));
r1b: DFF PORT MAP (HL_rx(1), rclk, r_data(1));
r1c: DFF PORT MAP (HL_rx(2), rclk, r_data(2));
r1d: DFF PORT MAP (HL_rx(3), rclk, r_data(3));
r1e: DFF PORT MAP (HL_rx(4), rclk, r_data(4));
r1f: DFF PORT MAP (HL_rx(5), rclk, r_data(5));
r1g: DFF PORT MAP (HL_rx(6), rclk, r_data(6));
r1h: DFF PORT MAP (HL_rx(7), rclk, r_data(7)); -- add SC/D bit and RVS
r1j: DFF PORT MAP (HL_rsc_d, rclk, rcom_data); -- registerd SC/D
r1k: DFF PORT MAP (HL_r_rvs, rclk, rvs);        -- registered RVS signal
-- create double buffered signals
db1: BUF PORT MAP (rcom_data, r_com_data);
db2: BUF PORT MAP (rcom_data, r_com_data);
-------------------------------------------------------------------------
-- receive path output register
r2a: DFF PORT MAP (r_mux(0), rclk, c_rxd(0));
r2b: DFF PORT MAP (r_mux(1), rclk, c_rxd(1));
r2c: DFF PORT MAP (r_mux(2), rclk, c_rxd(2));
r2d: DFF PORT MAP (r_mux(3), rclk, c_rxd(3));
r2e: DFF PORT MAP (r_mux(4), rclk, c_rxd(4));
r2f: DFF PORT MAP (r_mux(5), rclk, c_rxd(5));
r2g: DFF PORT MAP (r_mux(6), rclk, c_rxd(6));
r2h: DFF PORT MAP (r_mux(7), rclk, c_rxd(7));-- command/data bit and rvs
r2j: DFF PORT MAP (r_com_data, rclk, c_rxs0);
r2k: DFF PORT MAP (rvs, rclk, r_error);
```

Appendix A. Top-Level pASIC Code (continued)

```
-------------------------------------------------------------------------
-- add receive parity generate
r3: TTL180 PORT MAP (
  r_mux(0), r_mux(1), r_mux(2), r_mux(3), r_mux(4), r_mux(5),
  r_mux(6), r_mux(7), INV(r_com_data), r_com_data, rp_odd, open);

r3a: DFF PORT MAP (rp_odd, rclk, c_rxp);
-------------------------------------------------------------------------
-- add in receive CRC block
r4: crc_rx PORT MAP (
  rclk,                    -- receive path clock
  r_com_data,              -- enable only for data bytes
  r_data,                  -- receiver data bus
  r_crc_err);              -- receive path crc status
-------------------------------------------------------------------------
-- add CRC check register
r5: DFF PORT MAP (r_CRC_d, rclk, c_rxs1);
r_CRC_d <= r_crc_err AND r_com_data AND (NOT(c_rxs0));
-------------------------------------------------------------------------
-- add in byte-sync state machine
r6: byte_syn PORT MAP (
  rclk,                    -- receiver clock
  reset,                   -- system reset
  rvs,                     -- receiver RVS signal
  sync,                    -- decoded k28.5
  b_sync);                 -- byte sync acquired

sync <= '1' WHEN (r_com_data='1' AND r_data(0 TO 3)="1010") ELSE '0';
-------------------------------------------------------------------------
-- add command transposition logic and mux
r7: PROCESS (r_com_data, r_data(0), r_data(1), r_data(2), r_data(3))
BEGIN
  IF (r_com_data='0') THEN
    r_mux <= r_data;
  ELSE
    r_mux <= t_code;         -- add in command decoder
  END IF;
END PROCESS r7;
-------------------------------------------------------------------------
-- add receiver path command encoder
-- t_code is output vector
r8: t_encode PORT MAP (
  r_data,                  -- HOTLink data bus
  t_code);                 -- decoded Triquint commands

END escon_top;
```

Appendix B. Transmit Path CRC Generator

```
-- CRC_T.VHD
--
-- transmit 16-bit CCITT CRC for use in data mover
--
-- When sequencing bytes out, the qt(15)-qt(8) byte must be sent out first.
-- Per the ESCON spec, the CRC is the 1's compliment (inversion) of the
-- qt[15:0] bus.
--
PACKAGE crc_T IS
  COMPONENT crc_tx PORT (
  clk,                        -- system clock
  preset: IN      BIT;        -- synchronous preset, set to all 1s
  enable: IN      BIT;        -- enable when not a command byte
  mux_hi: IN      BIT;        -- enable high-byte onto bus
  dt:     IN      BIT_VECTOR (0 TO 7); -- Input data byte
  q_out:  OUT     BIT_VECTOR (0 TO 7)  -- CRC register
  );
  END COMPONENT;
END crc_T;

use work.rtlpkg.all;
use work.cypress.all;

ENTITY crc_tx IS PORT (
  clk,                        -- system clock
  preset: IN      BIT;        -- synchronous reset, set to all 1s
  enable: IN      BIT;        -- enable when not a command byte
  mux_hi: IN      BIT;        -- enable high CRC byte out
  dt:     IN      BIT_VECTOR (0 TO 7); -- Input data byte
  q_out:  OUT     BIT_VECTOR (0 TO 7)  -- CRC register
  );
END crc_tx;

ARCHITECTURE ccitt_tx OF crc_tx IS
SIGNAL qt: BIT_VECTOR (0 TO 15); -- CRC register
BEGIN
proc1: PROCESS BEGIN
  WAIT UNTIL (clk='1');
    IF (preset='1') THEN
      qt <= x"FFFF";      -- Preset to 1's for reset
    ELSIF (enable='1') THEN
      qt <= qt;           -- keep same value
    ELSE
      qt(0) <= qt(8) XOR qt(12) XOR dt(3) XOR dt(7);
      qt(1) <= qt(9) XOR qt(13) XOR dt(2) XOR dt(6);
      qt(2) <= qt(10) XOR qt(14) XOR dt(1) XOR dt(5);
      qt(3) <= qt(11) XOR qt(15) XOR dt(0) XOR dt(4);
```

Appendix B. Transmit Path CRC Generator (continued)

```
      qt(4) <= qt(12) XOR dt(3);
      qt(5) <= qt(13) XOR qt(12) XOR qt(8) XOR dt(7) XOR dt(3) XOR dt(2);
      qt(6) <= qt(14) XOR qt(13) XOR qt(9) XOR dt(1)XOR dt(2) XOR dt(6);
      qt(7) <= qt(15) XOR qt(14) XOR qt(10) XOR dt(0) XOR dt(1) XOR dt(5);
      qt(8) <= qt(15) XOR qt(11) XOR qt(0) XOR dt(0) XOR dt(4);
      qt(9) <= qt(12) XOR qt(1) XOR dt(3);
      qt(10) <= qt(13) XOR qt(2) XOR dt(2);
      qt(11) <= qt(14) XOR qt(3) XOR dt(1);
      qt(12) <= qt(15) XOR qt(12) XOR qt(8) XOR qt(4)
                XOR dt(0) XOR dt(3) XOR dt(7);
      qt(13) <= qt(13) XOR qt(9) XOR qt(5) XOR dt(2) XOR dt(6);
      qt(14) <= qt(14) XOR qt(10) XOR qt(6) XOR dt(1) XOR dt(5);
      qt(15) <= qt(15) XOR qt(11) XOR qt(7) XOR dt(0) XOR dt(4);
    END IF;
END PROCESS;

-- mux and Invert CRC and swap bits
m1: PROCESS (mux_hi)
BEGIN
-- Mux out high and low bytes and transpose bit order
IF mux_hi = '0' THEN
  q_out(7) <= not qt(8);
  q_out(6) <= not qt(9);
  q_out(5) <= not qt(10);
  q_out(4) <= not qt(11);
  q_out(3) <= not qt(12);
  q_out(2) <= not qt(13);
  q_out(1) <= not qt(14);
  q_out(0) <= not qt(15);
ELSE
  q_out(7) <= not qt(0);
  q_out(6) <= not qt(1);
  q_out(5) <= not qt(2);
  q_out(4) <= not qt(3);
  q_out(3) <= not qt(4);
  q_out(2) <= not qt(5);
  q_out(1) <= not qt(6);
  q_out(0) <= not qt(7);
END IF;
END PROCESS m1;

END ccitt_tx;
```

Appendix C. Transmit Path CRC Controler

```
-- CTL_CRCT.VHD
--
-- Control transmit CRC function
--
--
-- All actions are based on the CTXC0 input.  This input is active
-- at the end of every data sequence and is a 1 (HIGH) for all
-- non-data bytes.
--

PACKAGE crc_ctl IS
  COMPONENT tx_ctl_crc PORT (
  clk,                      -- transmit clock
  ctxc0: IN BIT;            -- command/data control bit
  ctxc1,                    -- registered ctxc0
  ctxc2,                    -- 2x registered ctxc0
  ctxc3: OUT BIT);          -- 3x registered ctxc0
  END COMPONENT;
END crc_ctl;

ENTITY tx_ctl_crc IS PORT (
  clk,                      -- transmit clock
  ctxc0: IN BIT;            -- command/data control bit
  ctxc1,                    -- registered ctxc0
  ctxc2,                    -- 2x registered ctxc0
  ctxc3: OUT BIT);          -- 3x registered ctxc0
END tx_ctl_crc;

USE work.cypress.all;
USE work.rtlpkg.all;

ARCHITECTURE ctl_1 OF tx_ctl_crc IS

SIGNAL cq1: BIT;                -- single registered c/d
SIGNAL cq2: BIT;                -- double registered c/d

BEGIN

-- Instantiate DFF to track status of ctxc0 bit
d1: DFF PORT MAP (ctxc0, clk, cq1);
d2: DFF PORT MAP (cq1, clk, cq2);
d3: DFF PORT MAP (cq2, clk, ctxc3);

-- assign outputs
ctxc1 <= cq1;
ctxc2 <= cq2;

END ctl_1;
```

Appendix D. Command Mapper

```
-- TRI_CODE.VHD
--
-- Command decode/translate between the Triquint GA9104 and HOTLink
-- K-code command sets

--------------------------------------------------------------------------------
-- Triquint/Cypress Command mapping
--          GA9104          HOTLink HEX
--          HEX BIN         TX RX        BIN
-- k28.0*   1C 00011100     00           00000000
-- k28.1    3C 00111100     01           00000001
-- k28.2    5C 01011100     02           00000010
-- k28.3*   7C 01111100     03           00000011
-- k28.4    9C 10011100     04           00000100
-- k28.5    BC 10111100     05,E1,E2     00000101
-- k28.6    DC 11011100     06           00000110
-- k28.7    FC 11111100     07,27,47     00000111
-- k23.7*   F7 11110111     08           00001000
-- k27.7*   FB 11111011     09           00001001
-- k29.7*   FD 11111101     0A           00001010
-- k30.7*   FE 11111110     0B           00001011
-- * - Illegal for use in ESCON operations

PACKAGE triq_code IS
  COMPONENT t_encode PORT (
  c_code : IN BIT_VECTOR(0 TO 7); -- Cypress HOTLink C-codes
  t_code : OUT BIT_VECTOR(0 TO 7) -- Triquint K-codes
  );
  END COMPONENT;

  COMPONENT t_decode PORT (
  t_data : IN BIT_VECTOR(0 TO 7); -- Triquint K-codes
  t_comm : OUT BIT_VECTOR(0 TO 7) -- Cypress HOTLink C-codes
  );
  END COMPONENT;
END triq_code;

USE work.cypress.all;
USE work.table_bv.all;          -- use for command encoder

ENTITY t_encode IS PORT (
  c_code : IN BIT_VECTOR(0 TO 7); -- Cypress HOTLink C-codes
  t_code : OUT BIT_VECTOR(0 TO 7) -- Triquint K-codes
  );
END t_encode;
```

Appendix D. Command Mapper (continued)

```
ARCHITECTURE t_encoder OF t_encode IS
-- use TTF function to translate from one command set to the other
-- Command constants
-- T-codes (output vectors)
CONSTANT K28_0: x01_VECTOR(0 TO 7) := "00111000";
CONSTANT K28_1: x01_VECTOR(0 TO 7) := "00111100";
CONSTANT K28_2: x01_VECTOR(0 TO 7) := "00111010";
CONSTANT K28_3: x01_VECTOR(0 TO 7) := "00111110";
CONSTANT K28_4: x01_VECTOR(0 TO 7) := "00111001";
CONSTANT K28_5: x01_VECTOR(0 TO 7) := "00111101";
CONSTANT K28_6: x01_VECTOR(0 TO 7) := "00111011";
CONSTANT K28_7: x01_VECTOR(0 TO 7) := "00111111";
CONSTANT K23_7: x01_VECTOR(0 TO 7) := "11101111";
CONSTANT K27_7: x01_VECTOR(0 TO 7) := "11011111";
CONSTANT K29_7: x01_VECTOR(0 TO 7) := "10111111";
CONSTANT K30_7: x01_VECTOR(0 TO 7) := "01111111";
-- C-codes (input vectors)
CONSTANT C00_0: x01_VECTOR(0 TO 7) := "0000xxxx";
CONSTANT C01_0: x01_VECTOR(0 TO 7) := "1000xxx0";
CONSTANT C02_0: x01_VECTOR(0 TO 7) := "0100xxx0";
CONSTANT C03_0: x01_VECTOR(0 TO 7) := "1100xxxx";
CONSTANT C04_0: x01_VECTOR(0 TO 7) := "0010xxxx";
CONSTANT C05_0: x01_VECTOR(0 TO 7) := "1010xxxx";
CONSTANT C06_0: x01_VECTOR(0 TO 7) := "0110xxxx";
CONSTANT C07_0: x01_VECTOR(0 TO 7) := "1110xxxx";
CONSTANT C08_0: x01_VECTOR(0 TO 7) := "0001xxxx";
CONSTANT C09_0: x01_VECTOR(0 TO 7) := "1001xxxx";
CONSTANT C10_0: x01_VECTOR(0 TO 7) := "0101xxxx";
CONSTANT C11_0: x01_VECTOR(0 TO 7) := "1101xxxx";
CONSTANT C12_0: x01_VECTOR(0 TO 7) := "0011xxxx";
-- errors and special mappings
CONSTANT C01_7: x01_VECTOR(0 TO 7) := "1000xxx1";
CONSTANT C02_7: x01_VECTOR(0 TO 7) := "0100xxx1";

CONSTANT table: x01_TABLE(0 TO 13, 0 TO 15) := (   -- command mappings
--          Command
--       Input   Output
--       -----   ------
         C00_0 & K28_0,
         C01_0 & K28_1,
         C02_0 & K28_2,
         C03_0 & K28_3,
         C04_0 & K28_4,
         C05_0 & K28_5,
         C06_0 & K28_6,
         C07_0 & K28_7,
```

Appendix D. Command Mapper (continued)

```
        C08_0 & K23_7,
        C09_0 & K27_7,
        C10_0 & K29_7,
        C11_0 & K30_7,
        C01_7 & K28_5,
        C02_7 & K28_5);

BEGIN
p1: PROCESS (c_code)
  BEGIN
    t_code <= ttf(table,(c_code));
  END PROCESS p1;
END t_encoder;

USE work.cypress.all;

ENTITY t_decode IS PORT (
  t_data : IN BIT_VECTOR(0 TO 7); -- Triquint K-codes
  t_comm : OUT BIT_VECTOR(0 TO 7) -- Cypress HOTLink C-codes
  );
END t_decode;

ARCHITECTURE t_decoder OF t_decode IS

BEGIN

t_comm(7) <= '0';
t_comm(6) <= '0';
t_comm(5) <= '0';
t_comm(4) <= '0';
t_comm(3) <= '0' WHEN (t_data(0 TO 1) = "00") ELSE '1';
t_comm(2) <= '1' WHEN ((t_data(7) = '1')
             AND (t_data(0 TO 1) = "00")) ELSE '0';

t1: PROCESS (t_data(0), t_data(1), t_data(6),
           t_data(5), t_data(3), t_data(2))
  BEGIN
    IF (t_data(0 TO 1) = "00") THEN
      t_comm(1) <= t_data(6);
      t_comm(0) <= t_data(5);
    ELSE
      t_comm(1) <= t_data(3) AND t_data(2);
      t_comm(0) <= t_data(2) AND t_data(0);
    END IF;
  END PROCESS t1;

END t_decoder;
```

Appendix E. Receive Path CRC Checker

```
-- CRC_R.VHD
--
-- receiver 16-bit CCITT CRC for use in data mover
PACKAGE crc_r IS
  COMPONENT crc_rx PORT (
  clk,                      -- system clock
  preset: IN      BIT;      -- synchronous reset, set to all 1s
  dr:      IN      BIT_VECTOR (0 TO 7); -- Input data byte
  crc_err:  OUT   BIT       -- error detected
  );
  END COMPONENT;
END crc_r;

use work.rtlpkg.all;
use work.cypress.all;

ENTITY crc_rx IS PORT (
  clk,                      -- system clock
  preset:  IN BIT;          -- synchronous preset, set to all 1s
  dr:      IN      BIT_VECTOR (0 TO 7); -- Input data byte
  crc_err:  OUT   BIT       -- error detected
  );
END crc_rx;

ARCHITECTURE ccitt_rx OF crc_rx IS
-- declare CRC register
SIGNAL qr: BIT_VECTOR (0 TO 15);    -- CRC register
ATTRIBUTE POLARITY OF qr:SIGNAL IS PL_KEEP;  -- maintain polarity f
BEGIN
proc1: PROCESS BEGIN
  WAIT UNTIL (clk='1');
    IF (preset='1') THEN
      qr <= x"FFFF";      -- Preset to 1's for reset
    ELSE
      qr(0) <= qr(8) XOR qr(12) XOR dr(3) XOR dr(7);
      qr(1) <= qr(9) XOR qr(13) XOR dr(2) XOR dr(6);
      qr(2) <= qr(10) XOR qr(14) XOR dr(1) XOR dr(5);
      qr(3) <= qr(11) XOR qr(15) XOR dr(0) XOR dr(4);
      qr(4) <= qr(12) XOR dr(3);
      qr(5) <= qr(13) XOR qr(12) XOR qr(8) XOR dr(7) XOR dr(3) XOR dr(2);
      qr(6) <= qr(14) XOR qr(13) XOR qr(9) XOR dr(1) XOR dr(2) XOR dr(6);
      qr(7) <= qr(15) XOR qr(14) XOR qr(10) XOR dr(0) XOR dr(1) XOR dr(5);
      qr(8) <= qr(15) XOR qr(11) XOR qr(0) XOR dr(0) XOR dr(4);
      qr(9) <= qr(12) XOR qr(1) XOR dr(3);
      qr(10) <= qr(13) XOR qr(2) XOR dr(2);
      qr(11) <= qr(14) XOR qr(3) XOR dr(1);
```

Appendix E. Receive Path CRC Checker (continued)

```
      qr(12) <= qr(15) XOR qr(12) XOR qr(8) XOR qr(4)
                XOR dr(0) XOR dr(3) XOR dr(7);
      qr(13) <= qr(13) XOR qr(9) XOR qr(5) XOR dr(2) XOR dr(6);
      qr(14) <= qr(14) XOR qr(10) XOR qr(6) XOR dr(1) XOR dr(5);
      qr(15) <= qr(15) XOR qr(11) XOR qr(7) XOR dr(0) XOR dr(4);
    END IF;
END PROCESS;

-- Need to look for a 1D0F at the receiver
-- output is LOW when 1D0F present
crc_err <= NOT(qr(0)) OR NOT(qr(1)) OR NOT(qr(2)) OR NOT(qr(3))
      OR qr(4) OR qr(5) OR qr(6) OR qr(7)
      OR NOT(qr(8)) OR qr(9) OR NOT(qr(10)) OR NOT(qr(11))
      OR NOT(qr(12)) OR qr(13) OR qr(14) OR qr(15);

END ccitt_rx;
```

Appendix F. Byte Sync Controller

```
-- B_SYNC.VHD - byte synchronization state machine
--
-- This machine has a five state supervisor machine that tracks
-- the number of errors detected within a specific period of time.
-- It also tracks valid characters and SYNC codes.

PACKAGE sync_det IS
  COMPONENT byte_syn PORT (
  clk,                    -- Receiver clock
  reset,                  -- system reset
  error,                  -- bad character
  sync: IN BIT;           -- valid k28.5
  bsync: OUT BIT);        -- byte-sync acquired
  END COMPONENT;
END sync_det;

USE work.cypress.all;
USE work.rtlpkg.all;
USE work.counterpkg.all;

ENTITY byte_syn IS PORT (
  clk,                    -- Receiver clock
  reset,                  -- system reset
  error,                  -- bad character
  sync: IN BIT;           -- valid k28.5
  bsync: OUT BIT);        -- byte-sync acquired
END byte_syn;

ARCHITECTURE arch1 OF byte_syn IS
-- declare internal signals
SIGNAL ctr_en: BIT;             -- counter enable
SIGNAL ctr_reset: BIT;          -- counter reset
SIGNAL bbsync: BIT;             -- interface in sync
SIGNAL cnt: BIT_VECTOR(0 TO 3); -- 4-bit counter vector
-- declare state machine
TYPE sync_state IS (
        state0,                 -- reset or errors, waiting for SYNC codes
        state1,                 -- no errors, in sync
        state2,                 -- 1 error, in sync
        state3,                 -- 2 errors, in sync
        state4);                -- 3 errors, in sync

-- declare state machine encoding, state variable, and initial state
SIGNAL s_state : sync_state := state0;
```

Appendix F. Byte Sync Controller (continued)

```
BEGIN
proc1: PROCESS BEGIN
  WAIT UNTIL (clk='1');
  IF (reset='1') THEN
    s_state <= state0;       -- don't even look yet
  ELSE
    CASE s_state IS
      WHEN state0 =>
        IF ((cnt="1111") AND (error='0')) THEN
          s_state <= state1;
        ELSE
          s_state <= state0;
        END IF;
      WHEN state1 =>
        IF (error='1') THEN
          s_state <= state2;
        ELSE
          s_state <= state1;
        END IF;
      WHEN state2 =>
        IF (error='1') THEN
          s_state <= state3;
        ELSIF (cnt="1111") THEN
          s_state <= state1;
        ELSE
          s_state <= state2;
        END IF;
      WHEN state3 =>
        IF (error='1') THEN
          s_state <= state4;
        ELSIF (cnt="1111") THEN
          s_state <= state2;
        ELSE
          s_state <= state3;
        END IF;
      WHEN state4 =>
        IF (error='1') THEN
          s_state <= state0;
        ELSIF  (cnt="1111") THEN
          s_state <= state3;
        ELSE
          s_state <= state4;
        END IF;
      WHEN others =>
        s_state <= state0;
    END CASE;
  END IF;
END PROCESS proc1;
```

Appendix F. Byte Sync Controller (continued)

```
-- build 4-bit counter with enable and reset
ctr_en <=  '1' WHEN ((s_state=state0 AND reset='0' AND sync='1')
        OR (s_state=state2)
        OR (s_state=state3)
        OR (s_state=state4))
        ELSE '0';

ctr_reset <= '1' WHEN ((reset='1') OR (error='1')) ELSE '0';

-- add standard counter module
ctr1: cntr4 PORT MAP (                    -- contains the 4 bits of ctr1
  one,                              -- set carry in always active
  open,                             -- carry out unused
  ctr_en,                           -- counter enable
  zero,                             -- never load this counter
  zero, zero, zero, zero,           -- load inputs are not used
  clk,                              -- counter clock
  ctr_reset,                        -- will need to expand this signal
  cnt(3), cnt(2),                   --  counter holding register inputs
  cnt(1), cnt(0)
  );

-- assign output
bbsync <= '0' WHEN (s_state=state0) ELSE '1';
d1: DFF PORT MAP (bbsync, clk, bsync);

END arch1;
```

Appendix G. I/O Support

```
-- IOPLUS.VHD

-- Create enhanced I/O buffer that is not part of the io.vhd
-- package for the pASIC 380 family

PACKAGE iopluspkg IS
  COMPONENT HDI2PAD PORT (
    p0 : IN BIT;
    p1 : IN BIT;
    qn : OUT BIT);
  END COMPONENT;
END iopluspkg;

USE work.cypress.all;
USE work.rtlpkg.all;
USE work.iopkg.all;
USE work.resolutionpkg.all;

ENTITY HDI2PAD IS PORT (
  p0 : IN BIT;
  p1 : IN BIT;
  qn : OUT BIT);
END HDI2PAD;

ARCHITECTURE archHDI2PAD OF HDI2PAD IS

SIGNAL o : multi_buffer BIT;

BEGIN

u0: PAINCELL PORT MAP ( ip => p0, ini => o, iz => OPEN);
u1: PAINCELL PORT MAP ( ip => p1, ini => o, iz => OPEN);
qn <= o;

END archHDI2PAD;
```

Using the CY7B923 as an ECL Clock Source

Abstract

This application note details the use of an inexpensive data communications transmitter device as a high-precision, flexible, and programmable Emitter-Coupled-Logic (ECL) or Positive-Emitter-Coupled Logic (PECL) clock source. Issues concerning clock characteristics, stability, distribution and design techniques are discussed in detail. Information is provided to allow the user to configure the device for a variety of applications.

The Ideal Clock Circuit

The ideal clock source would provide the designer with several attributes that would benefit the eventual design. It would be flexible in that it would provide for a broad range of frequency coverage. Its frequency would be stable from one cycle to the next, its pulsewidth would be stable over time and both of these parameters would be consistent over temperature and voltage variations. The clock output transition time from one level to another (the rise and fall time) would be short in order to minimize the skew caused by sampling threshold effects at the receive end of the clock. It would be capable of sourcing significant amounts of current into multiple single-ended or differential PECL/ECL loads with a minimum amount of output skew. It would provide for relatively low power consumption when compared to PECL/ECL clock sources currently available. And lastly, it would be a low-cost device, available in a variety of packages for compatibility with commercial, industrial, military, and surface-mount applications. The device makes use of an inexpensive TTL clock oscillator instead of the expensive PECL/ECL devices typically used. The Cypress CY7B923 HOTLink™ Transmitter, although not specifically designed as an ECL clock source, provides the features to address these needs in a highly effective manner.

HOTLink Transmitter Features and Specifications

The HOTLink chip set is comprised of a pair of high-speed point-to-point communications building blocks that operate over high-speed serial data links (fiber-optic, coaxial cable, and twisted/parallel pair) at 160 to 330 Mbits/second. The HOTLink pair consists of the CY7B923 Transmitter and the CY7B933 Receiver. The transmitter features a set of three positive 100K (referenced to +5) ECL differential output buffers, a data input register, an encoder to encode 8-bit data into a 10-bit word, a built-in self-test (BIST) pattern generator, a serializer to convert parallel data to serial data, and a clock generator to produce a bit-rate clock from the incoming word-rate clock input. These features of the CY7B923, with the exception of the encoder and built-in delf-yest circuits, make it ideal for use as a clock generator device.

CY7B923 Block Diagram Description

The HOTLink Transmitter is designed to transform information from a word-rate or byte-rate parallel format into a high-speed serial format. A block diagram of the CY7B923 HOTLink Transmitter is shown in *Figure 1* and a description of each module follows.

Clock Generator

The clock generator contains a phase-locked loop (PLL) that multiplies a word-rate reference clock (CKW) by a factor of ten to produce the serial bit-

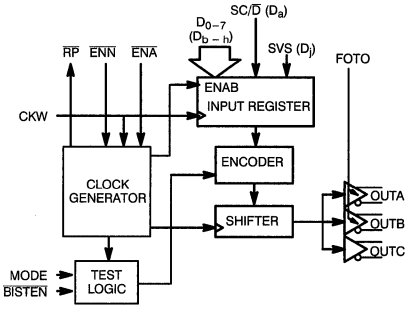


Figure 1. CY7B923 HOTLink Transmitter Logic Block Diagram

rate. Data is clocked into the input register on the rising edge of CKW. The duty cycle of CKW does not affect the outgoing serial-bit stream since the PLL is capable of maintaining proper phase and duty cycle on its own. The clock generator also produces a signal called $\overline{RP}$ (Read Pulse) and is used to read new data from a FIFO in a data communications application. $\overline{RP}$ is not used when the HOTLink Transmitter is used as a clock source.

Input Register

The input register captures the data present at the Da through Dj inputs at the rising edge of the CKW clock. This parallel data is then loaded directly into the shifter for serialization if the encoder is disabled.

Encoder

The encoder is used to encoded the incoming data from the input register into an 8B/10B format for ANSI X3T9.3 (Fibre Channel) or IBM® ESCON™ applications only. In unencoded mode, the data passes directly from the input register into the shifter. This application of the HOTLink Transmitter as an ECL clock source uses the CY7B923 in unencoded mode.

Shifter

The shifter accepts the the 10-bit word which was loaded into the input register. With the encoder disabled, the data is converted from parallel to serial with the data present on input pin Da (pin 19) shifted out first.

Output

The device supports three PECL (100K referenced to +5V) outputs. These outputs provide differential (true and complement) capability, offer an enable pin (the FOTO pin for output pairs A and B) and the ability to drive 50Ω transmission lines directly.

Test Logic

The test logic is not used in the ECL clock source application. It contains the logic to generate the built-in self-test pattern that is used to test the integrity of a data communications interface and link.

Fulfilling the Requirements

Frequency Range

Since the HOTLink transmitter was designed to communicate or send data at a rate of 160 Mbps to 330 Mbps, it is ideally suited for the application of generating precise transitions or clock edges over a broad range of frequencies. As the transmitter operates, data in the form of 10-bit words are loaded into the serializer of the CY7B923 at the word-rate clock intervals. The on-board PLL takes the incoming word-rate clock and multiplies it by a factor of ten to generate the rate at which the individual bit transitions will be shifted out by the serializer. The encoder function is disabled when the transmitter is used as a clock source to provide maximum control of the data patterns being shifted out. The two primary factors that affect clock output frequency are word-rate clock frequency and the number of bit transitions within the 10-bit word. See *Equation 1* below:

$$\textit{clock out} = \textit{(word-rate clock)}\ \textit{(\# of transitions)} / 2 \quad \text{Eq. 1}$$

Where:

clock out = clock frequency present at the outputs of the transmitter

word-rate clock = rate at which the 10-bit words are loaded into the serializer

Table 1.

Data Pattern	Word Rate	Duty Cycle	Bit Transitions	Clock Frequency
0000011111	16 MHz	50%	2	16 MHz (Min. Rate)
0000001111	25 MHz	40%	2	25 MHz
0011100111	16 MHz	60%	4	32 MHz
0000011111	33 MHz	50%	2	33 MHz
0000100001	25 MHz	20%	4	50 MHz
0001100011	33 MHz	40%	4	66 MHz
0101010101	16 MHz	50%	10	80 MHz
0101010101	25 MHz	50%	10	125 MHz
0101010101	33 MHz	50%	10	165 MHz (Max. Rate)

NOTE: The minimum duty cycle is 10% and the maximum duty cycle is 90%. The minimum clock out is 16 MHz and the maximum clock out is 165 MHz.

# of transitions = the number of transitions between one logic level and another

Assume a 20-MHz word-rate clock and a data pattern of 0000011111:

Now, assume a data pattern of 0101010101:

clock out = (20 MHz) (10 transitions) / 2 => 100 MHz

The duty cycle, or relationship of clock HIGH time to clock LOW time, can also be affected by the data pattern loaded into the serializer. The duty cycle is controlled by the ratio of consecutive ones to consecutive zeros. If there are six consecutive ones and four consecutive zeros in the data pattern, the duty cycle would be 60%. If the pattern was 0001100011 the duty cycle would be three LOW and two HIGH or 40% but the number of bit transitions would double and so would the clock out frequency.

Table 1 shows examples of clock out frequencies and duty cycles that can be obtained using the data patterns and source frequencies given.

Test Circuit

A typical test circuit is shown in *Figure 2,* detailing the HOTLink Transmitter in a PECL clock generator application. The circuit uses the CY7B923 with a 10-position DIP switch to select the desired data pattern (e.g., *Table 1* patterns). The $\overline{\text{BISTEN}}$, and MODE pins are pulled to a logic HIGH while $\overline{\text{ENA}}$ or $\overline{\text{ENN}}$ are tied to a logic LOW. This configures the device to operate with built-in self-test disabled, 8B/10B Encoding disabled, and data present at the Da to Dj to be loaded into the input register on each rising edge of the CKW input. The FOTO input is used as a clock output enable for output pairs OUTA and OUTB. When FOTO is LOW, transmit data will continuously be driven on output pairs A and B. When FOTO is HIGH, the output pairs A and B will remain at a logic zero state. Output pair OUTC is always enabled and will reflect the current state of the transmitter shifter output. The $\overline{\text{RP}}$ or Read Pulse output is typically used to indicate new data can be read from a FIFO or other storage device into the transmitter. It is not used in the clock generator application. In the test circuit shown, the word-rate clock could be any stable TTL clock source operating between 16 MHz and 33 MHz. As described above, the resultant output clock frequency is dependent on word clock frequency (CKW) and the number of 0-to-1 or 1-to-0 bit transitions present in the 10-bit word loaded into the input register of the transmitter.

Clock Issues

Since the CY7B923 was originally intended for very high-speed communications, the inherent stability of the communications device must be extremely

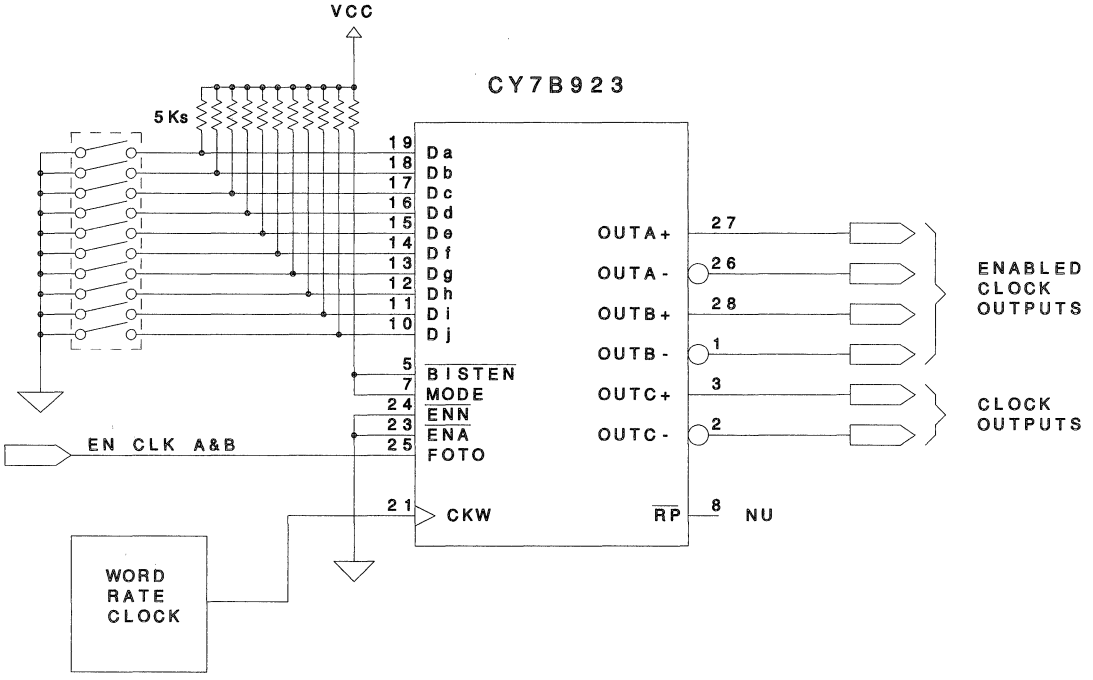


Figure 2. CY7B923 Clock Generator Test Circuit

good to prevent data communication errors and meet the rigid requirements of the standards imposed by industry. These same principles relating to clock stability also apply when the device is used as a PECL/ECL clock source. In general, the most critical factors relating to clock performance are jitter, duty cycle stability, rise and fall times, and output skew.

Jitter

Jitter is typically defined as the variation of one clock edge with respect to another. One source of jitter can be caused by noise-induced variations in the PLL, often known as random jitter. An additional form of jitter can result from the data patterns fed to the transmitter. This data dependent jitter is not relevant in the case of the clock generator because the pattern is constant and repeating. Jitter can also have an effect on the duty cycle of a clock waveform, generally referred to as duty cycle distortion. Refer to the CY7B923 datasheet for more specifications on jitter.

Duty Cycle Stability

For the clock generator application, the duty cycle, or relationship of a logic HIGH time period to a logic LOW time period, is dependent on three factors: random jitter, transmit data pattern, and rise and fall times. Random jitter has an affect on duty cycle based on the fact that it will vary the placement of one clock edge with respect to another. Another factor relating to duty cycle stems from the variation of the data pattern presented to the inputs of the transmitter. This is considered a very coarse adjustment as it can only be varied by a minimum of a single-serial bit-time. The last factor is rise and fall time, and is largely dependent on the circuit the outputs are driving.

Rise and Fall Time

Rise and fall times are defined as the period of time required for a signal to transition from a logic LOW to a logic HIGH or a logic HIGH to a logic LOW. The rise time of an ECL output is mainly determined by internal parameters such as the internal driver resistor and the parasitic capacitance of the output and is generally fixed. The fall time however, is generally based on the biasing of the output, the load capacitance, and the termination of the clock circuit. If each of the outputs are properly biased and treated as a transmission line, the driver is capable of matching rise and fall times. A proper biasing technique is to tie the PECL output to V_{CC} – 2.0V through a 50Ω resistor. Since ECL outputs switch at such high speeds, typically in the 1-ns range, most ECL circuit board traces greater than 1 inch in length should be treated as transmission lines and require termination (Reference 3). When a circuit board trace acts as a transmission line and is unterminated, it will exhibit a reflection of the energy pulse from the destination back to the source. If this reflection is significant, it can cause erroneous triggering of digital logic circuits. The CY7B923 datasheet indicates a maximum rise time and fall time of 1.2 ns measured at the 80% and 20% voltage points driving an ECL load of 5 pF and 50Ω terminated to V_{CC} – 2.0 Volts. This is specified as a guaranteed maximum. Typical rise and fall times are less than the maximum.

Termination

The two types of termination techniques generally used to control transmission line effects are series and parallel termination. A series termination is designed to match the source driver to the characteristic impedance of the line being driven. This termination approach is not recommended for the HOTLink Transmitter. A parallel termination, on the other hand, will match the characteristic impedance of the line being driven to the load. This termination procedure, also called a Thevenin Termination, consists of a pull-up resistor to the positive supply and a pull-down resistor to the negative supply. This termination can also double as a biasing network and serve both purposes: transmission line termination and ECL output biasing. An example of an improperly terminated ECL waveform is shown in *Figure 3*. Notice the excessive ringing on the logic LOW level. *Figure 4* shows what a properly terminated ECL signal should look like. Notice the symmetrical rise and fall times and the absence of any ringing on the waveform.

Refer to the Cypress Semiconductor *Applications Handbook* or the Cypress Semiconductor "HOTLink Design Considerations" Application Note for more information on transmission line termination techniques.

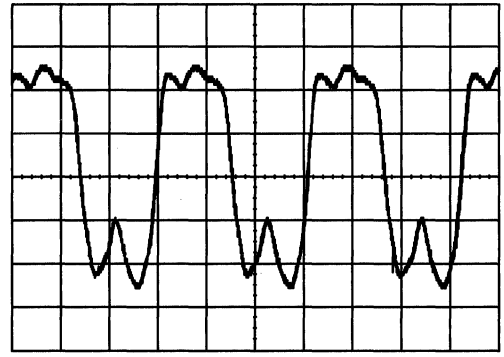


Figure 3. Improperly Terminated Waveform

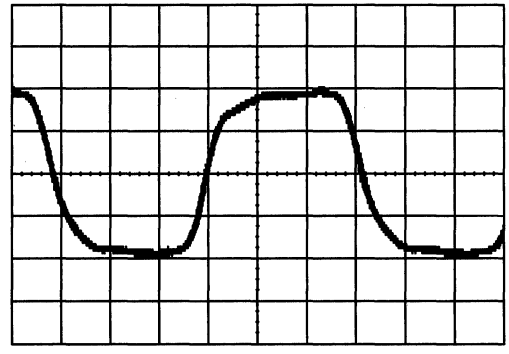


Figure 4. Properly Terminated ECL Waveform

Clock Skew

Clock skew is introduced into a digital system in two ways. The first is called output skew and is defined as the difference in time between clock edges being driven from the OUTA, OUTB, and OUTC transmitter output pairs. Output skew is caused internally by the clock driver circuit itself. It can result from the differences in output driver characteristics between output pairs or even in layout and placement differences of the physical driver structures on the die. The second source of clock skew is related to the printed circuit board layout and placement. Trace length, capacitive loading, termination components, printed circuit board characteristics, supply voltages, and many other factors affect these external delays. It is important that the designer understand the issues affecting clock skew because one must be able to accurately predict when clock edges will arrive at a load or destination for proper synchronization of a digital system.

Drive Capability

The HOTLink Transmitter features three sets of differential PECL/ECL outputs. Each of these outputs is capable of driving a 50Ω load with a maximum output current of 50 mA.

Power Supply Current

The HOTLink Transmitter has a maximum I_{CCT} specification of 85 mA for commercial and 95 mA for military temperature devices. Additionally, each enabled output pair contributes 35 mA to I_{CCT} when loaded to 50Ω Unused outputs may be left open, or better yet, tied to V_{CC} to minimize the power dissipated by the output circuit and reduce a source of unwanted noise. A 5-mA power savings can be obtained by disabling the output current source in this manner.

HOTLink Transmitter Printed Circuit Layout

Care must be taken when laying out a printed circuit board for the HOTLink Transmitter and when designing any clock circuit in general. Proper power-supply filtering and bypass techniques must be employed to ensure reliable operation and the correct components must be selected. Everything from the oscillator used to feed the CKW input to the type and placement of the bypass capacitors used is critical. Refer to the Cypress Semiconductor "HOTLink Design Considerations" Application Note for specific details on circuit layout and bypassing.

Device Packaging

Like virtually all Cypress devices, the CY7B923 HOTLink Transmitter is available in commercial (0 to 70 degrees C), industrial (−40 to +85 degrees C) and military (−55 to +125 degrees C) temperature ranges at V_{CC} ± 10%. The device comes packaged in a 28-pin PLCC, 28-pin LCC, or a 28-pin 300-mil-wide SOIC to suit a broad range of packaging requirements. The device is not available in Dual-In-Line (DIP) or through-hole packages due to the excessive lead-frame inductance and its effect on device performance.

Conclusion

The HOTLink Transmitter offers the designers of pseudo ECL systems an alternative to the expensive, high-power clock sources currently available on the market. The combination of BiCMOS process technology and robust feature set makes CY7B923 suitable for many PECL logic circuit clock generation applications where cost, power, flexibility, and performance are of prime concern.

References

1. Blood Jr., William R., *MECL System Design Handbook*, Fourth Edition, 1988.
2. Cypress Semiconductor Corporation, *High Performance Databook*, 1993 Edition.
3. Cypress Semiconductor Corporation, HOTLink Design Considerations, Application Note.
4. Cypress Semiconductor Corporation, *Applications Handbook*, 1993 Edition.

Replace Your Am7968 TAXI™ Transmitter With a CY7B923 HOTLink™

Introduction

The TAXI family of data communications parts was one of the first to provide the benefits of high-speed serial transport of parallel information. Because of its flexibility and wide data-rate range, it has found usage in numerous commercial and military applications.

Time, however, has moved on and the original TAXI has in many cases been left behind. The Am7968 is a full bipolar design and consumes over 1W while newer components, like the Cypress HOTLink, are capable of operating at twice the data rate and less than half the power. In addition, the military version of the Am7968 has been discontinued, leaving numerous designs in jeopardy.

Fortunately, a relatively simple replacement is available for the Am7968 that (in most cases) requires little or no change in surrounding system logic, *including* the Am7969 TAXI receiver. This simple replacement uses the Cypress CY7B923 HOTLink Transmitter, along with a small PLD, to form a logic and timing equivalent replacement. The use of such a replacement allows the continued use and manufacture of these legacy systems with minimal impact to the equipment and system interconnect

Overview

The Am7968 TAXI transmitter, when operating in 8-bit mode, uses a 4B/5B encoding scheme to convert input data and commands into a form suitable for serial transmission and clock recovery. Communication with an existing Am7969 TAXI receiver requires the use of this same encoding scheme, presented in the same form and data-rate as that generated by the Am7968. By operating the CY7B923 HOTLink Transmitter in Bypass mode (unencoded 10-bit data path) mated to a small PLD, it is possible to exactly emulate the 4B/5B encoding used by the Am7968.

Am7968 Functionality

The Am7968 is both very similar to the HOTLink transmitter, and very different. Both parts communicate serially over a differential PECL (Positive ECL) link. Both parts employ a PLL clock multiplier to change a slow byte-rate clock into a fast bit-rate clock. However, most of the similarity ends here.

Data Encoding

Unlike HOTLink, which normally operates with an 8B/10B DC-balanced code, the Am7968 encodes its data stream using a 4B/5B algorithm standardized for use with the FDDI (Fiber Distributed Data Interface). This encoding converts four bits of parallel data into five bits of serial data. With such a small a code set to work with, it is not possible to maintain a DC-balance in the data stream. To improve this somewhat, the Am7968 also performs an NRZI (non-return-to-zero, invert on ones) encoding of the serial data.

4B/5B Encoding

The data is encoded to ensure a minimum density of transitions in the serial interface. These transitions are necessary to allow the receive end of the serial link to locate the boundaries of bits on the serial interface. Without this (or a similar) encoding, trans-

mission of a long string of zeros or ones would turn into a DC level on the serial interface. Without any transitions to identify some of the bit boundaries, the receiver clock would eventually drift slightly in frequency and capture incorrect information from the serial interface.

The 4B/5B encoding used with the Am7968 allows all sixteen possible 4-bit data groupings to be represented by 5-bit patterns that all contain transitions. Since the complete 5-bit data space actually contains a total 32 possible combinations, only half of the available patterns are used to represent data. These data combinations are listed in *Table 1.*

Table 1. 4B/5B/NRZI Data Encoding

HEX Data	Binary Data	4B/5B Encoded	0-Carry NRZI	1-Carry NRZI
0	0000	11110	10100	01011
1	0001	01001	01110	10001
2	0010	10100	11000	00111
3	0011	10101	11001	00110
4	0100	01010	01100	10011
5	0101	01011	01101	10010
6	0110	01110	01011	10100
7	0111	01111	01010	01010
8	1000	10010	11100	00011
9	1001	10011	11101	00010
A	1010	10110	11011	00100
B	1011	10111	11010	00101
C	1100	11010	10011	01100
D	1101	11011	10010	01101
E	1110	11100	10111	01000
F	1111	11101	10110	01001

NRZI Encoding

In addition to converting the parallel 4-bit data into serial 5-bit data, a second level of encoding is added to improve its signaling characteristics. This encoding (called NRZI) removes the need to know if a transmitted bit was sent as a one or a zero. This is done by converting 1-bits into inversions in the serial stream, while 0-bits maintain the same HIGH or LOW signal level. Because all 1 and 0 information is now determined only by transitions (not by active level), the serial receiver can now correctly decode the serial data stream even if the differential inputs are swapped.

An example of an NRZI-encoded serial stream and encoder is shown in *Figure 1.* Two different output streams are shown in the figure. Which of the two streams is actually generated is determined by the state of the encoder flip-flop when the NRZI encoding of the current character is started. The two possible NRZI encodings of each 4B/5B data character are also listed in *Table 1.* Notice that these two columns are the exact inverse of each other.

Am7968 Commands

The 4B/5B code makes use of specific patterns from a 32-symbol space. Of these 32 possible symbols, sixteen are allocated to represent the hex data values x'0' through x'F'. This leaves sixteen additional 5-bit patterns that can be assigned meanings other than data.

For the Am7968, eight of the remaining sixteen patterns are used to define synchronization and in-band command codes that can be used for various interface control functions. These eight patterns are identified as other alphabetic letters, similar to the hexadecimal characters greater than 9. These control code names and their associated encodings are listed in *Table 2.*

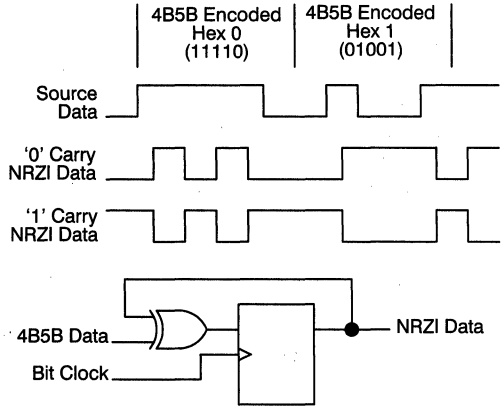


Figure 1. NRZI Encoder

Table 2. 4B/5B/NRZI Control Code Encoding

Control Code	4B/5B Encoded	0-Carry NRZI	1-Carry NRZI
H	00100	00111	11000
I	11111	10101	01010
J	11000	10000	01111
K	10001	11110	00001
Q	00000	00000	11111
R	00111	00101	11010
S	11001	10001	01110
T	01101	01001	10110

Unlike the data characters, which can be combined in any fashion to transmit bytes of information, the Control Codes are only defined for use in specific pair combinations. These control code pairings are generated when specific combinations of bits are present on the four command input lines to the Am7968. These command input groupings are listed in *Table 3.*

Table 3. Am7968 Command Codes

HEX Command	Binary Command	Control Code Pair
0	0000	Data
No STRB	No STRB	JK (8-bit Sync)
1	0001	II
2	0010	TT
3	0011	TS
4	0100	IH
5	0101	TR
6	0110	SR
7	0111	SS
8	1000	HH
9	1001	HI
A	1010	HQ
B	1011	RR
C	1100	RS
D	1101	QH
E	1110	QI
F	1111	QQ

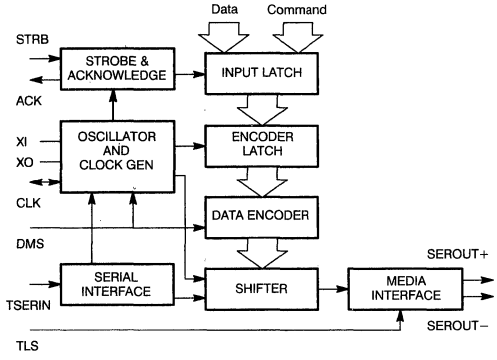


Figure 2. Am7968 Logic Diagram

Am7968 Control Signals

A block diagram of the Am7968 is shown in *Figure 2.* This figure shows the control signals and data/command buses used to control the part. Unlike the CY7B923 HOTLink transmitter (see *Figure 3*), the Am7968 has separate input buses for data and commands. The data input bus is eight bits in width while the command bus is only four bits wide.

Loading of data into the Am7968 is also handled differently. This is performed using the STRB input to clock the information present in the data and command buses into the the Am7968. This STRB signal may be semi-asynchronous to the normal transmitter reference clock on the X1 input.

To operate the Am7968 at or near its reference clock byte rate it is necessary to strobe data into the part with much more care than when operating at slower

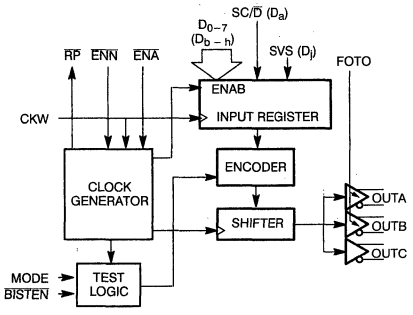


Figure 3. CY7B923 Transmitter Logic Diagram

rates. There is, in effect, a "stayout" area around the falling edge of the reference clock where data and commands should not be strobed into the part.

HOTLink Emulation of Am7968

To create a drop-in replacement for a part, it is necessary to present an interface to the host system that contains the same signals, clocks, and timing as the logic element being replaced. In the case of the Am7968, the critical signals used for operation are

- DI[7:0]—eight-bit data bus
- CI[3:0]—four-bit command bus
- STRB—data strobe
- ACK—data strobe acknowledge
- ±SEROUT—differential PECL serial data
- X1—external byte reference clock

While there are other signals present on the Am7968, they are primarily static signals used for configuration.

Emulator Block Diagram

The emulator is built from two components, as shown in *Figure 4*: a CY7C343 EPLD that performs the 4B/5B and NRZI encoding, and a CY7B923 HOTLink transmitter to sequence the bits and drive the serial PECL interface. This two-chip design assumes that double frequency byte clock is present in the system to clock both the EPLD and the HOTLink transmitter. For those systems that only have the byte-rate clock present, it is possible to generate the 2x clock using a single Cypress CY7B991 Robo-Clock Programmable Skew Clock Buffer.

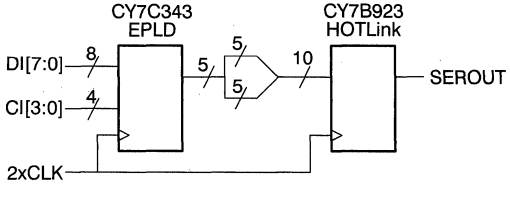


Figure 4. Am7968 Emulator Block Diagram

The 2x clock is necessary in the system because the HOTLink transmitter is normally only capable of sequencing bits with the data rate range of 160 to 330 Mbits/second. This is significantly faster than the maximum 125-Mbit/second data rate of the Am7968 transmitter. To allow the HOTLink transmitter to generate a serial stream that is data-rate compatible with an attached Am7969 receiver requires sequencing out bits in pairs. This effectively cuts the data rate of the transmitter in half. This timing relationship is shown in *Figure 5*.

This bit timing is accomplished by having the encoder EPLD generate only five NRZI bits on each 2x clock cycle. Each of these five bits is attached to two adjacent bit-inputs on the HOTLink transmitter. For example, encoder output bit-0 would be wired to HOTLink transmitter bits 0 and 1,

Emulator PLD Block Diagram

The majority of the emulator signals are on the parallel TTL-compatible side of the design. These parallel signals (all except the PECL ±SEROUT signals) all tie into the CY7C343 control EPLD. This EPLD performs all the data capture, 4B/5B encoding, NRZI encoding, and byte timing for the emulator. A block diagram of the internal functions of the EPLD is shown in *Figure 6*.

The logic is effectively split into five major sections. These sections control the data capture, holding register, 4B/5B/NRZI encoding, NRZI carry encoding, and clocking.

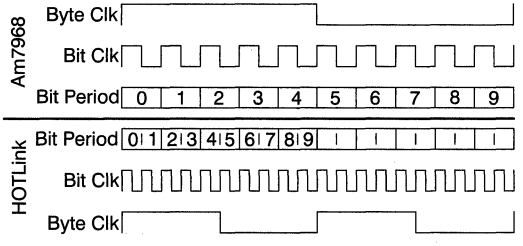


Figure 5. Am7968 vs CY7B923 Bit Timing

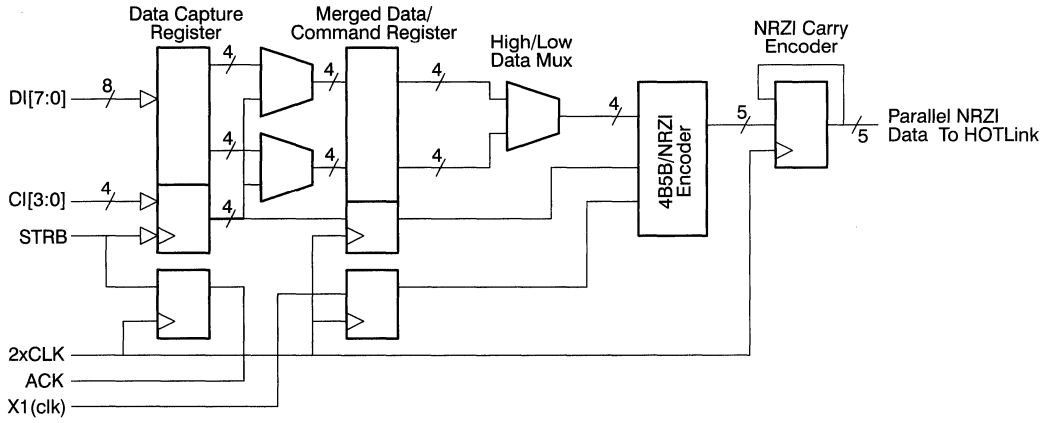


Figure 6. 4B/5B/NRZI Encoder PLD Block Diagram

Control EPLD Operation

Data Capture Register

Data is loaded into the 12-bit Data Capture register on the rising edge of any STRB pulse. Once latched, the contents of the CI[3:0] bits determine what data is fed to the Merged Data/Command register. If any of the CI[3:0] bits are HIGH the CI bus is fed to both the upper and lower halves of the register. If all CI bits are LOW, the DI[7:0] data bus is fed to the register instead.

Merged Data/Command Register

The Merged Data/Command register is a 9-bit register that is loaded every other cycle of the 2xCLK. The upper eight bits of this register are loaded with the output of the multiplexer from the data Capture register. The lowest bit identifies if the data in the register is a command or data.

If a STRB has occurred to load data into the Data Capture register during the previous cycle, that information is clocked into the Merged Data/Command register. If a STRB has not occurred, then a x'00' command is forced into the Merged Data/ Command register.

4B/5B/NRZI Encoder

The data in the Merged Data/Command register is sequenced through the 4B/5B/NRZI encoder in two four-bit groups. The first group encodes the upper four bits of the command or data byte, while the second group encodes the lower four bits. In addition to the data bits, the encoder also needs to know if the bits represent a command or data, and (for commands) if the information is the upper or lower half-byte.

The NRZI output of the encoder assumes a zero for the starting or carry-in state of the NRZI encode operation. By pre-encoding the NRZI information, a large number of XOR gates can be removed from the design.

NRZI Carry Encoder

To generate the correct NRZI sequence it is necessary to track the state of the previous bit in the output sequence. This is done by feeding the most significant bit of the output register back to the input of the register, and XORing it with the next five bits of information. This effectively performs a selective inversion of the pre-encoded NRZI data. This inversion allows the data output to follow the NRZI encoding listed in *Tables 1* and *2*.

Clocking

In the implementation documented here, this design uses two independent clocks: one for the STRB

signal and the 2xCLK for the remainder of the logic. In addition to these two clocks, the EPLD monitors the X1 clock to determine which phase of the 2xCLK to capture and mux the internal data.

Conclusion

This design implements a two-chip drop-in replacement for the Am7968 TAXI transmitter. The design makes use of programmable logic to implement an external encoder that mimics the interface and timing of the Am7968.

The control EPLD was implemented using a CY7C343 EPLD. This PLD was designed and coded with VHDL (VHSIC Hardware Description Language), and compiled and simulated using the Cypress *Warp3*™ tool. The full source code for the design is present in Appendix A of this application note, and is available from the Cypress electronic Bulletin Board System (BBS).

For those Am7968-based systems that are truly synchronous in nature, this design may be modified to operate with a single clock, and allow usage of the FLASH370™ family of CPLDs in addition to the CY7C34x series.

Because of the modularity and reusability of VHDL code, it is possible to incorporate the code in Appendix A with additional functionality in larger or more complex CPLDs or FPGAs, thereby reducing the hardware impact of this emulation to a reprogrammed logic part and a simple replacement of the Am7968 with the more capable CY7B923. Such a system would then be able to support a much faster data rate in the future with the simple reprogramming of the controlling PLD.

References

1. Cypress Semiconductor, CY7B923/CY7B933 HOTLink Transmitter/Receiver Datasheet, Cypress Semiconductor *Data Book*, May, 1995.
2. Cypress Semiconductor, *HOTLink User's Guide*, 2nd Edition, June 1995.
3. Advanced Micro Devices, TAXIchip Integrated Circuits Transparent Asynchronous Transmitter/Receiver Interface Am7968/Am7969–125 Am7968/Am7969–175 Data Sheet and Technical Manual, 1992

Appendix A. 4B/5B Encoder PLD

```
-- TAXI8SM.VHD

-- This design describes the operation of a PLD used to convert a
-- standard HOTLink transmitter (CY7B923) into a part set equivalent
-- to the older AMD TAXI-125.  This PLD only emulates the TAXI
-- in 8-bit mode (dual 4B/5B encoders).

-- This design only operates in the standard synchronous mode
-- of the TAXI, as it does not contain any FIFO stages.  It does
-- correctly generate all 16 TAXI command codes present.  It does
-- not support cascade mode.
ENTITY taxi8top IS PORT (
  -- TAXI Parallel-side pins
  clk: IN BIT;                    -- PLD Clock, 2X multiple of
                                  --  standard TAXI clock
  sys_clk: IN BIT;                -- standard TAXI clock, sampled
                                  --  by the PLD for phase alignment
  strobe: IN BIT;                 -- TAXI data load clock, used
                                  -- to control loading of the
                                  -- input register.  Needs to
                                  -- be interruptible to force
                                  -- generation of SYNC codes
  D_In: IN BIT_VECTOR(0 TO 7);    -- data input bus
  CL: IN BIT_VECTOR(0 TO 3);      -- command input bus
  -- HOTLink parallel-side pins
  D_Out: OUT BIT_VECTOR(0 TO 4)   -- HOTLink data inputs, two/pin
  );
ATTRIBUTE part_name OF taxi8top:ENTITY IS "C343";
END taxi8top;

USE work.cypress.all;
USE work.table_bv.all;
USE work.rtlpkg.all;
USE work.memorypkg.all;

ARCHITECTURE struct OF taxi8top IS
-- add internal signals
SIGNAL outreg : BIT_VECTOR(0 TO 4);   -- output data register
SIGNAL encode : BIT_VECTOR(0 TO 4);   -- 4B/5B/NRZI  encoder output
SIGNAL xreg   : BIT_VECTOR(0 TO 4);   -- output XOR register
SIGNAL in_reg : BIT_VECTOR(0 TO 11); -- 12-bit input register
SIGNAL hld_reg: BIT_VECTOR(0 TO 8);   -- data input hold register
SIGNAL in_data: BIT_VECTOR(0 TO 5);   -- encoder input
SIGNAL strb_in: BIT;                  -- strobe received flag
SIGNAL strb_n:  BIT;                  -- inverted strobe
SIGNAL phase1:  BIT;                  -- hold enable for STROBE in
```

Appendix A. 4B/5B Encoder PLD (continued)

```
-- 4B/5B encoder data constants
-- data half-bytes
CONSTANT DI_0: x01_VECTOR(0 TO 4) := "00000";
CONSTANT DI_1: x01_VECTOR(0 TO 4) := "00001";
CONSTANT DI_2: x01_VECTOR(0 TO 4) := "00010";
CONSTANT DI_3: x01_VECTOR(0 TO 4) := "00011";
CONSTANT DI_4: x01_VECTOR(0 TO 4) := "00100";
CONSTANT DI_5: x01_VECTOR(0 TO 4) := "00101";
CONSTANT DI_6: x01_VECTOR(0 TO 4) := "00110";
CONSTANT DI_7: x01_VECTOR(0 TO 4) := "00111";
CONSTANT DI_8: x01_VECTOR(0 TO 4) := "01000";
CONSTANT DI_9: x01_VECTOR(0 TO 4) := "01001";
CONSTANT DI_A: x01_VECTOR(0 TO 4) := "01010";
CONSTANT DI_B: x01_VECTOR(0 TO 4) := "01011";
CONSTANT DI_C: x01_VECTOR(0 TO 4) := "01100";
CONSTANT DI_D: x01_VECTOR(0 TO 4) := "01101";
CONSTANT DI_E: x01_VECTOR(0 TO 4) := "01110";
CONSTANT DI_F: x01_VECTOR(0 TO 4) := "01111";

-- command constants
CONSTANT CI_0: x01_VECTOR(0 TO 4) := "10000";
CONSTANT CI_1: x01_VECTOR(0 TO 4) := "10001";
CONSTANT CI_2: x01_VECTOR(0 TO 4) := "10010";
CONSTANT CI_3: x01_VECTOR(0 TO 4) := "10011";
CONSTANT CI_4: x01_VECTOR(0 TO 4) := "10100";
CONSTANT CI_5: x01_VECTOR(0 TO 4) := "10101";
CONSTANT CI_6: x01_VECTOR(0 TO 4) := "10110";
CONSTANT CI_7: x01_VECTOR(0 TO 4) := "10111";
CONSTANT CI_8: x01_VECTOR(0 TO 4) := "11000";
CONSTANT CI_9: x01_VECTOR(0 TO 4) := "11001";
CONSTANT CI_A: x01_VECTOR(0 TO 4) := "11010";
CONSTANT CI_B: x01_VECTOR(0 TO 4) := "11011";
CONSTANT CI_C: x01_VECTOR(0 TO 4) := "11100";
CONSTANT CI_D: x01_VECTOR(0 TO 4) := "11101";
CONSTANT CI_E: x01_VECTOR(0 TO 4) := "11110";
CONSTANT CI_F: x01_VECTOR(0 TO 4) := "11111";

-- data output constants
-- zero carry-in, NRZI encoded
CONSTANT DO_0: x01_VECTOR(0 TO 4) := "10100";   -- 11110 4B/5B
CONSTANT DO_1: x01_VECTOR(0 TO 4) := "01110";   -- 01001 4B/5B
CONSTANT DO_2: x01_VECTOR(0 TO 4) := "11000";   -- 10100 4B/5B
CONSTANT DO_3: x01_VECTOR(0 TO 4) := "11001";   -- 10101 4B/5B
CONSTANT DO_4: x01_VECTOR(0 TO 4) := "01100";   -- 01010 4B/5B
CONSTANT DO_5: x01_VECTOR(0 TO 4) := "01101";   -- 01011 4B/5B
CONSTANT DO_6: x01_VECTOR(0 TO 4) := "01011";   -- 01110 4B/5B
CONSTANT DO_7: x01_VECTOR(0 TO 4) := "01010";   -- 01111 4B/5B
```

Appendix A. 4B/5B Encoder PLD (continued)

```
CONSTANT DO_8: x01_VECTOR(0 TO 4) := "11100";   -- 10010 4B/5B
CONSTANT DO_9: x01_VECTOR(0 TO 4) := "11101";   -- 10011 4B/5B
CONSTANT DO_A: x01_VECTOR(0 TO 4) := "11011";   -- 10110 4B/5B
CONSTANT DO_B: x01_VECTOR(0 TO 4) := "11010";   -- 10111 4B/5B
CONSTANT DO_C: x01_VECTOR(0 TO 4) := "10011";   -- 11010 4B/5B
CONSTANT DO_D: x01_VECTOR(0 TO 4) := "10010";   -- 11011 4B/5B
CONSTANT DO_E: x01_VECTOR(0 TO 4) := "10111";   -- 11100 4B/5B
CONSTANT DO_F: x01_VECTOR(0 TO 4) := "10110";   -- 11101 4B/5B
CONSTANT DO_H: x01_VECTOR(0 TO 4) := "00111";   -- 00100 4B/5B
CONSTANT DO_I: x01_VECTOR(0 TO 4) := "10101";   -- 11111 4B/5B
CONSTANT DO_J: x01_VECTOR(0 TO 4) := "10000";   -- 11000 4B/5B
CONSTANT DO_K: x01_VECTOR(0 TO 4) := "11110";   -- 10001 4B/5B
CONSTANT DO_Q: x01_VECTOR(0 TO 4) := "00000";   -- 00000 4B/5B
CONSTANT DO_R: x01_VECTOR(0 TO 4) := "00101";   -- 00111 4B/5B
CONSTANT DO_S: x01_VECTOR(0 TO 4) := "10001";   -- 11001 4B/5B
CONSTANT DO_T: x01_VECTOR(0 TO 4) := "01001";   -- 01101 4B/5B

-- generate decoder table
CONSTANT table: x01_TABLE(0 TO 41, 0 TO 10) := (
-- data mappings
--
--Input    HI_LO    Output
-------    -----    ------
  DI_0  &  'x'  &  DO_0,
  DI_1  &  'x'  &  DO_1,
  DI_2  &  'x'  &  DO_2,
  DI_3  &  'x'  &  DO_3,
  DI_4  &  'x'  &  DO_4,
  DI_5  &  'x'  &  DO_5,
  DI_6  &  'x'  &  DO_6,
  DI_7  &  'x'  &  DO_7,
  DI_8  &  'x'  &  DO_8,
  DI_9  &  'x'  &  DO_9,
  DI_A  &  'x'  &  DO_A,
  DI_B  &  'x'  &  DO_B,
  DI_C  &  'x'  &  DO_C,
  DI_D  &  'x'  &  DO_D,
  DI_E  &  'x'  &  DO_E,
  DI_F  &  'x'  &  DO_F,

  CI_0  &  '1'  &  DO_J,
  CI_0  &  '0'  &  DO_K,
  CI_1  &  'x'  &  DO_I,
  CI_2  &  'x'  &  DO_T,
  CI_3  &  '1'  &  DO_T,
  CI_3  &  '0'  &  DO_S,
  CI_4  &  '1'  &  DO_I,
```

Appendix A. 4B/5B Encoder PLD (continued)

```
  CI_4  &  '0'  &  DO_H,
  CI_5  &  '1'  &  DO_T,
  CI_5  &  '0'  &  DO_R,
  CI_6  &  '1'  &  DO_S,
  CI_6  &  '0'  &  DO_R,
  CI_7  &  'x'  &  DO_S,
  CI_8  &  'x'  &  DO_H,
  CI_9  &  '1'  &  DO_H,
  CI_9  &  '0'  &  DO_I,
  CI_A  &  '1'  &  DO_H,
  CI_A  &  '0'  &  DO_Q,
  CI_B  &  'x'  &  DO_R,
  CI_C  &  '1'  &  DO_R,
  CI_C  &  '0'  &  DO_S,
  CI_D  &  '1'  &  DO_Q,
  CI_D  &  '0'  &  DO_H,
  CI_E  &  '1'  &  DO_Q,
  CI_E  &  '0'  &  DO_I,
  CI_F  &  'x'  &  DO_Q);

BEGIN
-- declare input register.  Data is clocked by the external STROBE
-- signal.  This same strobe signal is used to synchronize the internal
-- two-state machine.

p1: PROCESS BEGIN
  WAIT UNTIL (strobe='1');
    in_reg(0 TO 7) <= D_In(0 TO 7);
    in_reg(8 TO 11) <= CL(0 TO 3);
END PROCESS p1;-- capture strobe event

-- async set when strobe is present
-- use synchronous clear from clk when part is set and sys_clk present
phase1 <= strb_in AND sys_clk;
st1: DSRFF PORT MAP  (phase1, strobe, zero, clk, strb_in);

-- setup input data hold register
p2: PROCESS BEGIN
  WAIT UNTIL (clk='1');
    IF sys_clk = '0' THEN       -- hold data
      hld_reg <= hld_reg;
    ELSIF strb_in='0' THEN       -- no data, load a SYNC command
      hld_reg <= "000000001";
    ELSIF (in_reg(8 TO 11) /= "0000") THEN  -- check for a command
      hld_reg(0 TO 3) <= in_reg(8 TO 11);
      hld_reg(4 TO 7) <= in_reg(8 TO 11);
      hld_reg(8) <= '1'; -- set as a command
    ELSE
```

Appendix A. 4B/5B Encoder PLD (continued)

```
      hld_reg(0 TO 7) <= in_reg(0 TO 7);
      hld_reg(8) <= '0'; -- set as data
  END IF;
END PROCESS p2;

-- declare data mux select for input to the 4B/5B encoder
p3: PROCESS (hld_reg, sys_clk)
  BEGIN
  in_data(5) <= NOT sys_clk;      -- hi/low nibble select
  in_data(0) <= hld_reg(8);
  IF sys_clk = '0' THEN           -- enable high nibble first
    in_data(1 TO 4) <= hld_reg(4 TO 7);
  ELSE
    in_data(1 TO 4) <= hld_reg(0 TO 3);
  END IF;
END PROCESS p3;

-- declare 4B/5B encoder
p4: PROCESS (in_data)
BEGIN
  encode <= ttf(table,(in_data));
END PROCESS p4;

-- declare output register

dr0: DFF PORT MAP (encode(0), clk, outreg(0));
dr1: DFF PORT MAP (encode(1), clk, outreg(1));
dr2: DFF PORT MAP (encode(2), clk, outreg(2));
dr3: DFF PORT MAP (encode(3), clk, outreg(3));
dr4: DFF PORT MAP (encode(4), clk, outreg(4));

dx0: XDFF PORT MAP (outreg(0), xreg(4), clk, xreg(0));
dx1: XDFF PORT MAP (outreg(1), xreg(4), clk, xreg(1));
dx2: XDFF PORT MAP (outreg(2), xreg(4), clk, xreg(2));
dx3: XDFF PORT MAP (outreg(3), xreg(4), clk, xreg(3));
dx4: XDFF PORT MAP (outreg(4), xreg(4), clk, xreg(4));

-- assign output register to outputs
D_Out <= xreg;
END struct;     -- end of top level design
```

Upgrade Your TAXI–275™ with HOTLink™

This application note will explain how to upgrade TAXI–275™ (Am79168/Am79169) devices with the HOTLink™ (CY7B923/CY7B933) devices from Cypress Semiconductor. It will aid in the migration of TAXI–275 designs to the HOTLink architecture. This note begins with an introduction to HOTLink and then gives advantages of HOTLink and replacement suggestions for the TAXI–275 devices.

HOTLink Introduction

The HOTLink family of devices transfers data from point to point over high-speed serial links at 160 to 330 Mbits/second (*Figure 1*). The CY7B923 Transmitter (*Figure 2*) takes an 8-bit parallel data stream and encodes it using the Fibre Channel and ESCON compliant 8B/10B code. This code maps all 8-bit data characters into a 10-bit transmission code that ensures that the transmission signal contains suitable transitions for recovery by the receiving device. The transmitter then takes this 10-bit data word and converts it to a serial bit stream and sends it at 10 times the byte rate over a serial transmission link.

The CY7B933 HOTLink Receiver (*Figure 3*) connects to the other end of a transmission link that may consist of anything from a few inches of printed circuit board trace to several kilometers of fiber-optic cable. The receiver decodes the incoming bit stream and reconstructs the original parallel data character, which is presented at the outputs and aligned with the recovered clock. The receiver, in addition to these tasks, checks the incoming data stream for errors that may have occurred in the serial transmission.

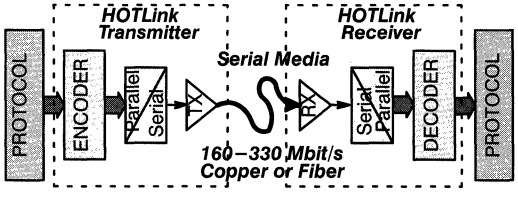


Figure 1. HOTLink System Diagram

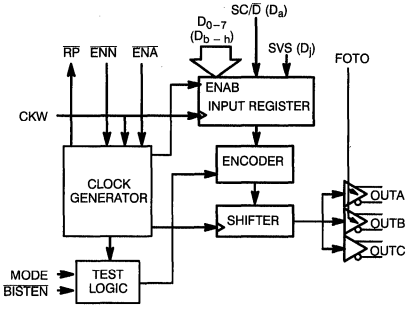


Figure 2. CY7B923 Transmitter Logic Diagram

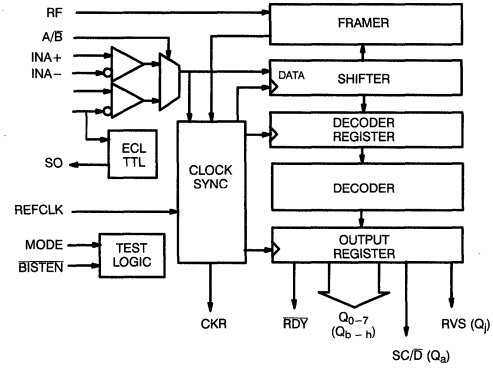


Figure 3. CY7B933 Receiver Logic Diagram

The SC/$\overline{D}$ (Special Character/$\overline{Data}$) pin permits the transmission of command codes in addition to data characters. The codes are mapped to 10-bit transmission characters defined in the 8B/10B codes of the Fibre Channel standard. Commands can be sent as part of the transmission stream, to signal events such as Idle, Start-of-frame, End-of-frame, etc.

Other features provide a complete solution for high-speed point-to-point communication in applications including interconnecting workstations, servers, mass storage, and video transmission equipment. These features include built-in self-test (BIST) for in-system diagnostic testing, unencoded mode for sending 10-bit data in systems that use a different encoding method, and a seamless parallel interface for connection to both asynchronous and clocked FIFOs. A brief description of the various features of HOTLink are given below with a more detailed discussion found in the CY7B923/CY7B933 HOTLink Transmitter/Receiver datasheet. The PLCC pinouts for these devices are shown in *Figure 4*.

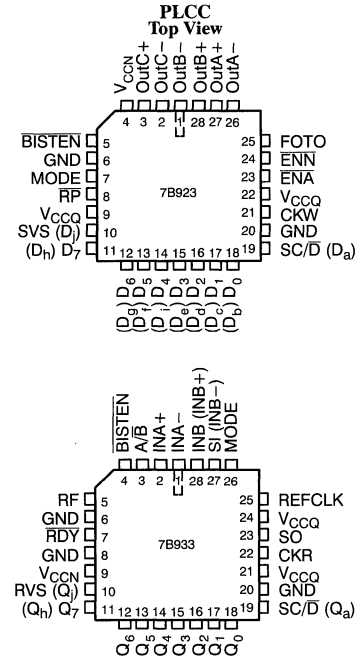


Figure 4. CY7B923 and CY7B933 Pin Configurations

Upgrade from TAXI−275

The following sections explain the architectural advantages of the Cypress CY7B923/CY7B933 HOTLink Transmitter and Receiver over the devices from AMD. This section begins with a brief explanation of the Am79168/Am79169 TAXI−275 devices. It then follows with a list of HOTLink features that make designing these high-speed point-to-point systems easier.

A Brief Explanation of TAXI−275

The Am79168/Am79169 TAXI−275 devices are similar to HOTLink. The Am79168 TAXI−275 Transmitter, shown in *Figure 5*, converts 8-bit or 10-bit parallel data into 10- or 12-bit transmission codes using either the 8B/10B code or the 10B/12B code. This data is encoded and shifted out serially over a transmission link operating at speeds of 175 to 275 Mbaud. The Am79169 TAXI−275 Receiver, shown in *Figure 6*, converts the incoming serial data into parallel words, and decodes and presents the original words in 8-bit or 10-bit format to the outputs along with the recovered clock. The pinouts of the Am79168 Transmitter and the Am79169 Receiver are shown in *Figure 7*.

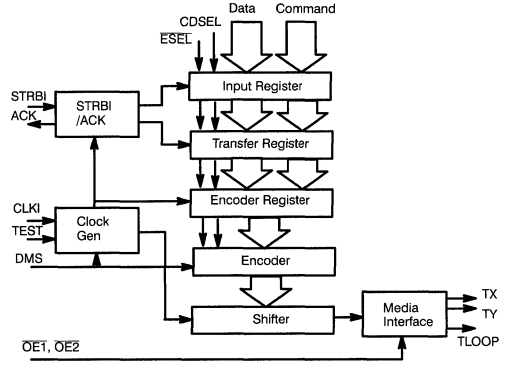


Figure 5. TAXI-275 Transmitter Block Diagram

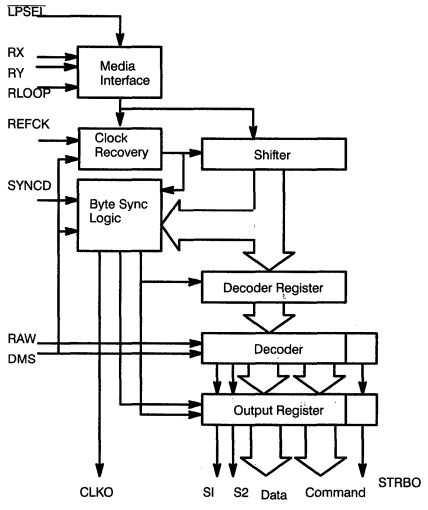


Figure 6. TAXI−275 Receiver Block Diagram

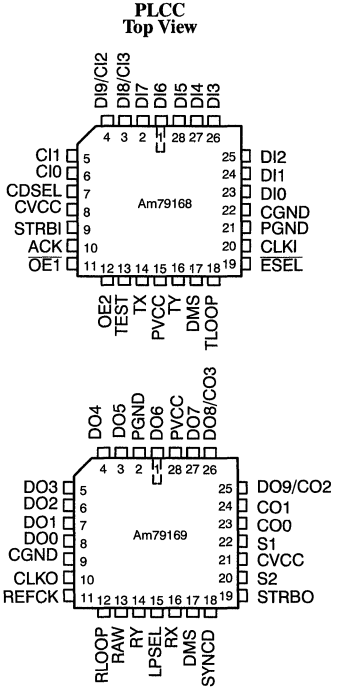


Figure 7. Am79168 and Am79169 Pin Configurations

Simplifying Your System with HOTLink

HOTLink offers additional features that will simplify system design. Below is a list of these features along with their benefits when designing high-speed point-to-point serial communications systems with Cypress HOTLink devices.

Multiplexed Command and Data

The TAXI−275 has separate inputs for command and data, while the HOTLink devices have an integrated command and data path. The status of SC/$\overline{D}$ pin (Special Character/$\overline{Data}$) determines if HOTLink sends a Special Character (Command) or data.

The integrated command and data paths of HOTLink allow simplification of the controller architecture. Instead of creating a separate command path, command codes can be integrated within the data stream with the addition of a ninth bit (the SC/$\overline{D}$) bit that indicates the status of the associated 8 bits of information.

More Outputs

The HOTLink transmitter has three identical differential Positive ECL (PECL) serial output ports. Two of these outputs can be turned off under control of FOTO (Fiberoptic Transmitter Off) pin. The TAXI−275 devices have only one differential PECL output pair and an additional single-ended TLOOP output intended for use in loop-back testing.

More Inputs

The HOTLink Receiver has two differential interfaces to the serial transmission medium (INA± and INB±) whereas the TAXI−275 devices have only a single input pair (RX,RY) and a single ended PECL input, RLOOP, used for loop-back testing. The media inputs of the HOTLink Receiver can be used to provide loop-back testing, redundant transmission paths, or more complex networks configurations.

Loop-back testing ensures that a node is sending and receiving data properly. In a typical network-style configuration, both the transmitter and receiver will exist for each node. In loop-back testing a redundant output from the transmitter is fed back to the additional input on the receiver. The TAXI−275 devices have an extra single-ended 100K

PECL output, TLOOP, and an extra single-ended 100K PECL input, RLOOP, that are used for loop-back testing. The HOTLink devices offer a more robust loop-back capability by offering redundant differential output pairs that can be connected to an additional differential input structure on the receiver, as shown in *Figure 8*. The additional single-ended input/output pair of the TAXI−275 does not provide a robust loop-back testing configuration.

In addition, the redundant outputs of the transmitter can be used in conjunction with the additional inputs of the receiver to build more complex network structures by allowing a single transmitter to communicate with multiple receivers, or a single receiver to be connected with multiple transmitters. The multiple outputs can also be used to to build redundant paths between two nodes.

More Flexible Command Codes

A coding system is necessary in serial communication systems to ensure that the receiving device can determine the boundary between adjacent bits. The code makes sure that enough signal transitions exist on the transmission channel to track bit boundaries. In other words, the code must ensure that the clock used by the transmitter to transmit the data is embedded within the data stream. The code maps each character into a code word that ensures that a minimum transition density and run length is maintained.

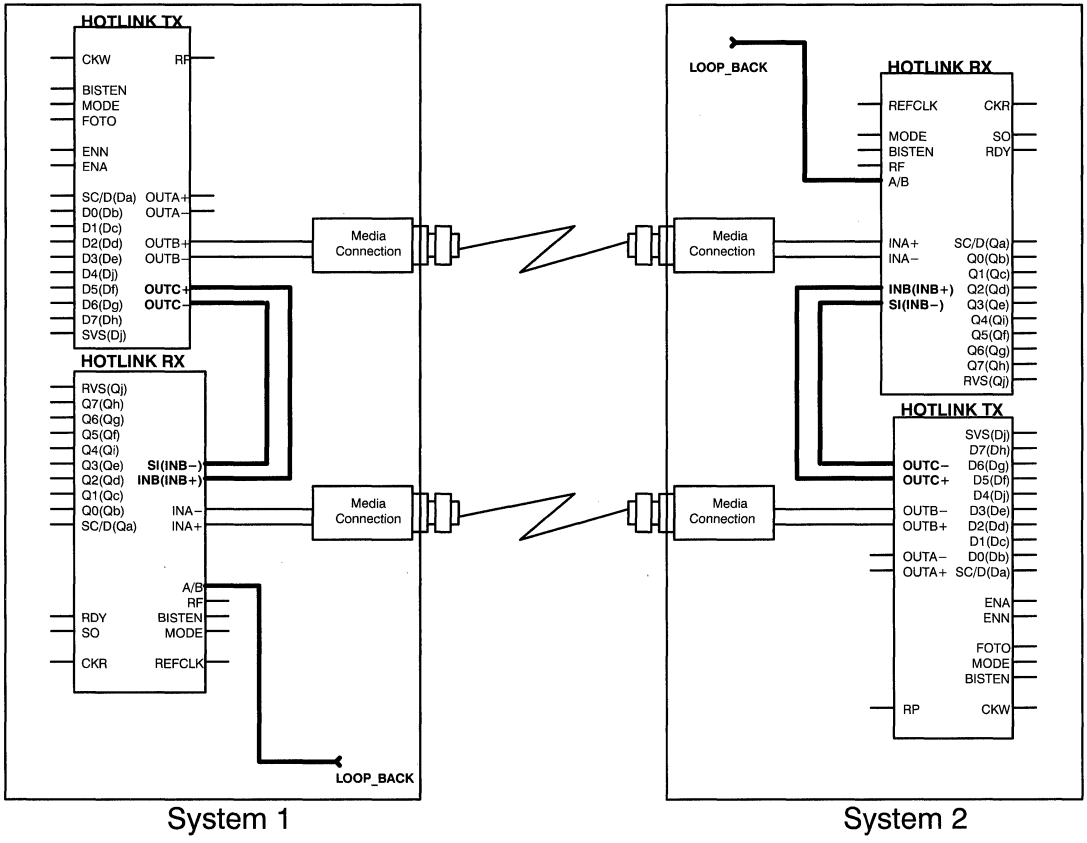


Figure 8. Example HOTLink Loop-Back System Connection

Both HOTLink and TAXI−275 use the transmission code specified by ANSI X3T9.3 Fibre Channel and IBM ESCON standards. This code converts 8 bits into 10 transmission bits (8B/10B). This code generates NRZ (Non Return to Zero) transmission data where a logical 1 is represented by a HIGH level and a logic 0 is represented by a LOW level. The complete code tables are listed at the end of the CY7B923/CY7B933 HOTLink Transmitter/Receiver datasheet. This code ensures not only minimum transition density and run length, but also that the average number of 1s and 0s are equal. This feature of the code prevents the average DC level on the transmission link from "wandering" based on the data that is being sent.

In addition to specifying a mapping of every character into a transmission symbol, the code also specifies several command codes. These codes are useful for low level signaling without involving higher level protocols. They can be used to indicate information such as HALT, End Of Frame, or Start of Frame.

Table 1 shows the valid special characters and sequences that HOTLink can both encode on the transmitting end and decode on the receiving end. The first column in the table indicates the byte name of the special character. In the Fibre Channel and ESCON notation Special Characters are denoted with a 'K' prefix and Data Characters are denoted with a 'D' prefix. The first twelve Special Characters are defined in the Fibre Channel and ESCON specifications. The second column of the table gives the code name, both in decimal and hexadecimal notation, of the binary pattern on the I/O pins. The third column, bits, shows the pattern presented to the transmitter's data lines. This pattern, in combination with SC/$\overline{D}$ HIGH, will cause either the pattern in column four or column five to be sent. The pattern that the transmitter sends depends on the current Running Disparity.

In order to ensure that the average number of 1s and 0s that are sent across the communications channel is equal, both the transmitter and receiver keep track of the Running Disparity of the data that was previously sent. Running Disparity (RD) can either be positive (+) or negative (−). In general, Running Disparity will be positive if, in the last transmission word, there were more 1s sent than 0s and it will be negative if there were more 0s sent than 1s. If RD is negative, the transmitter will send the code in column four and if RD is positive then the transmitter will send the code in column five.

Both HOTLink and TAXI−275 can send all codes labeled C0.0 through C11.0 in *Table 1*. Because of the different architectures of these two devices, the data presented to the inputs of the transmitter will be different, but the code sent across the transmission medium will be identical.

The next three codes represent sequences that the transmitter can send. For example, if the transmitter controller presents C0.1 (binary pattern 001 00000) to the data lines, then the transmitter will send −K28.5+, D21.4, D21.5, D21.5. In other words, the transmitter will send a negative K28.5 Special Character, a D21.4 (binary 100 10101) Data Character, and two D21.5 (binary 101 10101) Data Characters. It will continue to send this pattern as long as C0.1 is present at its inputs. The receiver will decode this pattern as a C1.7 or C5.0 depending on its current Running Disparity followed by D21.4 and two D21.5s. This pattern is defined in the Fibre Channel standard as the IDLE pattern. The ability of the transmitter to send this pattern as well as the R_RDY (Receiver Ready) pattern greatly simplifies controller design. The TAXI−275 devices have no ability to send complex data patterns with a single code as shown by the word **NONE** in *Table 1* under the TAXI−275 Code column.

In addition, if C2.1 is presented to the transmitter, it will send either a negative K28.5 or a positive K28.5, depending on the Running Disparity. It will then modify the Least Significant Bit (LSB) of the subsequent data word to be either a 0 if RD was (−) or a 1 if RD was (+). This simplifies controllers when building End of Frame (EOF) delimiters where the second byte is determined by the current RD. These packet structures are necessary to conform with the Fibre Channel specification. The TAXI−275 device only has the capability of modifying the LSB of two different Data Characters, limiting the possible EOF delimiters that can be constructed.

Table 1. HOTLink Valid Special Character Codes and Sequences (SC/$\overline{D}$ = HIGH)

HOTLink Special Code Byte Name	Special Code Code Name	Bits HGF	Bits EDCBA	Current RD− abcdei	Current RD− fghj	Current RD+ abcdei	Current RD+ fghj	Receiver Output Code Name	TAXI Code
K28.0	C0.0 (C00)	000	00000	001111	0100	110000	1011	C0.0	K28.0
K28.1	C1.0 (C01)	000	00001	001111	1001	110000	0110	C1.0	K28.1
K28.2	C2.0 (C02)	000	00010	001111	0101	110000	1010	C2.0	K28.2
K28.3	C3.0 (C03)	000	00011	001111	0011	110000	1100	C3.0	K28.3
K28.4	C4.0 (C04)	000	00100	001111	0010	110000	1101	C4.0	K28.4
K28.5	C5.0 (C05)	000	00101	001111	1010	110000	0101	C5.0	K28.5
K28.6	C6.0 (C06)	000	00110	001111	0110	110000	1001	C6.0	K28.6
K28.7	C7.0 (C07)	000	00111	001111	1000	110000	0111	C7.0	K28.7
K23.7	C8.0 (C08)	000	01000	111010	1000	000101	0111	C8.0	K23.7
K27.7	C9.0 (C09)	000	01001	110110	1000	001001	0111	C9.0	K27.7
K29.7	C10.0 (C0A)	000	01010	101110	1000	010001	0111	C10.0	K29.7
K30.7	C11.0 (C0B)	000	01011	011110	1000	100001	0111	C11.0	K30.7
Sequences									
Idle	C0.1 (C20)	001	00000	-K28.5+,D21.4,D21.5,D21.5, repeat				C5.0, D21.4, D21.5, D21.5	**NONE**
R_RDY	C1.1 (C21)	001	00001	-K28.5+,D21.4,D10.2,D10.2, repeat				C5.0, D21.4, D10.2, D10.2	**NONE**
EOFxx	C2.1 (C22)	001	00010	-K28.5,Dn.xxx0		+K28.5,Dn.xxx1		C5.0,Dn.xxx0 or C5.0,Dn.xxx1	**NONE**
Follows K28.1 for ESCON Connect-SOF (Rx indication only)									
C-SOF	C7.1 (C27)	001	00111	001111	1000	110000	0111	C7.1	**NONE**
Follows K28.5 for ESCON Passive-SOF (Rx indication only)									
P-SOF	C7.2 (C47)	010	00111	001111	1000	110000	0111	C7.2	**NONE**
Code Rule Violation and SVS Tx Pattern									
Exception	C0.7 (CE0)	111	00000	100111	1000	011000	0111	C0.7	**NONE**
-K28.5	C1.7 (CE1)	111	00001	001111	1010	001111	1010	C5.0 or C1.7	K28.5+
+K28.5	C2.7 (CE2)	111	00010	110000	0101	110000	0101	C5.0 or C2.7	**NONE**
Running Disparity Violation Pattern									
Exception	C4.7 (CE4)	111	00100	110111	0101	001000	1010	C4.7	**NONE**

C7.1 sends the ESCON Connect-Start of Frame (SOF) delimiter and C7.2 sends the Passive-SOF delimiter. C0.7 sends a deliberate code rule violation and has the same effect as having the SVS (Send Violation Symbol) pin HIGH during a character transmission. C1.7 sends a negative K28.5 regardless of the current running disparity. The receiver will decode this as either a C5.0 if its current RD was negative or as a C1.7 if its current RD was positive. C2.7 sends +K28.5 with the receiver decoding this as either a C5.0 or a C2.7 if its current RD was negative. Lastly, C4.7 sends a deliberate Running Disparity violation pattern. All of these codes simplify controller design as well as assist with in-system testing. TAXI−275 does not have the ability to send any of these codes.

Reframing

In a serial transmission system, the receiving device must have a method of determining byte boundaries. In many systems a unique special character is used for this purpose. When the byte framer is active, the receiver looks for (frames on) K28.5 SYNC characters present in the data stream. This character must be unique, such that any valid combination of other bits within the transmission stream will not erroneously create this synchronization (SYNC) symbol. Transmission line errors may cause some of

the bits within the information stream to become changed in such a way that the bits produce an erroneous (alias) SYNC. If the receiver has single-byte framing, this will cause the receiver to become misaligned, with all subsequent data being decoded incorrectly.

Both the HOTLink and TAXI−275 devices have the capability for double byte framing. Both devices Reframe on two occurrences of the Special Character K28.5 separated by 0, 1, 2, or 3 words (0, 10, 20, or 30 bits) as shown in *Figure 9*.

The HOTLink RF pin is used to activate and deactivate the reframing option. This is useful in systems that wish to prevent byte misalignment from alias SYNCs during data packets. Byte misalignment will cause all subsequent data in a packet to be corrupted instead of just the word or words that were corrupted due to transmission errors. Single-byte reframing is active for the first 2K bytes after the RF pin is asserted HIGH. This feature allows the receiver to SYNC to the first K28.5. After 2K bytes during RF HIGH, double-byte reframing will be activated. When activated, the single-byte frame saves 10 mA.

HOTLink pads the spaces between data packets with SYNC characters. When the "No Enable" ($\overline{ENN}$ and $\overline{ENA}$ = HIGH) condition exists, the transmitter fills the unused bandwidth with K28.5s. This pad string should be identified at the receiver so that the receiving system is not forced to process this information.

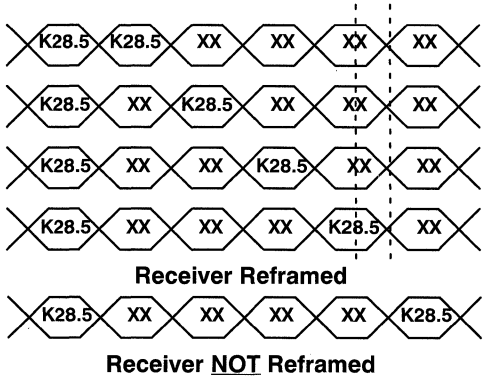


Figure 9. Double-Byte Reframing

TAXI−275 has no method of ignoring multiple SYNC characters and preventing them from being passed to the receiving system. The TAXI−275 STRBO pin pulses LOW in the presence of new Command or Data at the output register. It pulses LOW, therefore, every time a K28.5 character is received. If multiple SYNCs are passed to the outputs of the receiver, the receive FIFO will overflow with SYNC characters, which will require external decoder logic to discard this extraneous information.

HOTLink eliminates this problem by only pulsing the $\overline{RDY}$ pin LOW during the last SYNC character in a string of SYNC characters (the first SYNC character of a new packet of information). This is important in systems that have bursty data transmission or transmit data slower than the maximum data operating frequency. This prevents redundant information from being passed to the receive system, yet maintains packet boundaries for easy packet identification.

Higher Operating Frequency

HOTLink has a much broader frequency range than the TAXI−275 devices. TAXI−275 operates from 175 to 275 MBaud. This means that in 8B/10B mode, TAXI−275 can transmit and receive parallel data at rates from 17.5 to 27.5 MBytes/s. HOTLink, on the other hand, can transmit and receive parallel data at rates from 16 to 33 MBytes/s, allowing a much wider possible range of operating frequencies.

BIST

BIST (Built-In Self-Test) can be used to test the transmitter, receiver, and the link connecting them. During BIST (See *Figure 10*), the transmitter repeats a pattern representing all possible data and command characters, decodes them into transmission symbols and passes them to its outputs. The receiver, while in BIST, waits for the symbol that represents the beginning of the BIST pattern. It then decodes this and every following symbol and compares them with an internally generated pattern created by a pattern generator that matches the transmitter pattern generator. Detected errors at the receiver are indicated with pulses on the RVS (Received Violation Symbol) while completed BIST loops are indicated with pulses on the receiver $\overline{RDY}$ line. The BIST function checks the entire

function of the transmitter (except the transmitter input pins and the bypass function in the Encoder), the serial link, and the receiver.

These Built-In Self-Test functions are not implemented in the TAXI−275 devices. A substantial amount of additional circuitry is required in a system in order to integrate this function. This type of testing is necessary for many types of in-system diagnostic testing, including device functionality and link integrity.

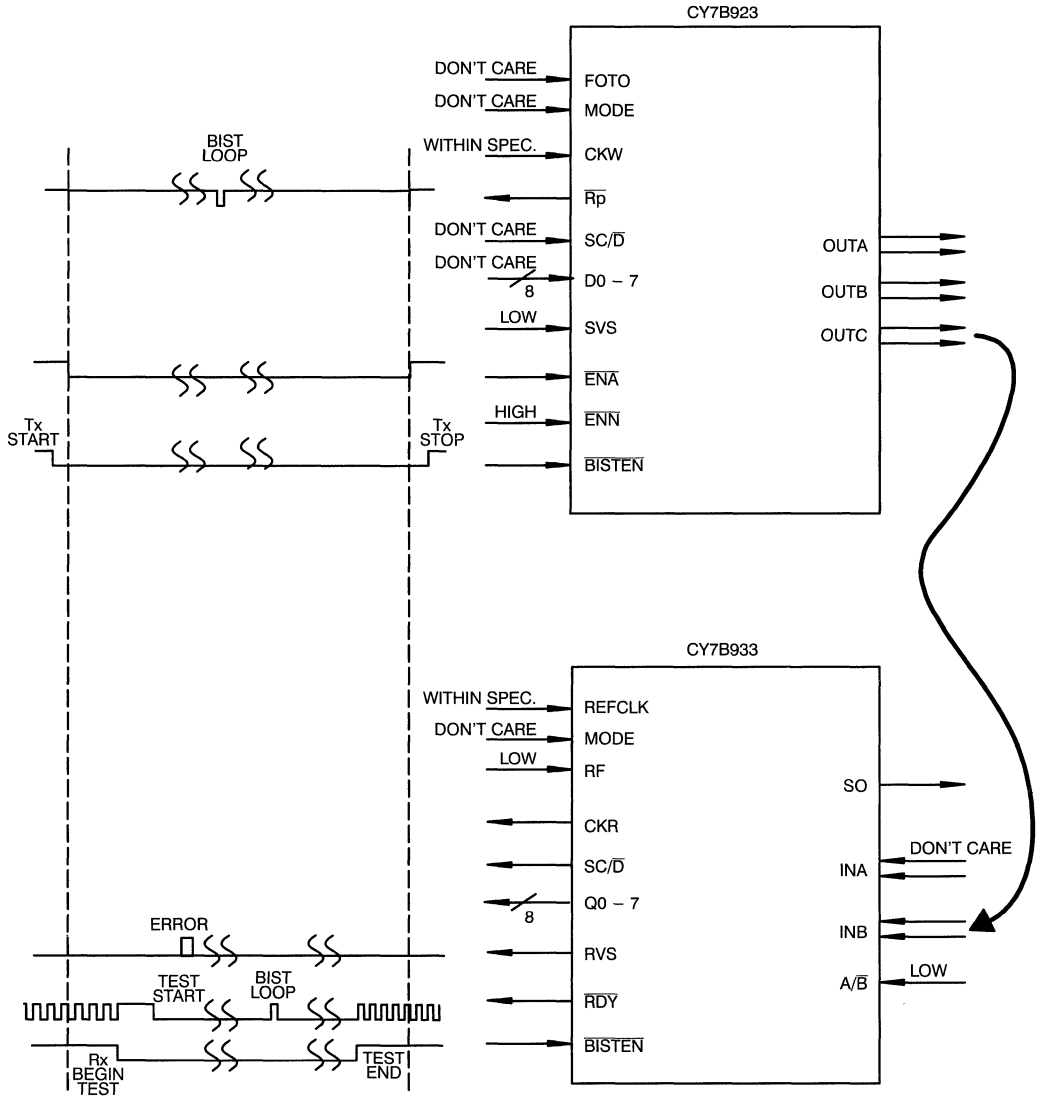


Figure 10. Built-In Self-Test

Parallel Interface

The TAXI−275 devices have two methods of strobing data into the device, synchronous and asynchronous. In the asynchronous mode of operation, a strobe line is used in conjunction with an acknowledge line to present data to the device. In this mode of operation the maximum operating frequency for the TAXI−275 devices under the most ideal of conditions is no faster than 20 MHz.

In the synchronous mode of operation, which is the most common method of device operation, the TAXI−275 device requires that the STRBI (Input Strobe) and the CLKI (Input Clock) be tied together. To enable or disable data in this mode requires external logic with slower than optimal (<275 Mbaud) operation. HOTLink has a very simple interface that allows seamless connection to both asynchronous and clocked FIFOs. On the transmitter, two enable inputs control when data is to be transmitted. When the $\overline{\text{ENA}}$ input is asserted, data on the data lines is serialized and transmitted. When the $\overline{\text{ENN}}$ line is asserted, data that is presented on the data lines during the next rising edge of the CLK input is transmitted. This allows efficient, synchronous state machines to control the flow of data over the serial link. In addition, the $\overline{\text{RP}}$ (read pulse) output can be connected to the $\overline{\text{R}}$ (read) input of asynchronous FIFOs, as shown in *Figure 11*, to provide a seamless asynchronous interface. The $\overline{\text{RP}}$ signal has timing that matches the timing required by asynchronous FIFOs. For clocked FIFO designs like that shown in *Figure 12*, the $\overline{\text{ENN}}$ input is used to not only read data from a Clocked FIFO like the Cypress CY7C443, but also to latch data into the Transmitter on the next rising edge of CKW.

The receiver has a $\overline{\text{RDY}}$ output that pulses LOW each time new data has been received. The $\overline{\text{RDY}}$ output has timing that allows the receiver to be seamlessly interfaced with both asynchronous and clocked FIFOs as shown in *Figures 11* and *12*. The TAXI−275 devices require a significant amount of additional circuitry to allow interfacing with FIFOs.

DC Specifications

The maximum current specification of the TAXI−275 Transmitter operating at 27.5 MB/s is 255 mA. The maximum current specification of the HOTLink Transmitter at 33 MB/s is only 80mA.

The TAXI−275 Receiver requires a maximum of 390 mA to operate at 27.5 MB/s whereas the HOTLink Receiver requires only 150 mA when operating at 33 MB/s.

Additionally, the TAXI−275 devices require 100 mV of differential input voltage at the receiver to accurately recover the clock and data from the input serial data stream. The HOTLink Receiver requires only 50 mV of differential input voltage. This translates into lower error rates, increased noise margins, higher jitter tolerance, and longer transmission distances when compared with the TAXI−275 devices.

Sending Violations

In many systems it is important to explicitly send violations. In normal system operation, a violation can be caused by either a received symbol having no corresponding decode value in the receiver, or a valid code received with the wrong Running Disparity. It is useful to send violation codes for testing, signaling, and interrupting the receiving system. The TAXI−275 devices have no method of code rule or Running Disparity violations. The HOTLink Transmitter, on the other hand, can send a pattern that will translate into a Code Rule Violation (C0.7) or Running Disparity Violation (C4.7) at the receiver. These Violations are indicated with a HIGH state on the RVS output with a Code Rule Violation indicated with command code C0.7 and a Running Disparity Violation indicated with command code C4.7. In addition, the SVS pin can be used to send a Code Rule Violation with the same indication at the Receiver.

ECL-to-TTL Translator

The TAXI−275 device does not include an ECL-to-TTL translator. The HOTLink Receiver has a built-in ECL-to-TTL translator where the SI input takes the single-ended ECL 100K (+5V referenced) signal in and the translated TTL signal is presented at

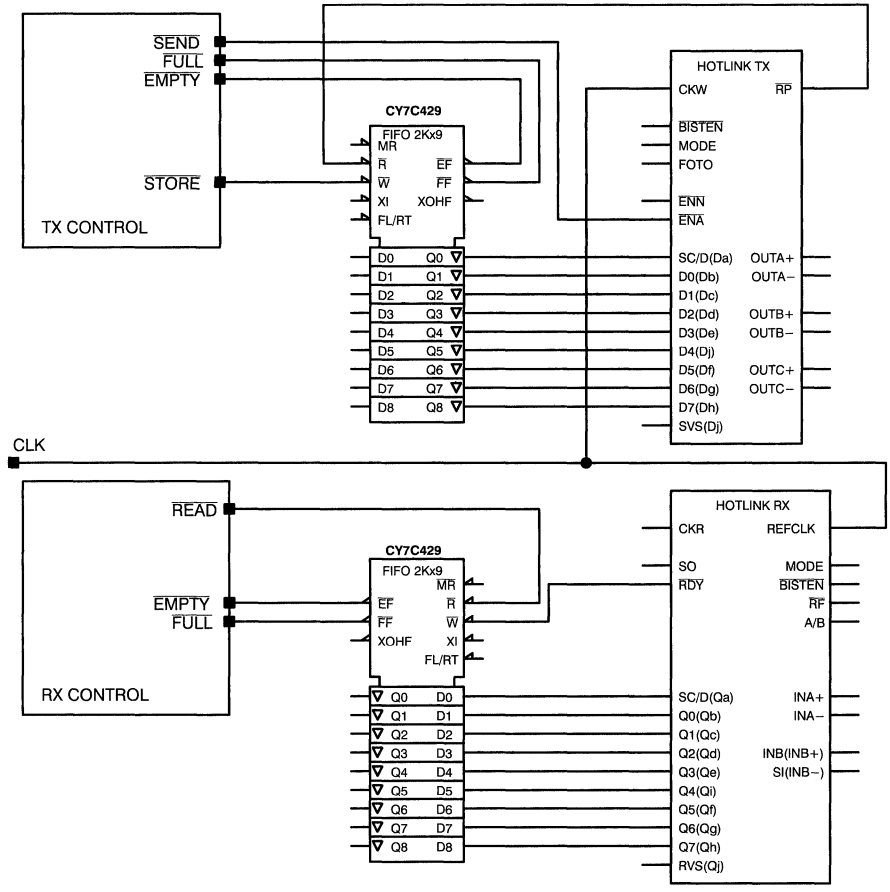


Figure 11. Asynchronous FIFO Interface

the SO output. The system can utilize this translator to convert an ECL carrier-detect signal from an optical module into its TTL equivalent for use by a controller.

Output Enable Considerations

The TAXI−275 devices use the $\overline{OE1}$ and $\overline{OE2}$ inputs to force the TX and TY outputs to their logic 0 state. A HIGH on $\overline{OE1}$ and a LOW on $\overline{OE2}$ will force TX LOW and TY HIGH. The analogous function on HOTLink is implemented with the FOTO (Fiberoptic Transmitter Off) pin. When the FOTO pin is held HIGH the OUTA+ and OUTB+ are forced LOW and the OUTA− and OUTB− outputs are forced HIGH. This causes a fiberoptic transmit module to extinguish its light output. The OUTC outputs are unaffected by the FOTO pin so that loop-back testing can be performed while the other outputs are turned off.

When the TAXI−275 $\overline{OE1}$ and $\overline{OE2}$ are both pulled HIGH, the TX and TY output drivers are turned off. This same result can be accomplished on HOTLink by either pulling both of the outputs of an output pair HIGH or simply leaving them unconnected. This will turn both outputs of an output pair off and save approximately 5 mA per output pair.

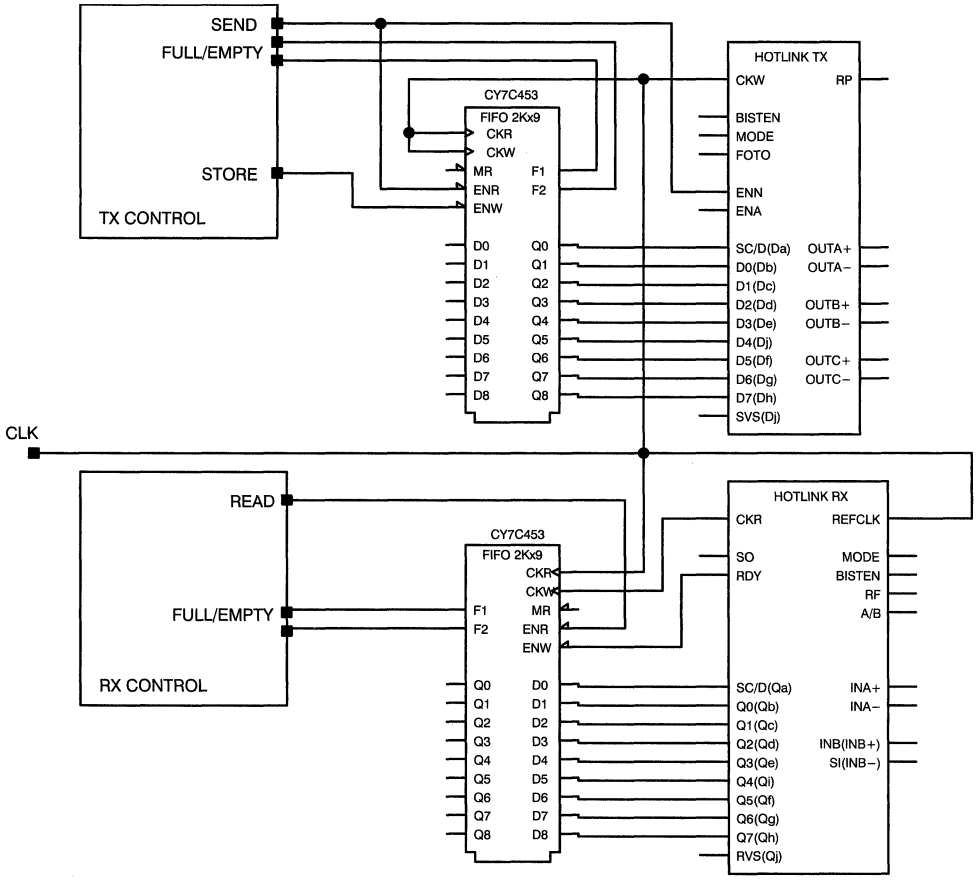


Figure 12. Clocked FIFO Interface

Status Indication

The TAXI−275 S1 and S2 status pins are used to indicate the status of the parallel output data as shown in *Table 2*.

Table 2. TAXI−275 Status Indication

Pin Status		Indication
S1	S2	
0	0	Data
1	0	Command
0	1	Violation
1	1	Re-Align

The SC/$\overline{D}$ (Special Character/$\overline{DATA}$), $\overline{RDY}$ ($\overline{Ready}$), and RVS (Receive Violation Symbol) outputs of HOTLink provide more status information than that provided by the TAXI−275 status pins as shown in *Table 3*. This table shows that Data and Command signalling on the HOTLink and the TAXI−275 devices are very similar. Violations, however, are indicated very differently between the two devices.

Table 3. HOTLink Status Indication

Function	SC/$\overline{D}$	$\overline{RDY}$	RVS	Q0−7	TAXI Indication
Data	0	0	0	Data	Data
Command	1	0	0	Command	Command
Code Rule Violation	1	0	1	C0.7	S1/S2=01
Running Disparity Violation	1	0	1	C4.7	S1/S2=01
Sync indication after reframe	1	0	1	C5.0	NONE

A Code Rule Violation is indicated with the SC/$\overline{D}$ pin HIGH, a LOW pulse on the $\overline{RDY}$ line, a HIGH on the RVS pin, and C0.7 on the data lines. A Code Rule Violation is a 10-bit transmission character that can not be decoded into an 8-bit symbol. Coding Violations are caused by errors during transmission across the link. A Running Disparity Violation is indicated in the same manner on the SC/$\overline{D}$, $\overline{RDY}$, and RVS pins as a Code Rule Violation, but the data output lines indicate the C4.7 command. A Running Disparity Violation is present when a transmission character is able to be decoded into an 8-bit symbol, but the transmission character had the wrong Running Disparity.

It is important that these two different types of violations are indicated separately to a controller. A Code Rule Violation indicates that the current symbol is corrupted. In this situation the controller would most probably throw away the erroneous word. A Running Disparity Violation, on the other hand, indicates that the current word probably is correct, but that at some point in the past the data became corrupted. In this situation, the controller would probably discard the entire packet.

Additionally, HOTLink provides a SYNC indication after entering reframing. When RF is brought high, the $\overline{RDY}$ line pulses low after the first SYNC character (K28.5) has been received. This feature assists the Reframe state machine in determining when the receiver has been reframed. The Reframe state machine could pull RF LOW after the SYNC indication. This would prevent alias SYNC characters from realigning the receiver. TAXI−275 devices do not have the ability to indicate when data has been framed to a K28.5.

The TAXI−275 devices only have an indication that the receiver was realigned (S1,S2=11). The status lines do not always indicate if the TAXI−275 Receiver has reframed when SYNCD is LOW. Only if the byte boundary has changed will the status pins change. A reframe controller, therefore, must monitor all of the command lines to determine if the receiver has correctly framed on the data stream. This complicates reframe state machine design.

Conclusion

HOTLink has many advantages when compared with the AMD Am79168 Transmitter and Am79169 Receiver (TAXI). These advantages include those listed below.

- Multiplexed command and data
- Three differential serial outputs
- Two differential serial inputs
- More flexible Command codes
- More flexible reframing
- Higher operating frequency
- Built-In Self-Test
- Simplified synchronous interface
- Reduced power consumption
- Ability to send violations
- Simplified output enable interface
- More complete receiver status indications
- ECL-to-TTL translator

These advantages of HOTLink provide greater system flexibility, simplified controller design, more reliable data communication, and lower power consumption.

References

1. Cypress Semiconductor, *CY7B923/CY7B933 HOTLink Transmitter/Receiver Preliminary Datasheet*, Cypress Semiconductor High Performance Data Book, August 1, 1993.
2. Cypress Semiconductor, *HOTLink Design Considerations Application Note*, October 1993.
3. Advanced Micro Devices, Am79168/Am79169−275 TAXI−275 Integrated Circuits Technical Manual, Rev. 1.0, 1993.
4. Advanced Micro Devices, Am79168/Am79169−275 TAXI−275 Transmitter/Receiver Transparent Asynchronous Transmitter/Receiver Interface Preliminary Data Sheet, March 1993 Rev B.

HOTLink™ Built-In Self-Test (BIST)

The Cypress CY7B923 and CY7B933 HOTLink™ Transmitter and Receiver offer the system integrator a tool to send data from place to place over a high-speed serial transmission link. They have been designed for easy integration into any data communication subsystem and have a data-interface equivalent to that of a data register or FIFO memory. *Figure 1* shows schematically where HOTLink might fit into a typical system. In this example, data is being sent from one data bus to another over a serial link. The controllers used at each end of this representative data link show the generic functions of all data-link controllers. The protocol controller might be simply a PLD state machine, or a VLSI protocol-specific subsystem. The Bus Interface might be an octal register, or a standard CPU bus controller. The data buffer might be a register in the bus interface or a packet-sized FIFO memory. HOTLink connects these typical logical building-blocks with a high-performance serial data link.

Serial data links are notoriously difficult to design and test. HOTLink simplifies the design problem, and offers the system integrator and end user a simple method to test the finished link. The Built-In Self-Test feature described in this application note

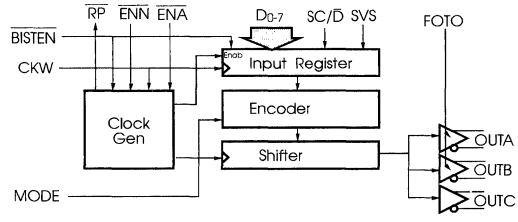


Figure 2. HOTLink Transmitter Block Diagram

gives an unambiguous, real-time, offline test of the entire link. Several ways to test serial links, and serial link components are described using BIST as the grading tool.

The Cypress CY7B923 and CY7B933 HOTLink Transmitter and Receiver block diagrams in *Figures 2* and *3* show the built-in functionality included in these parts. The transmitter uses a fully integrated PLL to multiply the user's byte-rate clock for use as a serial data rate clock. The receiver has a fully integrated PLL that tracks the serial data stream and recovers the bit-rate clock. The bit rate clock is used for internal data decoding and generating the parallel byte-rate clock aligned to the recovered data.

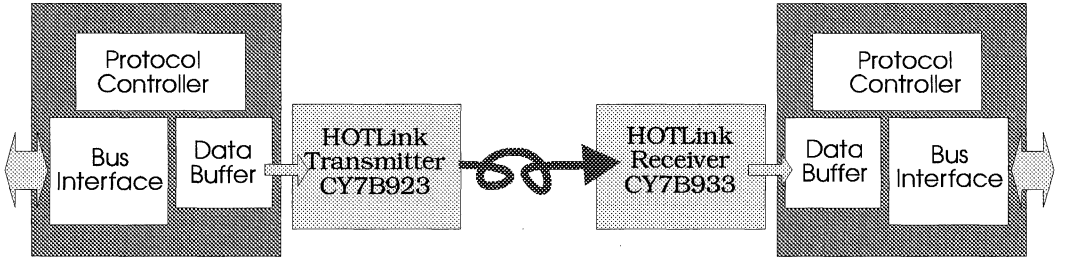


Figure 1. Typical Data-Communication Link using HOTLink

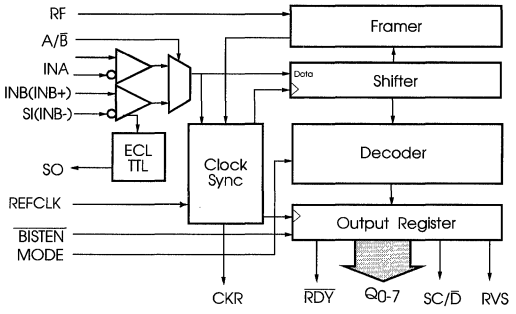


Figure 3. HOTLink Receiver Block Diagram

Both parts have built-in 8B/10B code converters that conform to ANSI X3T11 Fibre Channel and ESCON™ specifications. The encoder and decoder transform 8-bit user data to 10-bit characters more suitable for transmission systems built with fiber-optic cables or any common wire transmission line. Both parts have parallel I/O registers designed to interface with standard FIFO memories without any extra logic. These parts also offer a full-function Built-In Self-Test feature that tests all the circuitry in both ICs and all of the interconnect components that make up the link.

In most traditional serial interface links, the testing of the serial link components and interconnections was done off-line using high-speed analog test equipment and specialized parameter analyzers. If any in-line testing was done, it was done as part of some particular communication protocol or higher-level software functionality. In-line testing usually consisted of only a simple loop-back handshake test that often failed to isolate the problem (if any) to any specific component or connection. And, these simple tests were so superficial that they typically failed to indicate ANY problem at all.

HOTLink products offer the capability to do in-line testing of the entire serial link without adding or removing components from an operational system. The overhead associated with this capability is minimal, and doesn't affect any of the critical analog or high speed interfaces.

Figure 4 shows a link interconnect that is typical for HOTLink systems and illustrates the interface signals used for the BIST operation. This configuration is identical to that of a normally operating data transmission link, except for the simple control features that have been added to facilitate the test function. While the form of the transmission link shown in *Figure 4* illustrates a test being run across an operational communication link, similar testing can be done in either Local-Loop-Back or Long-Loop-Back configurations as shown in *Figures 5* and *6*.

Since there is no required feedback connection between the receiver and transmitter, the BIST function can be used in any configuration that the system designer might find desirable. The previous example (*Figure 4*) illustrates a cooperative test between the two ends of a point-to-point transmission system. The same function can be accomplished as part of the loop-back test of a single communicating node. *Figure 5* shows a typical Local Loop-Back connection wherein the test is run without any involvement of the remote station.

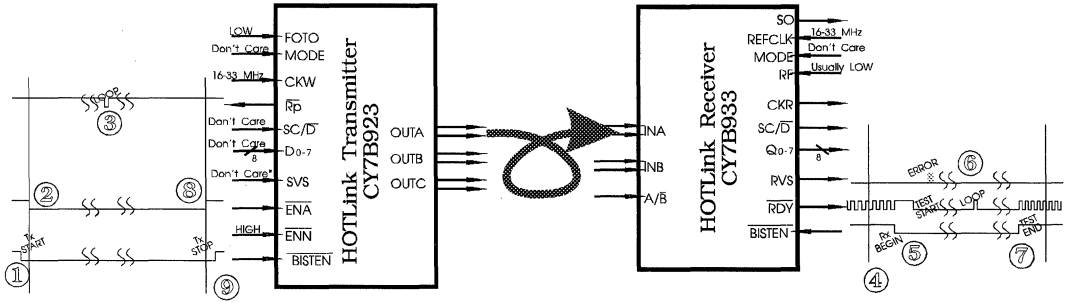


Figure 4. HOTLink BIST Connections

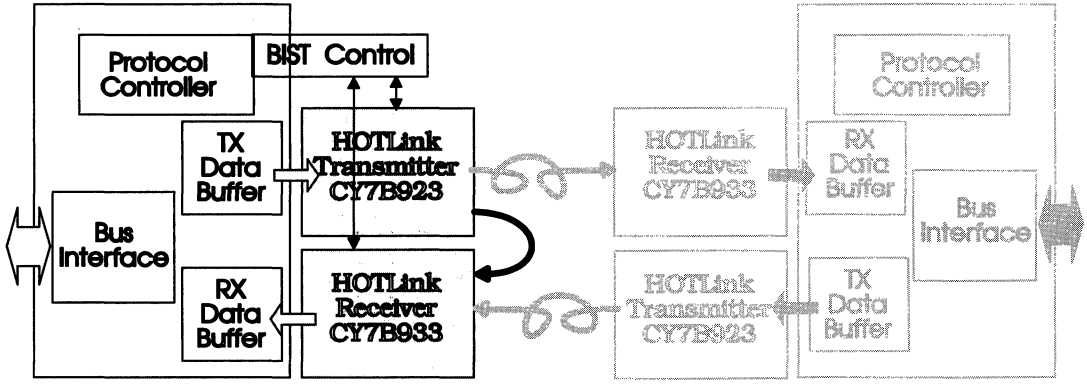


Figure 5. Local Loop Back Testing

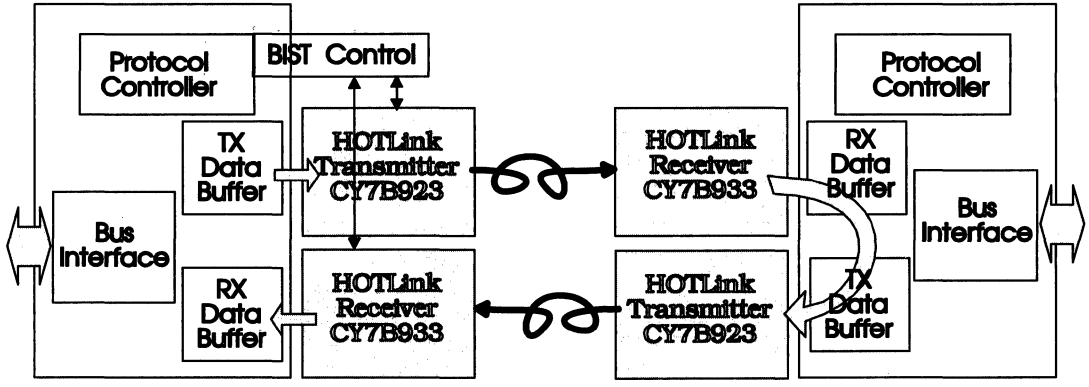


Figure 6. Long Loop Back Testing

A more complete test can be made if it is possible to gain the cooperation of the station at the remote end of the communication link. In the example shown in *Figure 6*, the remote station becomes a simple flow-through repeater. This connection is typical of many network systems, and can be implemented in many simple and obvious ways. This test takes advantage of the orthogonality built into HOTLink products. The data patterns that emerge from the parallel data outputs of the receiver will send equivalent codes when put into the parallel data inputs of the transmitter. This is true in either Bypass Mode or Encoded Mode. The only restriction is that the receiver and transmitter, which are connected via the parallel data path, must both be in the same mode. In the illustration shown in *Figure 6*, the pair of HOTLink chips on the left should be in the same mode, and the pair on the right should be in the same mode. Each pair could be in either Bypass or Encoded mode.

These BIST specific features do not affect the normal message transaction function of the HOTLink's host system. *Figure 7* shows the logical functions that would be added to the protocol controller to support BIST. The timer shown in both the transmitter and receiver BIST control functions might be an analog timer (one-shot) or digital logic that counts BIST-Loops. It could also be simply a user-controlled time-out that performs BIST while a TEST button is pressed.

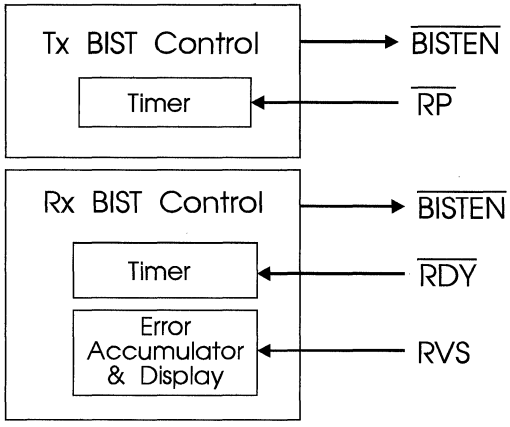


Figure 7. Control Functions for Built-In Self-Test

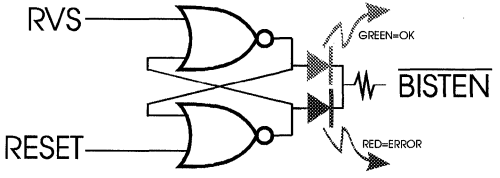


Figure 8. A Simple BIST Error Indicator

The error accumulator and display shown in the example could be as simple or as complex as desired. For a software supported BIST function, the error-count accumulator would be machine readable. For example, the CY9266 HOTLink Evaluation Board provides a two-digit counter that displays the accumulated error count, but in general any indicator will work. *Figure 8* shows a simple RS latch connected to trap errors and display any rare and random RVS indication.

The simple circuit shown above provides an unambiguous indication of errors detected by the BIST comparator and indicated by the RVS output of the HOTLink Receiver. If the $\overline{\text{BISTEN}}$ control is LOW, the LED output indicates the state of the R-S latch that records any pulses on RVS. After $\overline{\text{BISTEN}}$ is set LOW, and the circuit is RESET, the GREEN-LED is illuminated indicating no detected errors. If RVS ever pulses HIGH, then the RED-LED will light and stay illuminated until RESET is pulsed HIGH (probably by a momentary push-button switch or a control signal from the controller). When $\overline{\text{BISTEN}}$ is set HIGH, the LEDs will be extinguished.

The four areas that must be addressed to build BIST functionality into a system involve control of $\overline{\text{BISTEN}}$ pins, selection of transmitter enable pins ($\overline{\text{ENA}}$ or $\overline{\text{ENN}}$), test-progress supervision and error reporting.

The $\overline{\text{BISTEN}}$ control pins enable only the BIST function logic and can be externally controlled without affecting any system resources or timing. The BIST function performs its test sequence automatically, regardless of the encoding mode selected for the operational data link. No consideration needs to be made for this aspect of a HOTLink system.

The HOTLink Transmitter includes two separate enable inputs. The $\overline{\text{ENA}}$ input is used for systems that simultaneously present data and its validation signal (i.e., traditional controllers, and asynchronous FIFOs). The $\overline{\text{ENN}}$ input is used for systems that have a request function and expect the data to be presented on the next clock cycle (i.e., typical of read-request controllers and clocked FIFOs). The system integrator selects the appropriate enable pin to accommodate system functionality and timing. The other enable can be used to control BIST functionality, thus reducing the burden of additional control logic. When $\overline{\text{BISTEN}}$ is enabled (LOW) and one of the enables (either $\overline{\text{ENA}}$ or $\overline{\text{ENN}}$) is asserted (LOW) the HOTLink Transmitter creates a continuous 511 byte (2^9-1 bytes) pseudo-random stream of 8B/10B-encoded data-patterns, which the HOTLink Receiver checks byte-by-byte. The 256 data patterns are sent once each and the 12 special characters and the 4 error codes are sent sixteen times each (except C0.0 which is sent only 15 times) for a total of another 255 data patterns. For a complete list of codes used in the 8B/10B encoder and the special character and error codes, see the CY7B923/933 HOTLink Transmitter/Receiver datasheet.

The system control logic can monitor the test duration at the transmitter by observing the $\overline{RP}$ pin output. This pin is normally used as the byte-rate read-pulse for asynchronous FIFOs, but during the BIST function $\overline{RP}$ will pulse once per loop. By using a counter or timer that observes $\overline{RP}$ while $\overline{BISTEN}$ is LOW, the number of transmitter test-loops can be monitored. (See timing illustration in *Figure 4*, times 1, 2, 3, 8, and 9.)

The $\overline{RDY}$ output of the receiver serves a similar function as $\overline{RP}$ in the transmitter. In normal modes this output is a status indicator used to control the data flow to an external controller or FIFO. $\overline{RDY}$ pulses once per byte in Encoded Mode, and once per SYNC in Bypass Mode. In BIST Mode, it will rest HIGH while the receiver awaits the start of the BIST sequence, and then rest LOW for all but one byte-time of the BIST loop. A counter or timer similar to that described for the transmitter can be used to count when $\overline{RDY}$ pulses HIGH, recording the number of times that the receiver has executed the BIST loop. (See timing illustration in *Figure 4*, times 4, 5, 6 and 7.)

If errors are discovered in the received sequence, received running disparity or received transmission codes, they are flagged by the RVS output. (See timing illustration in *Figure 4*, time 6.) In Encoded Mode this output is not used as part of the communication data path. In Bypass Mode this output serves as the Qj output, but in either case the external BIST control engine will use RVS as an input with a minimal impact on the rest of the operating system.

Transmitter BIST Generator

The BIST generator in the transmitter is the Input register reconfigured into a nine-bit Linear Feedback Shift register (LFSR). In this configuration it makes every possible combination of nine bits (minus 1). The 256 data codes are sent once each, and the 12 special characters and 4 error codes shown in the datasheet are sent multiple times to complete the 511-byte pattern. This is the pattern that is sent regardless of the HOTLink Transmitter and Receiver encoding mode being used by the system. If the system is using the Bypass Mode (MODE input = V_{CC}) the Tx-Encoder and the Rx-Decoder is enabled upon the beginning of BIST Mode ($\overline{BISTEN}$ = LOW and $\overline{ENA}$ or $\overline{ENN}$ = LOW). When BIST Mode ends, the encoder and decoder are bypassed (if MODE = V_{CC}) and user data is directly serialized and deserialized.

Table 1 is a complete list of the codes sent by the transmitter when it has been enabled (either $\overline{ENA}$ or $\overline{ENN}$ held LOW) in BIST Mode ($\overline{BISTEN}$ input LOW). This sequence is a continuously repeating data loop that may start at any point and then continue through all of the codes. There are several places in the sequence where running disparity is explicitly forced, and as a result, each pass through the sequence will repeat exactly. (It is possible to precondition the BIST generator so that, on the first time through, the initial few codes will be sent using the wrong running disparity.) Running disparity will be corrected when one of the force codes (i.e., C1.7 or C2.7) is encountered and will certainly be correct before the next "Start of BIST".

Once per loop (synchronous with the D0.0 at the beginning of the table), $\overline{RP}$ will pulse (see *Figure 4*, time 3). This BIST specific behavior allows an external system monitor state machine to count the number of times that the transmitter has sent its BIST data. In normal non-BIST operation, $\overline{RP}$ pulses once per byte when $\overline{ENA}$ is asserted, and doesn't pulse at all otherwise.

Table 1 shows the transmitter BIST sequence expressed in codes that would be commonly sent by a controller connected to the transmitter input pins. This table would be used if a controller was programmed to send the BIST pattern as an external system function or as a test of the BIST pattern checker in the receiver.

In normal operation it is unnecessary for the user to initialize the transmitter BIST sequence since the receiver automatically aligns its code generator to the pattern being sent by the transmitter. If there is some ambiguity in the exact start of the transmitter BIST loop, the receiver will patiently wait for a good start and begin checking from there. If an erroneous code causes the receiver to make a false start, it will soon detect the error and resume waiting for the proper starting code.

Table 1. HOTLink Transmitter BIST Sequence

Start here ---->															
D0.0	C0.0	C4.0	C1.7	C4.7	C0.7	D27.7	D23.3-->	D29.1	D30.0	C0.7	D15.4	D11.2	D3.1	D17.0	D4.0
C8.0	C6.0	C2.7	D14.7	C11.0	D19.5	D21.2	D12.1	C10.0	C3.0	D5.6	D8.3	C2.0	C1.0	D0.6	C0.0
C4.0	C8.0	C2.0	C5.0	D24.7	C6.0	C9.0	D18.6	C5.0	D28.5	C4.7	C11.0	D23.6	D13.3	D26.1	C7.0
D9.4	D2.2	C1.0	D20.4	C1.7	C4.7	C0.7	D11.7	D19.3	D21.1	D28.0	C4.7	C0.7	D15.6	D11.3	D19.1
D21.0	D12.0	C10.0	C7.0	D13.6	D10.3	C3.0	D17.4	D4.2	C8.0	C6.0	C9.0	D6.7	C9.0	D18.5	C5.0
D8.5	C2.0	C1.0	D20.6	C1.7	C4.7	C11.0	D3.7	D17.3	D20.1	C1.7	C10.0	C7.0	D13.7	D26.3	C7.0
D25.4	D6.2	C9.0	D22.4	C2.7	D14.5	C11.0	D3.5	D17.2	D4.1	C8.0	C2.0	C5.0	D12.7	C10.0	C3.0
D17.6	D4.3	C8.0	C2.0	C1.0	D4.7	C8.0	C2.0	C1.0	D0.7	C0.0	C0.0	C0.0	C0.0	C4.0	C8.0
C2.0	***C1.0***	D16.7	C4.0	C8.0	C2.0	C1.0	D20.7	C1.7	C10.0	C3.0	D1.7	D16.3	C4.0	C8.0	C2.0
C5.0	D28.7	C4.7	C11.0	D19.6	D5.3	D24.1	C6.0	C9.0	D6.6	C9.0	D22.5	C2.7	D10.5	C3.0	D1.5
D16.2	C4.0	C1.7	C10.0	C7.0	D29.7	D30.3	C0.7	D27.4	D7.2	D9.1	D18.0	C5.0	D12.4	C10.0	C7.0
D29.6	D14.3	C11.0	D19.4	D5.2	D8.1	C2.0	C1.0	D4.6	C8.0	C6.0	C9.0	D2.7	C1.0	D16.5	C4.0
C8.0	C6.0	C9.0	D22.7	C2.7	D26.5	C7.0	D9.5	D18.2	C5.0	D28.4	C4.7	C0.7	D31.6	D15.3	D27.1
D23.0	D13.0	D10.0	C3.0	D5.4	D8.2	C2.0	C5.0	D8.6	C2.0	C5.0	D24.6	C6.0	C2.7	D26.6	C7.0
D29.5	D30.2	C0.7	D31.4	D15.2	D11.1	D19.0	D5.0	D8.0	C2.0	C5.0	D12.6	C10.0	C7.0	D25.6	D6.3
C9.0	D18.4	C5.0	D12.5	C10.0	C3.0	D21.6	D12.3	C10.0	C3.0	D1.6	D0.3	C0.0	C0.0	C0.0	C4.0
C1.7	C10.0	C3.0	D17.7	D20.3	C1.7	C10.0	C3.0	D5.7	D24.3	C6.0	C9.0	D2.6	C1.0	D20.5	C1.7
C10.0	C7.0	D9.7	D18.3	C5.0	D24.4	C6.0	C2.7	D30.6	C0.7	D31.5	D31.2	D15.1	D27.0	D7.0	D9.0
D2.0	C1.0	D4.4	C8.0	C6.0	C2.7	D10.7	C3.0	D17.5	D20.2	C1.7	C4.7	C11.0	D7.7	D25.3	D22.1
C2.7	D10.4	C3.0	D5.5	D24.2	C6.0	C2.7	D10.6	C3.0	D21.5	D28.2	C4.7	C0.7	D11.6	D3.3	D17.1
D20.0	C1.7	C4.7	C0.7	D15.7	D27.3	D23.1	D29.0	D14.0	C11.0	D7.4	D9.2	D2.1	C1.0	D0.4	C0.0
C4.0	C1.7	C10.0	C7.0	D25.7	D22.3	C2.7	D26.4	C7.0	D13.5	D26.2	C7.0	D29.4	D14.2	C11.0	D23.4
D13.2	D10.1	C3.0	D1.4	D0.2	C0.0	C4.0	C8.0	C6.0	C2.7	D26.7	C7.0	D25.5	D22.2	C2.7	D30.4
C0.7	D15.5	D27.2	D7.1	D25.0	D6.0	C9.0	D6.4	C9.0	D6.5	C9.0	D2.5	C1.0	D0.5	C0.0	C0.0
C4.0	C8.0	C6.0	C9.0	D18.7	C5.0	D24.5	C6.0	C9.0	D22.6	C2.7	D30.5	C0.7	D11.5	D19.2	D5.1
D24.0	C6.0	C2.7	D14.6	C11.0	D23.5	D29.2	D14.1	C11.0	D3.4	D1.2	D0.1	C0.0	C0.0	C4.0	C1.7
C4.7	C11.0	D19.7	D21.3	D28.1	C4.7	C11.0	D7.6	D9.3	D18.1	C5.0	D8.4	C2.0	C5.0	D28.6	C4.7
C0.7	D27.6	D7.3	D25.1	D22.0	C2.7	D14.4	C11.0	D7.5	D25.2	D6.1	C9.0	D2.4	C1.0	D4.5	C8.0
C2.0	C5.0	D8.7	C2.0	C1.0	D16.6	C4.0	C1.7	C10.0	C3.0	D21.7	D28.3	C4.7	C11.0	D3.6	D1.3
D16.1	C4.0	C8.0	C6.0	C2.7	D30.7	C0.7	D27.5	D23.2	D13.1	D26.0	C7.0	D13.4	D10.2	C3.0	D21.4
D12.2	C10.0	C7.0	D9.6	D2.3	C1.0	D16.4	C4.0	C1.7	C4.7	C11.0	D23.7	D29.3	D30.1	C0.7	D11.4
D3.2	D1.1	D16.0	C4.0	C1.7	C4.7	C0.7	D31.7	D31.3	D31.1	D31.0	D15.0	D11.0	D3.0	D1.0	GOTO Start

If the user wants to initialize the BIST pattern, there are two methods available. In the Encoded Mode (MODE input=GND) the SVS input overrides the BIST sequence and forces the transmitter to send the code indicating a Code Rule Violation (see the CY7B923/933 HOTLink datasheet for a complete list of 8B/10B data codes and special characters). It also resets the BIST LFSR to its initial state (D0.0). (Running disparity is not explicitly set by SVS, and the first few bytes after its release may be sent with the wrong disparity. The fourth byte of the sequence, C1.7, will explicitly set running disparity for the rest of the patterns.)

Alternatively, the transmitter BIST LFSR can be forced to start its pattern from any other point in the sequence by noting that the BIST sequence proceeds from the code that was in the Input register when BIST was asserted. (Note that the BIST sequence generator state sequence is expressed in En-

coded-Mode terms. If the transmitter is in Bypass Mode, the Next State can be deciphered by converting the actual bit pattern on the inputs to the code that the transmitter would send if it were in the Encoded Mode and that were its input pattern, and then looking for that code in the table.) In most cases this will be sufficient to initialize the sequence, except for the state of the internal running disparity flip-flop. If running disparity must also be assured, the codes for +K28.5 (C2.7) or −K28.5 (C1.7) should be used to initialize the LFSR. The C2.7 and C1.7 starting locations are shown in *Table 1* (row 9, items 1 and 2 respectively). The user would put, for example, C2.7 on the transmitter inputs for one or more byte times, then start the BIST test. The pattern would start from row 9, item 1 in this example. This technique can be useful if the user wants only a portion of the transmitter BIST pattern for some oscilloscope test.

Receiver BIST Comparator

The BIST generator in the receiver is the Output Register reconfigured into a nine-bit Linear-Feedback-Shift-Register (LFSR) that exactly matches the one in the transmitter. In this configuration it puts all possible combinations of nine bits on the receiver outputs (Q_{0-7} and SC/$\overline{D}$). *Table 2* is a complete list of the codes that must appear on the receiver serial inputs during a BIST test-loop if the receiver is to indicate "no-errors". These codes are slightly different than those shown in *Table 1* (e.g., the fourth element in *Table 1* = C1.7 = K28.5 with forced negative running disparity, the fourth element in *Table 2*, = C5.0 = correct K28.5) because of the way that the running disparity affects the interpretation made by the receiver when it decodes certain characters. This table shows the codes that would be sent by a controller sending the BIST sequence without depending upon the LFSR in the transmitter, or by a transmitter connected to an Encoded Mode receiver that was receiving a BIST sequence while not itself in BIST mode.

It should be noted that the first K28.5 (C5.0) (the fourth byte in the BIST loop at *1 in *Table 2*) may cause a false RVS indication on the first pass through the BIST loop, because the transmitter sends it as a C1.7 (−K28.5) to force the state of running disparity (RD). Depending upon the actual RD state in the receiver as BIST starts, this forced RD might appear to the receiver as a running disparity error on the first time through the BIST loop. All subsequent loops are interpreted correctly.

The SVS character (C0.7) combined with the adjacent D11.5 (*2 in *Table 2*, row 25 bytes 13 and 14) encoded as (011000 0111 110100 1010) creates an alias K28.5 (001111 1010) which will cause an erroneous reframe if RF is HIGH for short periods of time (less than 2048 bytes). This alias sync can be used to check the system response to clock stretching, a topic that will be covered later. This error (normal single-byte reframe behavior) will not occur when Multi-Byte-Sync is enabled (i.e., RF = HIGH for more than 2048 byte times).

Note that there are several intentional code rule violations and incorrect running disparity transitions included in the receiver sequence. RVS (during BIST) will indicate when an error in the expected code has been detected; it does not indicate that an illegal code is present. *Figure 9* illustrates this behavior and shows the timing of RVS when the receiver detects an error in the sequence. RVS will pulse HIGH for the byte time following detection of a mismatch between the received-decoded pattern and the internally generated code.

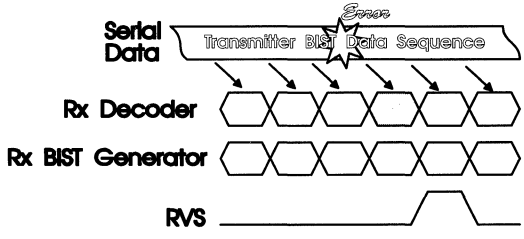


Figure 9. RVS Indicates Errors in Received Sequence

Table 2. HOTLink Receiver Input BIST Sequence

Start here ---->															
D0.0	C0.0	C4.0	**C5.0*1**	C4.7	C0.7	D27.7	D23.3 -->	D29.1	D30.0	C0.7	D15.4	D11.2	D3.1	D17.0	D4.0
C8.0	C6.0	C2.7	D14.7	C11.0	D19.5	D21.2	D12.1	C10.0	C3.0	D5.6	D8.3	C2.0	C1.0	D0.6	C0.0
C4.0	C8.0	C2.0	C5.0	D24.7	C6.0	C9.0	D18.6	C5.0	D28.5	C4.7	C11.0	D23.6	D13.3	D26.1	C7.0
D9.4	D2.2	C1.0	D20.4	C5.0	C4.7	C0.7	D11.7	D19.3	D21.1	D28.0	C4.7	C0.7	D15.6	D11.3	D19.1
D21.0	D12.0	C10.0	C7.0	D13.6	D10.3	C3.0	D17.4	D4.2	C8.0	C6.0	C9.0	D6.7	C9.0	D18.5	C5.0
D8.5	C2.0	C1.0	D20.6	C1.7	C4.7	C11.0	D3.7	D17.3	D20.1	C1.7	C10.0	C7.0	D13.7	D26.3	C7.0
D25.4	D6.2	C9.0	D22.4	C2.7	D14.5	C11.0	D3.5	D17.2	D4.1	C8.0	C2.0	C5.0	D12.7	C10.0	C3.0
D17.6	D4.3	C8.0	C2.0	C1.0	D4.7	C8.0	C2.0	C1.0	D0.7	C0.0	C0.0	C0.0	C0.0	C4.0	C8.0
C2.0	C1.0	D16.7	C4.0	C8.0	C2.0	C1.0	D20.7	C1.7	C10.0	C3.0	D1.7	D16.3	C4.0	C8.0	C2.0
C5.0	D28.7	C4.7	C11.0	D19.6	D5.3	D24.1	C6.0	C9.0	D6.6	C9.0	D22.5	C5.0	D10.5	C3.0	D1.5
D16.2	C4.0	C1.7	C10.0	C7.0	D29.7	D30.3	C0.7	D27.4	D7.2	D9.1	D18.0	C5.0	D12.4	C10.0	C7.0
D29.6	D14.3	C11.0	D19.4	D5.2	D8.1	C2.0	C1.0	D4.6	C8.0	C6.0	C9.0	D2.7	C1.0	D16.5	C4.0
C8.0	C6.0	C9.0	D22.7	C2.7	D26.5	C7.0	D9.5	D18.2	C5.0	D28.4	C4.7	C0.7	D31.6	D15.3	D27.1
D23.0	D13.0	D10.0	C3.0	D5.4	D8.2	C2.0	C5.0	D8.6	C2.0	C5.0	D24.6	C6.0	C2.7	D26.6	C7.0
D29.5	D30.2	C0.7	D31.4	D15.2	D11.1	D19.0	D5.0	D8.0	C2.0	C5.0	D12.6	C10.0	C7.0	D25.6	D6.3
C9.0	D18.4	C5.0	D12.5	C10.0	C3.0	D21.6	D12.3	C10.0	C3.0	D1.6	D0.3	C0.0	C0.0	C0.0	C4.0
C1.7	C10.0	C3.0	D17.7	D20.3	C1.7	C10.0	C3.0	D5.7	D24.3	C6.0	C9.0	D2.6	C1.0	D20.5	C1.7
C10.0	C7.0	D9.7	D18.3	C5.0	D24.4	C6.0	C2.7	D30.6	C0.7	D31.5	D31.2	D15.1	D27.0	D7.0	D9.0
D2.0	C1.0	D4.4	C8.0	C6.0	C2.7	D10.7	C3.0	D17.5	D20.2	C5.0	C4.7	C11.0	D7.7	D25.3	D22.1
C5.0	D10.4	C3.0	D5.5	D24.2	C6.0	C2.7	D10.6	C3.0	D21.5	D28.2	C4.7	C0.7	D11.6	D3.3	D17.1
D20.0	C1.7	C4.7	C0.7	D15.7	D27.3	D23.1	D29.0	D14.0	C11.0	D7.4	D9.2	D2.1	C1.0	D0.4	C0.0
C4.0	C5.0	C10.0	C7.0	D25.7	D22.3	C2.7	D26.4	C7.0	D13.5	D26.2	C7.0	D29.4	D14.2	C11.0	D23.4
D13.2	D10.1	C3.0	D1.4	D0.2	C0.0	C4.0	C8.0	C6.0	C2.7	D26.7	C7.0	D25.5	D22.2	C5.0	D30.4
C0.7	D15.5	D27.2	D7.1	D25.0	D6.0	C9.0	D6.4	C9.0	D6.5	C9.0	D2.5	C1.0	D0.5	C0.0	C0.0
C4.0	C8.0	C6.0	C9.0	D18.7	C5.0	D24.5	C6.0	C9.0	D22.6	C5.0	D30.5	**C0.7*2**	D11.5	D19.2	D5.1
D24.0	C6.0	C2.7	D14.6	C11.0	D23.5	D29.2	D14.1	C11.0	D3.4	D1.2	D0.1	C0.0	C0.0	C4.0	C1.7
C4.7	C11.0	D19.7	D21.3	D28.1	C4.7	C11.0	D7.6	D9.3	D18.1	C5.0	D8.4	C2.0	C5.0	D28.6	C4.7
C0.7	D27.6	D7.3	D25.1	D22.0	C2.7	D14.4	C11.0	D7.5	D25.2	D6.1	C9.0	D2.4	C1.0	D4.5	C8.0
C2.0	C5.0	D8.7	C2.0	C1.0	D16.6	C4.0	C5.0	C10.0	C3.0	D21.7	D28.3	C4.7	C11.0	D3.6	D1.3
D16.1	C4.0	C8.0	C6.0	C5.0	D30.7	C0.7	D27.5	D23.2	D13.1	D26.0	C7.0	D13.4	D10.2	C3.0	D21.4
D12.2	C10.0	C7.0	D9.6	D2.3	C1.0	D16.4	C4.0	C5.0	C4.7	C11.0	D23.7	D29.3	D30.1	C0.7	D11.4
D3.2	D1.1	D16.0	C4.0	C5.0	C4.7	C0.7	D31.7	D31.3	D31.1	D31.0	D15.0	D11.0	D3.0	D1.0	GOTO Start

When the receiver $\overline{\text{BISTEN}}$ input is set LOW, it initializes its BIST pattern generator and begins searching for the start of the transmitter BIST pattern (see *Figure 4*, time 5). While it is awaiting this start code, $\overline{\text{RDY}}$ and RVS will be HIGH. When it finds the beginning of the pattern (a D1.0 followed by a D0.0, with the proper running disparity), first RVS falls then $\overline{\text{RDY}}$ falls. $\overline{\text{RDY}}$ will remain LOW for the next 510 bytes of the BIST loop and then pulse HIGH for one byte time (see *Figure 4*, time 6). This BIST specific behavior of the $\overline{\text{RDY}}$ output allows an external system monitor state machine to count the number of times that the receiver has checked the BIST data. In non-BIST modes, $\overline{\text{RDY}}$ pulses once per byte in Encoded mode, or once per K28.5 in Bypass mode.

The actual bit pattern appearing at the receiver outputs (Q0–7, and SC/$\overline{\text{D}}$) will match the decoder output for all data patterns, but may not match the data-

sheet pattern for all of the Special Character codes being received. *Table 3* shows the patterns which will appear at the outputs of the receiver while BIST is running. Many of the codes shown do not appear in the datasheet and no correlation should be inferred between these output patterns and the reserved codes mentioned therein. Likewise, if these codes are presented to a transmitter, it will not send the codes necessary to create a good BIST pattern. These codes will typically be monitored by a logic analyzer, and might assist in debugging a particular serial link error phenomenon.

Table 3. HOTLink Receiver Output Patterns during a BIST Sequence

D0.0	C16.0	C20.4	C28.6	C30.7	C15.7	D27.7	D23.3	D29.1	D30.0	C31.0	D15.4	D11.2	D3.1	D17.0	D4.0
C24.0	C22.4	C29.6	D14.7	C11.3	D19.5	D21.2	D12.1	C10.0	C19.4	D5.6	D8.3	C2.1	C1.4	D0.6	C16.3
C4.5	C8.6	C18.7	C5.7	D24.7	C6.3	C9.5	D18.6	C21.3	D28.5	C14.2	C27.5	D23.6	D13.3	D26.1	C7.0
D9.4	D2.2	C17.1	D20.4	C28.2	C30.5	C15.6	D11.7	D19.3	D21.1	D28.0	C30.0	C31.4	D15.6	D11.3	D19.1
D21.0	D12.0	C26.0	C23.4	D13.6	D10.3	C3.1	D17.4	D4.2	C24.1	C6.4	C25.6	D6.7	C9.3	D18.5	C5.2
D8.5	C2.2	C17.5	D20.6	C28.3	C14.5	C11.6	D3.7	D17.3	D20.1	C12.0	C26.4	C23.6	D13.7	D26.3	C7.1
D25.4	D6.2	C25.1	D22.4	C29.2	D14.5	C11.2	D3.5	D17.2	D4.1	C8.0	C18.4	C21.6	D12.7	C10.3	C3.5
D17.6	D4.3	C8.1	C2.4	C17.6	D4.7	C8.3	C2.5	C1.6	D0.7	C0.3	C0.5	C0.6	C16.7	C4.7	C8.7
C2.7	C1.7	D16.7	C4.3	C8.5	C2.6	C17.7	D20.7	C12.3	C10.5	C3.6	D1.7	D16.3	C4.1	C8.4	C18.6
C21.7	D28.7	C14.3	C11.5	D19.6	D5.3	D24.1	C6.0	C25.4	D6.6	C25.3	D22.5	C13.2	D10.5	C3.2	D1.5
D16.2	C20.1	C12.4	C26.6	C23.7	D29.7	D30.3	C15.1	D27.4	D7.2	D9.1	D18.0	C21.0	D12.4	C26.2	C23.5
D29.6	D14.3	C11.1	D19.4	D5.2	D8.1	C2.0	C17.4	D4.6	C24.3	C6.5	C9.6	D2.7	C1.3	D16.5	C4.2
C24.5	C6.6	C25.7	D22.7	C13.3	D26.5	C7.2	D9.5	D18.2	C21.1	D28.4	C30.2	C31.5	D31.6	D15.3	D27.1
D23.0	D13.0	D10.0	C19.0	D5.4	D8.2	C18.1	C5.4	D8.6	C18.3	C5.5	D24.6	C22.3	C13.5	D26.6	C23.3
D29.5	D30.2	C31.1	D31.4	D15.2	D11.1	D19.0	D5.0	D8.0	C18.0	C21.4	D12.6	C26.3	C7.5	D25.6	D6.3
C9.1	D18.4	C21.2	D12.5	C10.2	C19.5	D21.6	D12.3	C10.1	C3.4	D1.6	D0.3	C0.1	C0.4	C16.6	C20.7
C12.7	C10.7	C3.7	D17.7	D20.3	C12.1	C10.4	C19.6	D5.7	D24.3	C6.1	C9.4	D2.6	C17.3	D20.5	C12.2
C26.5	C7.6	D9.7	D18.3	C5.1	D24.4	C22.2	C29.5	D30.6	C31.3	D31.5	D31.2	D15.1	D27.0	D7.0	D9.0
D2.0	C17.0	D4.4	C24.2	C22.5	C13.6	D10.7	C3.3	D17.5	D20.2	C28.1	C14.4	C27.6	D7.7	D25.3	D22.1
C13.0	D10.4	C19.2	D5.5	D24.2	C22.1	C13.4	D10.6	C19.3	D21.5	D28.2	C30.1	C15.4	D11.6	D3.3	D17.1
D20.0	C28.0	C30.4	C31.6	D15.7	D27.3	D23.1	D29.0	D14.0	C27.0	D7.4	D9.2	D2.1	C1.0	D0.4	C16.2
C20.5	C12.6	C26.7	C7.7	D25.7	D22.3	C13.1	D26.4	C23.2	D13.5	D26.2	C23.1	D29.4	D14.2	C27.1	D23.4
D13.2	D10.1	C3.0	D1.4	D0.2	C16.1	C4.4	C24.6	C22.7	C13.7	D26.7	C7.3	D25.5	D22.2	C29.1	D30.4
C31.2	D15.5	D27.2	D7.1	D25.0	D6.0	C25.0	D6.4	C25.2	D6.5	C9.2	D2.5	C1.2	D0.5	C0.2	C16.5
C4.6	C24.7	C6.7	C9.7	D18.7	C5.3	D24.5	C6.2	C25.5	D22.6	C29.3	D30.5	C15.2	D11.5	D19.2	D5.1
D24.0	C22.0	C29.4	D14.6	C27.3	D23.5	D29.2	D14.1	C11.0	D3.4	D1.2	D0.1	C0.0	C16.4	C20.6	C28.7
C14.7	C11.7	D19.7	D21.3	D28.1	C14.0	C27.4	D7.6	D9.3	D18.1	C5.0	D8.4	C18.2	C21.5	D28.6	C30.3
C15.5	D27.6	D7.3	D25.1	D22.0	C29.0	D14.4	C27.2	D7.5	D25.2	D6.1	C9.0	D2.4	C17.2	D4.5	C8.2
C18.5	C5.6	D8.7	C2.3	C1.5	D16.6	C20.3	C12.5	C10.6	C19.7	D21.7	D28.3	C14.1	C11.4	D3.6	D1.3
D16.1	C4.0	C24.4	C22.6	C29.7	D30.7	C15.3	D27.5	D23.2	D13.1	D26.0	C23.0	D13.4	D10.2	C19.1	D21.4
D12.2	C26.1	C7.4	D9.6	D2.3	C1.1	D16.4	C20.2	C28.5	C14.6	C27.7	D23.7	D29.3	D30.1	C15.0	D11.4
D3.2	D1.1	D16.0	C20.0	C28.4	C30.6	C31.7	D31.7	D31.3	D31.1	D31.0	D15.0	D11.0	D3.0	D1.0	GOTO Start

BIST Auto-Abort and Restart

When the receiver detects an error in the received expected sequence of transmission codes it will assert RVS during the byte-time following the error. A normally operating system will rarely experience one error per hour (a bit error rate of $1x10^{-12} \approx$ 1 error/hour @ 266 Mbaud), and systems doing some kind of design tolerance or performance limit testing will usually run with less than a few errors per second (BER of $1x10^{-8} \approx$ 3 error/second @ 266 Mbaud) even during link length testing. At these rates, it can be assumed that each error flagged by RVS was caused by an error that corrupts a single bit. It is impossible to distinguish between single-bit errors and multiple-bit errors within a single byte, since errors are only reported on a byte-by-byte basis. Further, since many kinds of errors change a legal data-byte into another legal byte many errors will be reported at times unrelated to when the error occurred. Single-bit errors can cause changes in the data stream running disparity, and will be detected as errors in the forced running disparity codes.

In extreme cases, where the errors cause PLL cycle slipping, or loss of framing, it is possible to create ambiguous error indications and seemingly endless running error sequences. Once the bit sequence has been corrupted, or after the PLL has bit-slipped, the BIST comparator will indicate a 100% error rate (except for the 32 expected violations that occur as part of the BIST pattern).

Since the BIST generator is a free-running counter that is only initialized while it awaits the start of the transmitter BIST sequence, errors of any kind don't affect the LFSR sequence. This feature can be used to advantage for several types of testing that generate long sequences of errors, since when the errors are removed, the receiver BIST generator predicted data will eventually match the received serial digital data without having to be realigned. Unfortunately, if the error causes the PLL to slip a bit, the received stream will never match.

To account for any loss of BIST sequence condition, the BIST logic included in the receiver will abort an extremely damaged sequence. It will abandon the current sequence and search for the start-of-BIST character and then resume comparisons from the beginning. When this auto-abort happens, $\overline{RDY}$ will go HIGH and remain there until the beginning of a new sequence is detected. While the receiver is waiting, RVS will also remain HIGH. The criteria for Auto-abort requires that there be $\geq$16 RVS indications within 32 contiguous bytes, and is checked every 64 bytes.

For system tests where the user wants to use the BIST comparator to check for longer running errors (and receiver PLL recovery without slipping) it is possible to disable the auto-abort function. The counter that is used to sample the error counter runs on REFCLK. By disconnecting the REFCLK input from the receiver after the PLL has reached the correct operating frequency, the internal counters that manage the error monitor are disabled. (There is a 50/50 probability that when REFCLK is disabled, the auto-abort counter will still be enabled, but by reconnecting, and then disconnecting the REFCLK the auto-abort function can be disabled. The function is controlled by an internal REFCLK divided-by-64 counter. For the first 32-byte times, auto-abort is enabled, for the other 32-byte times it is disabled.)

Tests Using BIST

Built-In Self-Test is a valuable and versatile tool for performing offline-test in any system. It also offers an unambiguous method to examine the performance of HOTLink products and other serial link components. The following short test descriptions are intended to introduce the reader to the capabilities of HOTLink and BIST as an evaluation tool. The tests described are typical of those required to evaluate most physical layer components.

Transmission Line Length

To check for the maximum transmission line length over which HOTLink can communicate, it is only necessary to connect the selected transmission line between a HOTLink Transmitter and Receiver. Most transmission line testing uses arbitrary data patterns that represent typical communication patterns. The HOTLink transmitter and receiver BIST function serves this purpose so the user can check

for an acceptable error rate without extra test equipment and without reconfiguration of an operational link just to perform this test. Transmission lines can be extended or modified until RVS indicates an unacceptable error rate. Tests that might use BIST to indicate system margin include;

- fiber-optic optical attenuation budget and optical or electrical margin testing;
- wire transmission-line attenuation, crosstalk, emissions and noise susceptibility testing;
- electrical interface connections and signal-margin testing;
- data sources for serial interconnect hardware testing.

Rx Jitter Tolerance

The ultimate performance of any serial link is determined by the performance of the receiver. The function of the receiver is to recover data from a (seemingly arbitrary) serial data stream. This data stream is translated several times, coupled to and though several non-linear devices and subjected to all manners of distortion. The receiver must accept this serial pulse train and recover a high-speed bit synchronous clock, de-jitter it, and then separate the DATA from the CLOCK. Jitter tolerance is the typical term used for this function. HOTLink receiver jitter tolerance can be measured by connecting a suitable transmission media between the transmitter and receiver, and inserting a jitter generation source similar to that shown in *Figure 10*. By inserting measured jitter amplitudes and watching the RVS output of the receiver, jitter tolerance can be measured.

There are two basic types of jitter that must be accommodated, deterministic jitter and random jitter. Deterministic jitter is comprised of data dependent jitter (DDJ) and duty cycle distortion (DCD). DDJ is caused by imperfections in the serial link that cause signal corruption that is proportional (or at least a strong function of) the particular data stream. DCD is caused by imperfection or imbalances in the serial link that cause signal corruption that is related to the timing of the rising or the falling edges. Random jitter is unrelated to the data stream, the edge rates, or the link quality. It is typically caused by external noise events or by thermal noise in the optical components. Random jitter is uncorrelated to the data stream and is difficult to reproduce experimentally.

DCD creates high-frequency jitter at about the bit rate of the serial data stream since bit placement errors are complemented within the pulse that is distorted. DDJ creates high-frequency jitter at about the bit rate of the serial data stream since bit-placement errors are usually complemented within a bit or two. These high-frequency jitter components should be filtered by the PLL filter, and should

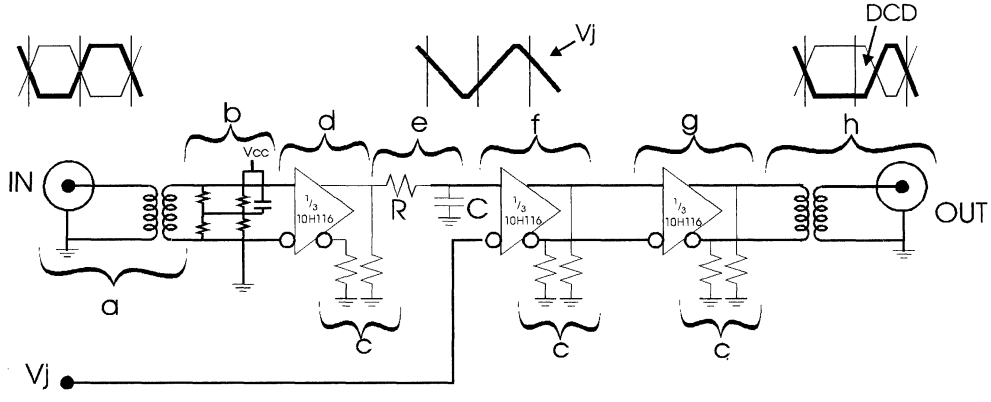


Figure 10. Jitter Generator Schematic

cause no significant jitter at the CKR output of the receiver. DDJ can cause baseline-wander at about the byte rate of the serial data stream, but since the 8B/10B code is balanced over multiple bytes, there should be little or no low-frequency components in the jitter. Random jitter has both high- and low-frequency components, and will cause output jitter as it causes the PLL to attempt to track a corrupted data stream. All three types of jitter must be accommodated by the receiver as it captures the data and aligns its serial clock.

Data dependent jitter can be generated by a suitable length of coaxial cable. If DDJ and input amplitude must be separately measured, an external line receiver and level restoration circuit might be needed. Duty cycle distortion can be generated by the circuit shown in *Figure 10*. This circuit uses the stages in a 10H116 (ECL triple-differential amplifier) to perform; (1) Differential-to-single-ended transformation; (2) Ramp generation; (3) Threshold shifting; (4) Level restoration; (5) Differential buffering. In this circuit the transmitter data stream is fed through the jitter generator while the receiver monitors and checks for correct operation. As the control voltage (Vj) input is varied between the 10KH Vil and Vih levels, the duty cycle of the data stream is corrupted in a repeatable and measurable manner.

Serial-data input to the jitter generator can use any appropriate connector, and coupling circuit. The connector and transformer shown at (a & h) will work with coaxial cable or STP cables. For fiber-optic interfaces, these could be eliminated by direct coupling to fiber-optic receiver/transmitter modules. Transmission-line termination and DC threshold adjustments are performed by the simple network shown at (b). The first differential stage of the '116 (d) is used as a differential-to-single-ended converter with a controlled output impedance and symmetrical rise and fall times. The ECL output termination resistors shown at the outputs of each differential stage (c) may be replaced with parallel termination resistors if better impedance control or closer edge-rate matching is required. The R-C ramp generator at (e) must be tuned to each data rate, to insure that 100% voltage swing is maintained for the narrowest pulses expected. If the Ramp is too long, it will be possible to raise Vj above the level of some data bits, thus losing data. The second differential stage of the '116 (f) serves as a voltage comparator between the control voltage (Vj) level and the level of the signal at the output of the ramp-generator. Additional DC-filtering may be required between the Vj input and its input to (f) to insure that high-frequency, single-ended noise does not corrupt the data flow. The third differential stage of the '116 (g) is used to restore crisp-edged, full-swing levels to the serial data, and to drive the subsequent transmission line. Further details on the fabrication of the jitter generator and the measurement techniques required for accurate measurement of this injected jitter is beyond the scope of this note.

Receiver Error-Free-Window Test

A normally operating receiver PLL will adjust its internal clock such that incoming data transitions are placed at the maximum distance from the data-separator flip-flop sampling window. This placement allows misplaced transitions (jittered edges) the maximum margin before data-misinterpretation occurs. The width of this error free zone is commonly called error-free-window. It is less than the actual bit width (expressed in nanoseconds) by the sum of maximum peak-to-peak receiver PLL jitter, data-separator flip-flop sampling window width, and absolute misalignment of the internal PLL sampling clock. (Actual test results will be additionally affected by clock source jitter and test equipment trigger and measurement inaccuracies.)

To measure the error-free-window (EFW) in the HOTLink Receiver, it is only necessary to connect a HOTLink Transmitter and receiver in BIST mode, while controlling the serial data stream with the transmitter FOTO pin. The FOTO input to the transmitter causes the OUTA+ and OUTB+ outputs to be LOW, and the OUTA− and OUTB− outputs to be HIGH for the time that FOTO is HIGH. (For purposes of this example it will be assumed that Tx−OUTA+ is connected directly to Rx−INA+ and that Tx−OUTA− is connected directly to Rx−INA−.) Since FOTO is an asynchronous TTL input, it is possible to use it as a Controlled Data Corrupter that can move an edge away from its nominal

position. The limits of the EFW will be signaled by an indication on RVS.

To set up the test, the user would connect a pulse generator to the transmitter FOTO pin. This generator would be triggered by the $\overline{RP}$ output and would be controllable in both delay and pulse width. Since $\overline{RP}$ pulses once each BIST loop, the generator would make pulses that were phase aligned with the serial data stream. By careful adjustment of pulse width (VERY narrow, adjustable-width pulses) and delay (alignment such that the forced LOW is placed in a position that is naturally LOW) it is possible to measure the EFW.

To perform the test, the user should first adjust the generator so that it causes no corruption in the actual data stream. Then, by carefully adjusting the delay and/or width of the generator a specific edge in the data stream can be realigned until the RVS output indicates a BIST error. By noting the position of the realigned edge relative to its nominal position, the early and late limits of the EFW can be measured.

The relationship between the FOTO pulse and its effect on the OUTA transition must be measured empirically. The OUTA+ expanded waveform shown in *Figure 11* illustrates the control that can be effected by FOTO. The vertical lines (Internal Rx Sampling Locations) indicate the location of ideal receiver sampling points, and the shaded regions around them indicate the built-in errors that limit EFW.

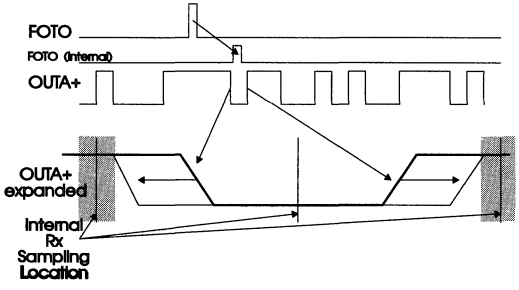


Figure 11. Example of Error Free Window Testing

Since FOTO only forces the OUTA+ & B+ output to a LOW (and OUTA− & OUTB− to a HIGH), it is not possible to check for rising and falling edge symmetry with this test. A falling edge can only be forced to an earlier LOW, and a rising edge can only be forced to a later HIGH. By careful adjustment of the FOTO generator, it is possible to adjust the position of all of the various 1, 2, 3, 4, and 5 bit-length pulses found in the 8B/10B code.

Rx Run-Length Tolerance Test

An extension of the EFW test will allow the user to measure the receiver's tolerance to missing pulses. If the pulse width of the FOTO generator described above is increased beyond a few bits, the resulting data stream will have missing pulses beyond the 5-bit run-length of the 8B/10B code. These missing transitions allow the PLL control voltage to drift, causing an arbitrary phase change. When the transitions resume, the PLL realigns to the incoming bit stream. However, if the phase drift has gone beyond the jitter limit, the PLL may align to a different bit position than the one to which it was previously aligned (see *Figure 12*). This realignment is commonly called cycle slip and equates to the loss or addition of a bit to a serial data stream.

Obviously the RVS output will indicate an error, as shown in *Figure 13*, while the data is masked, but since the indicated error is bounded (i.e., recovers within a few bytes), the BIST detector shows that the receiver is able to continue finding good data within a few bits or bytes of resumption of the sequence.

As the FOTO pulse width increases from a few bits to a few bytes, the RVS indication widens proportionately. There may be positions where a minor change in width causes multiple byte errors and others where multi-bit width changes cause the RVS to show apparently good data. The former is an indication of a running disparity error which might run for several bytes before being terminated by a code in the BIST sequence. The latter is an indication that at that particular position in the sequence, BIST was already expecting a violation, so would not flag an error for this type of data corruption.

When the FOTO pulse width (or the RVS pulse width) approaches 16 bytes, the BIST-Auto-abort

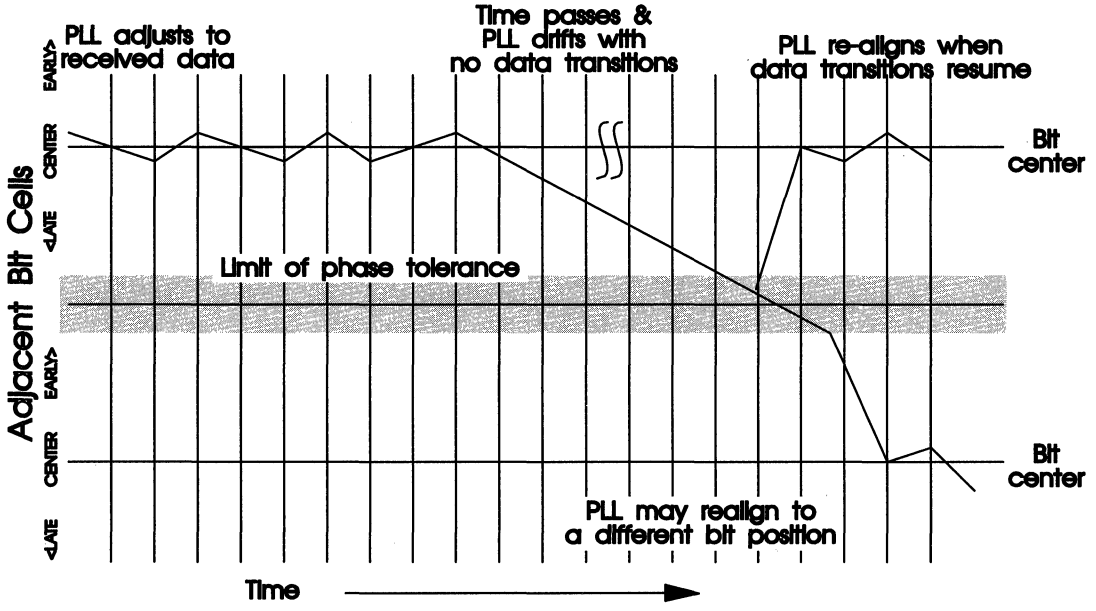


Figure 12. Long Spaces without Transitions May Cause Cycle Slip

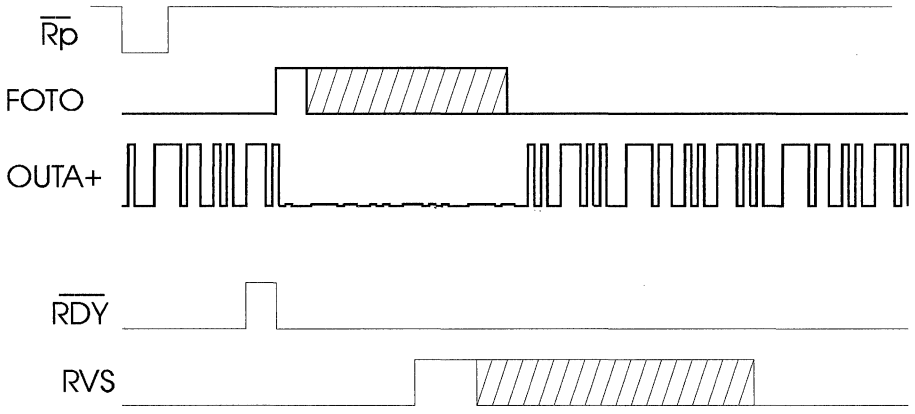


Figure 13. Missing-Transition Test Timing Diagram

mechanism described in an earlier section will begin to obscure the real receiver tolerance. When the error run length approaches 16 bytes, the RVS width will become almost continuous. $\overline{RDY}$ will cease its normal pulse-once-per-loop operation and will rise during the RVS pulse, and stay HIGH for the remainder of the BIST loop as the receiver BIST checking circuit waits for a start-of-BIST pattern. If $\overline{RP}$, RVS and $\overline{RDY}$ are all simultaneously displayed on the oscilloscope screen, it will be noted that the

BIST loop appears to start correctly, but that after the FOTO pulse, all the data is corrupted. This is the automatic-restart behavior and is characteristic of the BIST-auto-abort logic, not an indication of a real corrupted data stream.

HOTLink Receiver can tolerate nearly 100 transitionless bytes without cycle-slipping so meaningful testing requires suppression of the auto-abort function. As described earlier, this BIST-auto-abort logic can be suppressed by removing the REFCLK input from the receiver.

Even with the BIST-auto-abort logic disabled the pulse width of the FOTO generator (and thus the number of missing data transitions) will ultimately become long enough that the receiver PLL will cycle slip before data resumes. Since the BIST comparator requires an absolutely perfect data stream and cannot realign without external assistance, the BIST checker will show that the all of the received data is incorrect. Once the RVS indication becomes continuous it will be necessary to either reconnect REFCLK (and allow the Auto-abort logic to reinitialize the receiver BIST generator) or to toggle the $\overline{\text{BISTEN}}$ input on the receiver (which forces the BIST generator to begin from the beginning). The FOTO-generator pulse width (expressed in bit times) that causes an unrecoverable error is the missing-transition limit of the HOTLink receiver (i.e., run length tolerance).

Figure 13 illustrates signals involved in the run length tolerance test. The actual time measurement will be affected by the timing of the FOTO pulse, and the jitter added by the interconnect link. Care should be taken to assure that the first missing pulse comes after a normally placed pulse (i.e., make sure that FOTO takes effect while the data stream is naturally LOW and that it does not disturb the position of the last transition before it kills the data stream). If the receiver PLL is recovering from a large phase correction at the time it is left to float, reduced run-length tolerances will result.

Reframe-CKR Stretch

In normal systems it is difficult to cause the HOTLink Receiver to reframe off established byte boundaries using normal transmission data. The HOTLink BIST sequence includes one occurrence of a bit pattern that mimics a K28.5 aligned to incorrect byte boundaries. To view this clock stretch behavior, it will be necessary to synchronize the oscilloscope with the $\overline{\text{RP}}$ of the transmitter, and delay the display to the area of the alias sync. *Figure 14* shows the effect of an alias sync (five bits misaligned). In this example, taken from the BIST sequence, the C0.7−D11.5 cause an alias-sync realignment. The next several bytes are corrupted because of this misframing. When the C2.7 (+K28.5) arrives, it realigns the data to the proper boundaries. In the six byte times between the C0.7 and the C2.7 there have been two clock-stretching events and only five bytes have come out of the receiver (all bad without any RVS indication). Please note that this illustration shows the function of the receiver and is not intended to show actual timing with respect to the serial data stream.

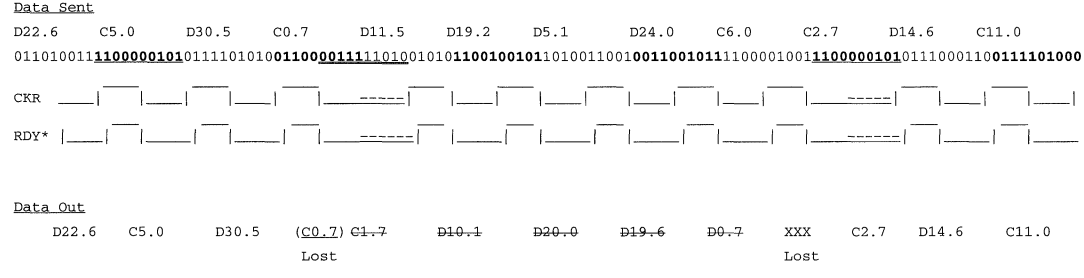


Figure 14. Illustration of Receiver Behavior during Reframe Clock Stretch

If the receiver RF input has been HIGH for less than 2048 bytes, this single alias-K28.5 (double-underlined in *Figure 8*) will cause a byte realignment (reframe) to the incorrect byte boundary (five bits off of the real byte alignment) and thus a stretch of CKR, $\overline{RDY}$ and the position of Q_{0-7}, SC/$\overline{D}$ and RVS until the next properly aligned K28.5 (approximately five bytes later). In the illustration, the C0.7 indication is lost because of the reframe caused by the alias sync and the adjacent clock edges are separated by fifteen bits. Similarly, when the real K28.5 arrives (C2.7 in the example), the Q_{0-7}, SC/$\overline{D}$ outputs will change twice between adjacent clocks (i.e., internal bit 0) between the D0.7 and the C2.7 (i.e., once on the old bit 2 and again on the new bit 2). These adjacent clock edges are separated by fifteen bits and the specified set-up and hold times for subsequent logic will be assured.

After the receiver RF has been HIGH for more than 2048 bytes, the internal byte framer changes from requiring a single K28.5 to re-align the byte, to requiring two K28.5s to reframe. To keep the receiver in single byte-framing mode, and to perform this test it will be necessary to pulse the RF input at a rate less than once per 2048 bytes (maybe triggered by $\overline{RP}$/4).

An alternative method to show byte realignment and CKR stretching involves sending a string of data that includes a positive running disparity K28.7 (C7.0) followed by D11.x or D20.x or by sending a positive running disparity SVS (C0.7) followed by D11.x. (e.g., C0.7 = 0110000111 or 1001111000 and D11.x = 110100xxxx or 001011xxxx so if the correct running disparity SVS is followed by the correct running disparity D11.x, a five bit misaligned-alias-sync is created as follows; 01100**00111110100**xxxx)

Receiver Offset Frequency

Differences in frequency between the transmitter crystal oscillator and receiver REFCLK crystal oscillator might limit performance of the data communication link. The HOTLink datasheet specifies that the receiver and transmitter frequencies can be different by ±0.1% (1000 ppm) without compromise to the reliability of the data link. This parameter is conveniently checked by operating a transmitter CKW on one generator or crystal oscillator, and the receiver REFCLK on another. If both HOTLink parts are operating in BIST mode the RVS output will indicate the quality of the link.

As the generator frequency is adjusted (slowly and smoothly) the RVS should stay LOW indicating correct operation. RVS may show errors when the generator frequency is adjusted, though it is unlikely. If this happens, it is probable that the frequency change is being made too abruptly. The test is still possible, if RVS is checked only after the generators stabilize at each new frequency.

Tolerance to Phase Changes in Received Data

Two transmitters operated from the same clock source will run at exactly the same data rate. If they are both in BIST mode and synchronized by simultaneous assertion of the SVS input, they will also be sending exactly the same serial data. If their respective clocks are phase adjusted over a narrow delay range, they can be used as a source of synchronized serial data with a known phase relationship.

The receiver has two equivalent serial inputs (INA± and INB±) which can be independently selected. If the two transmitters are each connected to one of the serial data inputs, and if a synchronized source alternately selects one, then the other (using A/$\overline{B}$ Select), the receiver's phase adjustment behavior can be examined. (See *Figure 15*.) Synchronized switching is easily accomplished by using the $\overline{RP}$ output of one of the transmitters to trigger a long-pulse-width ECL generator (200–300 bytes pulse width, carefully aligned so that the change happens during a quiescent portion of the serial stream).

As the two transmitters are alternately selected and as the delay between them is increased, the receiver sees a continuous BIST data stream containing instantaneous phase changes equal to the difference in transmitter-to-transmitter, clock-to-clock skew. It must adjust to the new data phase and realign its internal clock to correctly recover the data. The theoretical maximum phase adjustment range is slightly less than ±0.5 bit time (i.e., ±0.5 bit less Rx PLL jitter, static alignment, and flip-flop set-up/hold times). When the phase difference reaches the

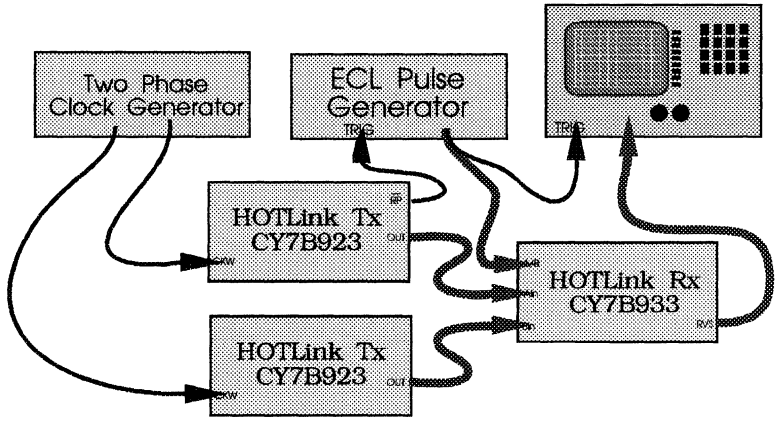


Figure 15. Receiver Phase Tolerance Test Setup

limit, errors will be indicated by pulses on RVS that are one or more bytes wide. (Even though the actual error might involve only one bit, in one byte, the RVS indication may run for several byte times because of running disparity corruption.) As the data phase hop increases, the RVS pulses will increase in width proportional to the time taken to adjust the phase of the internal PLL.

Eventually RVS will stay high continually from the time of the A/B switch to the next $\overline{\text{RDY}}$ pulse (i.e., the start of the next BIST loop). As the magnitude of transmitter clock-to-clock phase difference approaches the point where the PLL phase alignment slips from one bit to the next (i.e., at approximately 180° phase difference) the BIST loop will become irreversibly corrupted and will auto-abort-restart after each phase hop.

Conclusion

HOTLink BIST capability should help system integrators add features to high-performance communication links. These features can be made to enhance usability and improve reliability of the link.

Test methods that use BIST will aid in evaluation of HOTLink products and other link support hardware. The HOTLink built-in test features allow an unambiguous indication of data quality, many of which require only inexpensive test equipment.

HOTLink is a trademark of Cypress Semiconductor Corporation.
ESCON is a trademark of International Business Machines Corporation.

HOTLink™ Jitter Characteristics

Abstract

This application note describes the basics of jitter in transmission systems and, using HOTLink™ as the example, shows how it can be analyzed and measured. Specific characterization data is presented that will allow system integrators to understand the parameters needed to improve the reliability of their systems.

Introduction

This note examines jitter from three different perspectives. First, as a background overview, it describes a few basic "jitter" concepts that affect digital systems. Second, it describes the jitter performance and characterization of the HOTLink Transmitter (CY7B923). Third, it describes the jitter tolerance and feed through characteristics of the HOTLink Receiver (CY7B933).

Numerical characterization data is supported by descriptions of the various testing techniques and equipment that are required to obtain this information. Commercial, custom, and "home-brew" test equipment are described along with the connections used to gather data that illustrates the levels of performance attainable by HOTLink products.

The data contained in this application note will help users to understand the various characteristics of link components and HOTLink characteristics and capabilities. This data is offered to assist in the design of robust serial interconnect links.

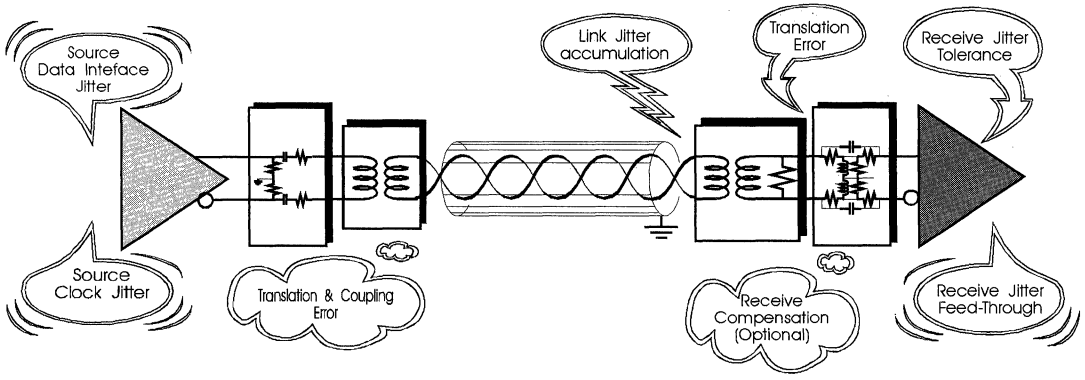


Figure 1. Link Jitter Budget Depends on Link Components

Jitter

Jitter is a high-frequency semi-random displacement of a signal from its ideal location. These displacements can occur in amplitude, phase, and pulse width, and are generally categorized as either deterministic or random. For data communications links based on (or similar to) HOTLink, measurement and specification of jitter is usually restricted to timing displacements.

Deterministic jitter are those timing variations that are repeatable within a system and whose cause can generally be directly attributable to specific physical components or events. An example of this would be the jitter caused by the frequency selective attenuation and phase delay of a signal in a transmission line.

Random jitter deals with those timing variations that are much more probabilistic in nature. While still observable and measurable in a system, this jitter is not directly predictable. Common sources for random jitter are thermal and electrical noise, both internal to and injected into a system or component.

Jitter in logic circuits is often characterized by its transfer function. This function, known as jitter feedthrough, is a measure of jitter output relative to jitter input of a system or component. Most circuits, when presented with jitter, tend to amplify that jitter in a few or many areas. Fortunately for data communications system (which are plagued by high jitter creation elements), application of properly designed PLLs (phase-locked loops) can actually reduce or remove large amounts of jitter from a clock or data stream.

Background—Jitter in Logic Systems

The timing of logic signals flowing through a logic system are often assumed to be a series of simple voltage transitions that occur after some fixed delay. While this is a convenient and usually sufficient assumption for the logical function of a device, it is insufficient to analyze the limits of the timing or the reliability of the design.

The delay through logic devices (i.e., gates, flip-flops and other common building blocks) is defined to a first order by the time it takes for the inputs, the internal circuit nodes, and the outputs to change from one voltage to another. Since there is always some uncertainty about the exact voltage present at any node in the circuit, various logic families have been devised with specific ways to assure reliable logic functions. Thresholds are well defined and intergate links have sufficient voltage margins to assure reliability. Typical components have output levels (e.g., Voh, Vol, etc.) that assure a significant voltage margin above and below the input thresholds (e.g., Vih, Vil, Vth, etc.).

Most logic model libraries document a fairly wide range of possible delays through a logic element. This range includes the effects of many internal characteristics such as differences in output resting voltage, threshold voltage, signal ramp rates, and (to some extent) the speed the signals travel along the interconnecting wire, metalization, and leadframes. These delays, while supposedly covering the minimum to maximum range for the part, assume specific external operating and signal conditions. By presenting the logic element with input, output, or power conditions beyond those assumptions, it is possible for these logic elements to exhibit apparent delays both faster and slower than the specified minimum and maximum.

The noise carried on the V_{CC} or Ground rails (both internal and external) affect the actual timing of the I/O transition by causing changes in the starting levels of the active transition. The illustration in *Figure 2* shows only the timing variation caused by ground bounce, but the influence of V_{CC} noise has a similar effect. If the signal begins its transition at some arbitrary but fixed time, and has a transition rate (i.e., rise time or fall time) that is mostly controlled by slew rate limiting effects not related to the power supply glitch, the effective timing will be determined by the placement of the glitch. If the transition begins on a glitch-peak, it will arrive at the threshold voltage a little early, and if the transition starts in a glitch-valley, it will arrive a little late. This change in timing is usually invisible to the external examiner (except as power supply induced timing variation) because much, if not all of the glitch is contained within the IC package, and is not externally observable.

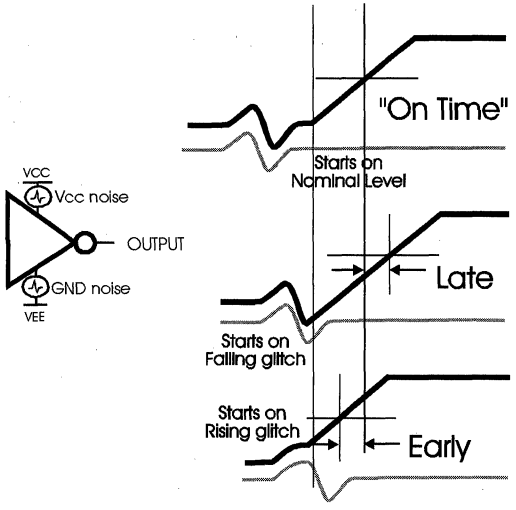


Figure 2. Power Supply Glitches Affect I/O Actual Timing

The effect of this variation in starting voltage can cause significant variations in timing. A signal that has a 1 ns/V ramp rate (TTL edges are usually between 1−2 ns/V, and can be much slower), will have an effective change in delay of about 1 picosecond per millivolt of disturbance. This equates to ±100 picoseconds of delay variation for ±100 millivolts of ground or V_{CC} noise, an amplitude which is normally deemed "quiet". When noise spikes approach 1 volt, delay variations could be expected to exceed 1 nanosecond. With a volt of power supply variation, other delay effects would surely begin to appear.

Additional timing variation can be caused by noise coupled into the external or internal logic through cross coupled logic paths (including package-pin crosstalk), or by power supply noise injection. These "minor" variations in delay are typically ignored in the analysis of the logical function, since there is sufficient overdrive (voltage noise margin) to assure that the logical function is achieved. However this assurance is not transferred to the timing margins of a logic design.

Most of the delay of today's high performance logic is caused by an output "ramping" from its resting voltage to the actual threshold voltage (the voltage at which the gate begins to make its logical decision and subsequently change its own outputs). Any disturbance in either the internal threshold or the ramping input or output will cause a change in the apparent delay through the gate (see *Figure 3*). All single-ended logic gates suffer from this variable-delay characteristic. Single ended circuits include all

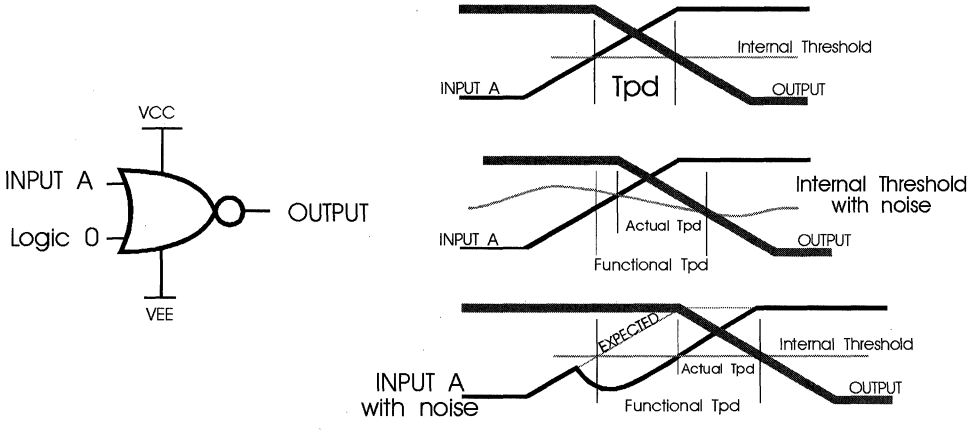


Figure 3. Delay Through a Logic Gate Changes with Injected Noise

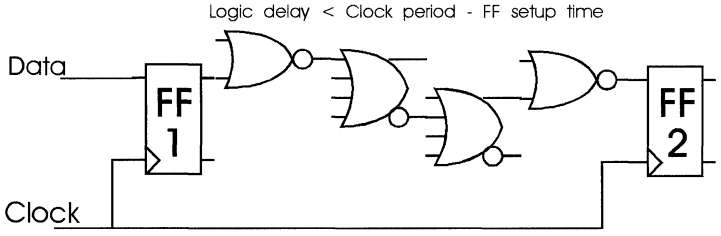


Figure 4. Typical Logic Path Delay Limited by Minimum Clock Period

TTL, CMOS, and any ECL logic that uses an internal or external threshold reference.

Differential circuitry can be used to partially mitigate the effects of injected noise, since the threshold of the gate is determined by a complementary output, hopefully carrying the same injected noise, but ramping in the opposite direction. The common mode range of such a differential gate helps to reduce many noise induced delay characteristics. All of the critical timing paths in HOTLink products are implemented with differential CML (Current-Mode Logic) signals to minimize crosstalk and V_{CC}-coupled noise-jitter effects.

Various design techniques have been developed that maximize timing margins in logic, but in most of these techniques the timing of any particular logic element is considered a constant (or a range of constants). Except for the well known metastability characteristic of storage elements, the design tools assume that each element has a fixed delay, and the only accommodation to metastability is to attempt to avoid the conditions that provoke the unpredictable behavior.

Traditional design practices work on the simple assumption that if the logic path (delay) between storage elements is less than the time between clocking edges by some comfortable margin, then the logic will behave exactly as the designer intended. As clock speeds increase and as product complexity increases this comfortable simplifying-fantasy becomes more difficult to maintain. As is well documented in other literature, if the transition on the DATA input changes later than the required set-up time prior to the active transition on the CLOCK input, the delay of FF−1 *(Figure 4)* may increase or it may refuse to store the expected data. If the path length to FF−2 is running near its maximum limit, this increased delay could propagate through the logic causing unexpected and undesirable results.

Designs that meet all manufacturers specified set-up and hold times can also experience variable delays through the flip-flop. As the input transition approaches the "actual" set-up time of the internal latches, delay will begin to change. *(Figure 5)* Typically, Tsetup is specified at the point where delay has changed by less than some arbitrary amount (usually about 10%) of the cell's "nominal" delay. Inside of that point, delay will increase radically until the flip-

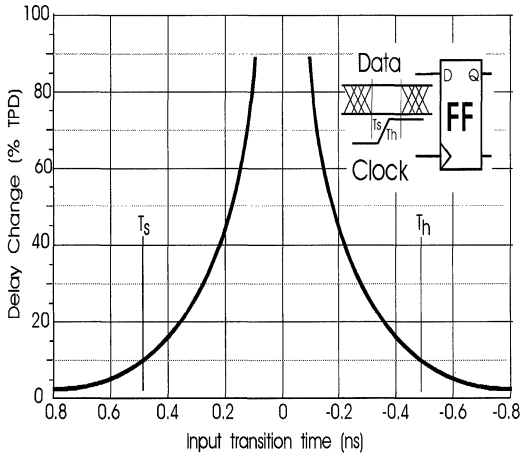


Figure 5. Propagation Delay Changes as Actual Tsetup is Approached

flop goes metastable. Similar effects occur as hold time approaches zero.

Even if the nominal delays of the intervening logic are within design margins, voltage-noise effects can change the delays of the combinational logic devices. If that happens, metastable effects might be observed in the system. Normally in digital-logic systems, great care is taken to assure adequate timing margin and then the error rate is "assumed to be zero," and ignored.

Jitter in PLL Systems

Phase Locked Loops are typically used as high speed clock multipliers or as precision clock recovery circuits. In their role as clock source generators, PLLs are characterized for their timing precision. This is usually because any jitter that appears on the clock line must be compensated by an equivalent reduction in the timing margin allowed between flip-flops.

Jitter can enter a multiplier PLL (see *Figure 6*) in several ways. The Clock input (1) can contain voltage-coupled noise or phase-noise that will affect the multiplied Bit Clock. The UP and DOWN outputs (2) of the PHASE FREQ DET are the digital to analog interface with the analog control circuits of the PLL and can suffer from the same voltage-coupled noise effects described earlier for logic. These digital signals carry the picosecond analog-timing information that controls the VCO. Any cross-talk or noise injection at this point will corrupt the "error" information that the PLL uses to maintain phase-lock with the input clock.

The output of the analog filter (3) contains both the gross center-frequency control, and the precision phase-control. Typically the input sensitivity of the VCO will be hundreds of megahertz per volt, and micro volts of crosstalk or power supply noise injection can add nanoseconds of jitter to the PLL output. Similarly, the capacitors (4) used in the FILTER (either internal or external) can be susceptible to noise injection which cannot be eliminated by any traditional circuit techniques.

HOTLink products use carefully designed, fully internal Metal/Oxide/Silicon (MOS) capacitors. These huge, matched devices minimize external noise coupling. For noise sources that cannot be avoided, the capacitors and all of the other analog circuitry are designed to make coupled noise more rejectable by using fully differential, common-mode noise reduction methods. Older PLLs often used ex-

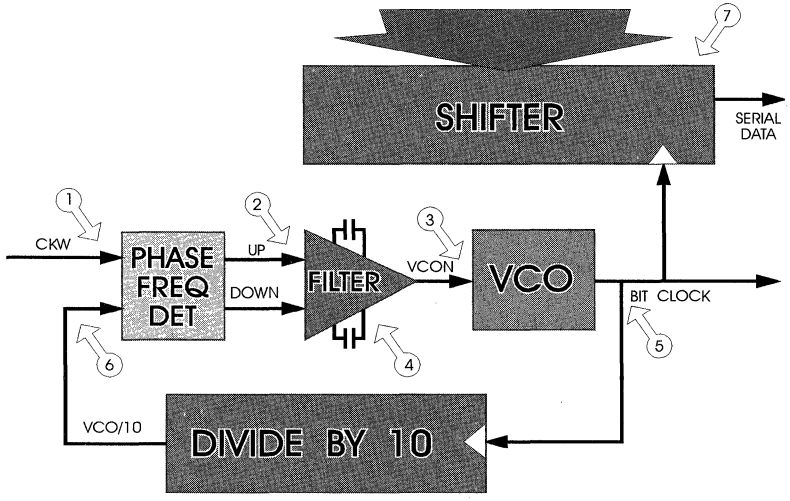


Figure 6. Clock Multiplier PLL Noise Injection Points

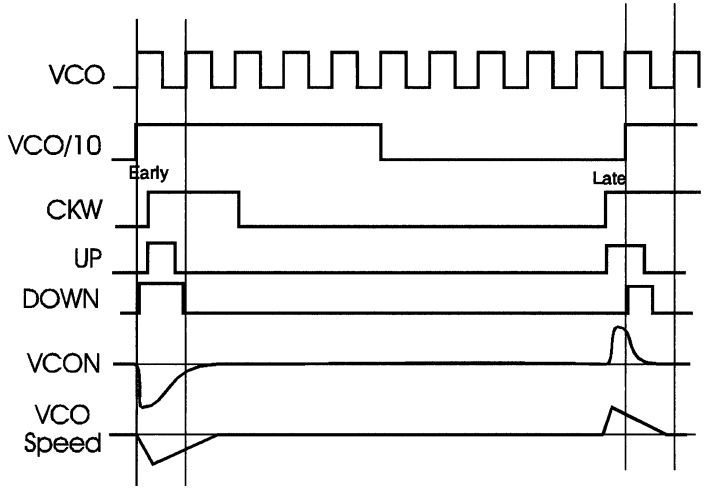


Figure 7. Phase/Frequency Corrections in Multiplier PLL

ternal capacitors which were notorious for noise injection through the external pins and circuit board traces required to connect these capacitors.

Noise injected at (2), (3) or (4), and to a lesser extent at the other points, can be only partially compensated by the normal filtering actions of the PLL. Noise at (2) or (6) will exhibit different effects than noise injected at (1) and will affect the Bit Clock (5) in different ways. These differences are illustrated in *Figure 11*, and will be discussed later.

Since the multiplier PLL only receives its correction information once every N VCO cycles (where N is the multiplication factor of the PLL, the VCO frequency divided by ten in this case), many specific errors will not cause a correction. Only the "average" of noise-induced errors will result in compensatable disturbances. "Instantaneous" errors will not be compensated by the PLL at all, especially if there are other errors of similar magnitude and opposite sign between reference updates.

Logic noise as described earlier can be injected into the Recovered Bit Clock (5) or at the feedback reference (6). These can be avoided by careful differential circuit and logic design. The parallel data input to the SHIFTER (7) can cause transmitted output jitter which is a function of the data pattern being sent (i.e., DDJ) as the set-up and hold times of the output flip-flop vary.

The operation of the PLL can cause jitter just by its normal operation *(Figure 7)*. Whenever the phase detector adjusts the frequency of the VCO, it causes an instantaneous change in phase as part of the adjustment operation. This instantaneous phase change, followed by a drift until the time of the next correction, is the normal operation of the loop. Ideally, the correction would be small, and entirely contained within one clock cycle, but if it is larger or lasts longer than one cycle of the VCO, it can cause bit-to-bit phase differences (i.e., jitter).

Clock Recovery, Data Separator PLL

The PLL used for clock synchronization and data recovery shown in *Figure 8* is different from the one described in *Figure 6*, which is used as a clock multiplier. The phase correction information comes from comparisons between an arbitrary input pulse stream and an internal bit-rate VCO. In contrast to the Phase-Frequency Detector (PFD) used in the clock multiplier, this PLL uses a detector that is sensitive only to phase errors. Missing data transitions are ignored, and corrections follow each and every data transition. In contrast to the predictable correc-

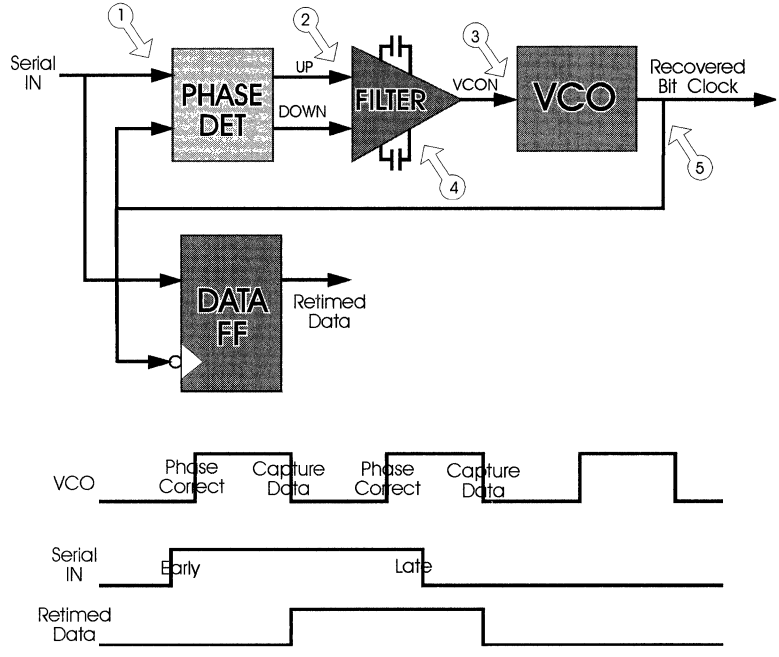


Figure 8. Receive-PLL Block Diagram

tion rate of the PFD, the Phase Detector will make corrections at the rate of the incoming data. It can vary from one correction per VCO cycle (when data contains alternating 10101...) to once per byte (or less) for some serial protocols. This variation in correction density can cause some forms of jitter and, by affecting the loop stability and bandwidth characteristics, will affect jitter feed through.

Jitter can enter a synchronizing PLL in several ways. The input data (1) will contain significant jitter which accumulates on the serial transmission link. This is the jitter that the receive PLL is intended to remove.

The noise injection points at (2), (3), (4), and (5) are the same as those in the multiplier PLL, and affect the receiver PLL in similar ways. The main difference is that this PLL gets a phase-error update on each input data transition. This allows noise events to be corrected more often than those in the multiplier PLL, but the noise induced corrections can be affected by the corrections already required by the jittered data. Conversely, these noise-induced jitter components reduce the data-recovery circuit's tolerance to input data jitter.

The Phase Detector (or PFD) in clock multiplier PLLs and in clock synchronizer PLLs is intended to give a "unit" of phase correction information for a "unit" of error. This correction should be directly proportional to the error, regardless of error magnitude. A poorly designed (or poorly implemented) phase detector in any PLL, either a multiplier or clock synchronizer loop, can exhibit what is typically called a "dead-zone" if the error/correction relationship does not hold for miniscule errors. This effect is illustrated in *Figure 9* as the less-than-ideal transfer function which effectively removes the phase correction control in the neighborhood of "zero error." This "hole" in the transfer function will cause an otherwise perfectly locked loop to exhibit jitter because the loop will be unable to maintain control and will wander between the two inflection points.

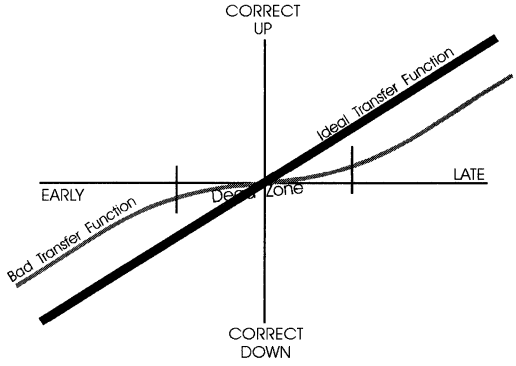


Figure 9. Phase Corrections Should Be Linear with Error Magnitude

HOTLink Transmitter and Receiver PLLs have been designed to eliminate this undesirable behavior.

The closed-loop PLL acts like a Low-Pass Filter to incoming noise (*Figure 10*). All frequency components that fall below the roll-off frequency of this filter are passed unattenuated. Frequencies above the roll-off frequency of the filter are attenuated, and frequencies around this point might be amplified to some extent. Some forms of jitter have low frequency characteristics that will pass through the PLL and appear on the resulting high frequency clock output (e.g., low-frequency wander passes unattenuated through the Receive PLL).

The PLL low-pass filter model is valid for jitter that enters the system at the PLL input. However, jitter that is injected (or is present) *inside* the loop "sees" the loop as a high-pass filter. The dynamics of the closed loop system allow it to compensate for low-frequency injected jitter with an automatic (and opposite) low-frequency phase adjustment. As the frequency of the injected jitter rises toward the roll-off frequency, the loop becomes incapable of fully compensating the injected jitter. Above the roll-off frequency, the loop will pass injected jitter without attenuation (see *Figure 11*).

Since V_{CC} noise is injected (and can result in undesirable delay variations) at multiple points inside the loop, the resulting component of jitter attributable to V_{CC} noise will probably show a peak at approximately the roll-off frequency, and less output jitter at both higher and lower frequencies. The ratio of jitter magnitude for V_{CC} noise frequencies above and be-

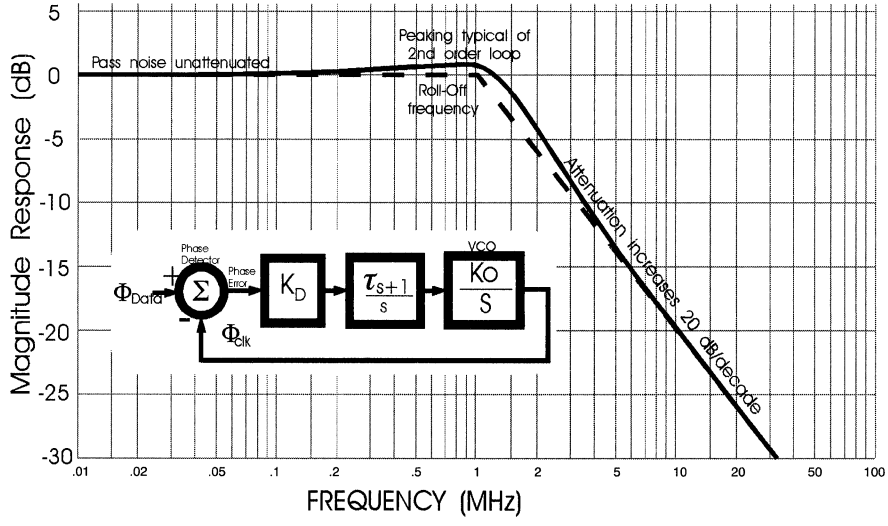


Figure 10. PLL Closed Loop Response is a Low Pass Filter

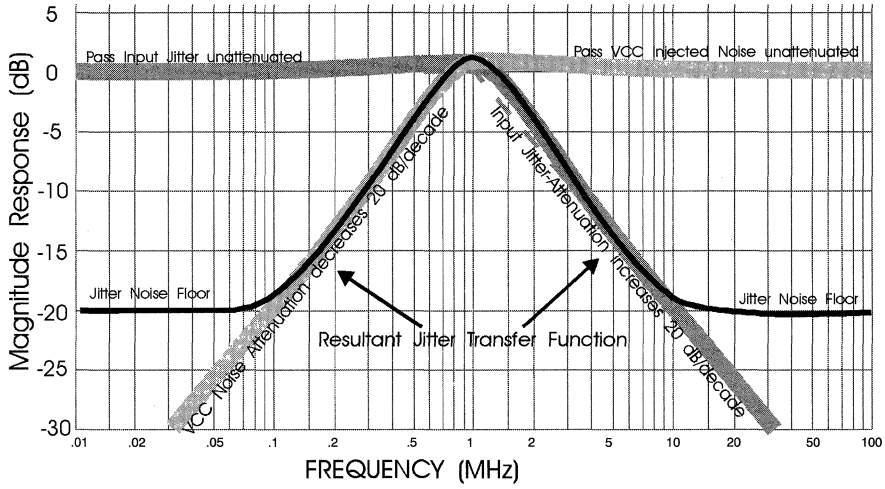


Figure 11. V_{CC} Noise Injection Transfer Function

low the PLL roll-off frequency might give a clue to the probable noise injection point(s). Larger jitter magnitude for V_{CC} noise frequencies below roll-off is probably attributable to logic delays in the input path the PLL (e.g., logic delay changes in gates prior to the loop, passing through the PLL low-pass filter). Larger jitter magnitude for V_{CC} noise frequencies above roll-off is probably attributable to noise injection into some circuit inside the loop (e.g., jitter inside the loop being affected by the PLL as a high-pass filter). Jitter magnitudes that are approximately the same above and below roll-off are probably attributable to delay changes being introduced into circuits that follow the loop.

Jitter on a Multiplier-PLL clock can affect the operation of a system in several ways. The most obvious effect is cycle-to-cycle jitter (high-frequency or random effects) that causes a reduction in clock-to-clock spacing and affects logic timing as was discussed previously. Another undesirable effect is the longer term clock wander that can cause problems in a system when two parts of the system are clocked by different clocks of the same frequency but with a variable phase relationship.

Jitter can affect a Clock-Synchronizer PLL in at least two ways. First, the jitter in the PLL clock directly reduces the jitter tolerance of the Clock/Data separator. Jitter tolerance is a measure of design margin in a serial communication link. Second, the jitter of the PLL clock is transferred directly to the receiving host system clock. This jitter-reduced timing margin affects the system in exactly the same way that Multiplier PLL jitter affects logic timing.

Bit Error Rate

Jitter is often directly equated with Bit Error Rate (BER). While it is true that jitter accumulation on a serial link is the primary determinant of the link's reliability, its statistical character makes it difficult to understand and impossible to accurately predict. Since PLLs also exhibit a statistical nature which must be added to the jitter of the link, some designers incorrectly assume that the PLL jitter is what causes the errors. In fact, the PLLs used to create the high frequency clocks in the transmitter and receiver serve to increase reliability rather than decrease it.

BER is a term which is common to both serial communication devices and communication systems, and could also be applied to any system where data moves from one storage location to another. BER is the ratio of "corrupted data" received to "good data" sent. Sometimes BER can mean "Byte Error

Rate" in systems that transmit multi-bit wide information. Errors that affect the fill-bits (the "non-information" bits that occupy the time between actual data transfers) are not usually counted toward BER, but might be used to predict overall system margins.

$$BER = \frac{Bits\ in\ error}{Bits\ transmitted}$$

Usually BER is expressed as a large, negative exponent (e.g., 2.5 x 10^{-12} or 1 x 10^{-9}) because acceptable systems perform almost flawlessly. For example, a 250 Mbaud system operating with a BER of $1x10^{-12}$ would only experience about one error per hour while sending continuous information (see *Figure 12*).

While one error per hour may seem excessive, it comes naturally from the technology of the serial interconnect used. Even when adequate margins are designed into the link, there are physical and electrical effects which will cause occasional errors. Communication systems have been engineered to accept this error rate by including some level of error checking in the physical layer hardware, and extensive error recovery built into most of the communication protocols.

The PLL(s) used in the communication link don't cause the failures. Their ability to recover timing information from a severely distorted serial interface is one reason that these types of links are possible at all. The PLL supplies the logical clock necessary to correctly process the data. Without it the difficulty and cost of aligning data and clock across any distance would be immense. But because of the designer's inability to predict to the exact statistical nature of the interconnect link, and because PLL jitter contributes to this uncertainty, PLLs are often wrongly equated with BER.

HOTLink CY9266–C Evaluation boards operating at 250 Mbaud with a short coax link have been tested continuously for over 4000 hours without *any* errors. During this time, over $3.6x10^{15}$ bits were sent, received, and checked by the HOTLinks using the BIST function. This error-free time yields an estimate of BER less than $8x10^{-16}$ with greater than 95% confidence. Link error rate is not impaired by the addition of a deterministic interconnect link, such as a long coaxial cable. HOTLink CY9266–C Evaluation boards running the BIST test at 250 Mbaud, and interconnected with 300 feet of RG–59 coaxial cable (90% of the maximum uncompensated distance) have operated with no errors during a 1500 hour test. This shorter test still yields an estimate of BER less than $2x10^{-15}$ with greater than 95% confidence. No error has *ever* been recorded in all the time spent testing HOTLinks with deterministic links (sensitivity to environmental-noise injection was not included in this test). Actual BER rates have never been determined because after an uneventful six month test, the tests were terminated to free the equipment for other uses.

Example: Bits transmitted $= 250 \times 10^6 \frac{bits}{sec} \times 3600 \frac{sec}{hour} = 9 \times 10^{11} \frac{bits}{hour}$

$$BER = \frac{N}{9 \times 10^{11}} = N \times 1.11 \times 10^{-12}$$

MTBF of a link running at 250 Mbaud @ BER of 1×10^{-12} = *1 hour between errors*

MTBF of a link running at 250 Mbaud @ BER of 1×10^{-15} = *46 days between errors*

MTBF of a link running at 250 Mbaud @ BER of 1×10^{-18} = *127 years between errors*

Figure 12. BER Example Calculations

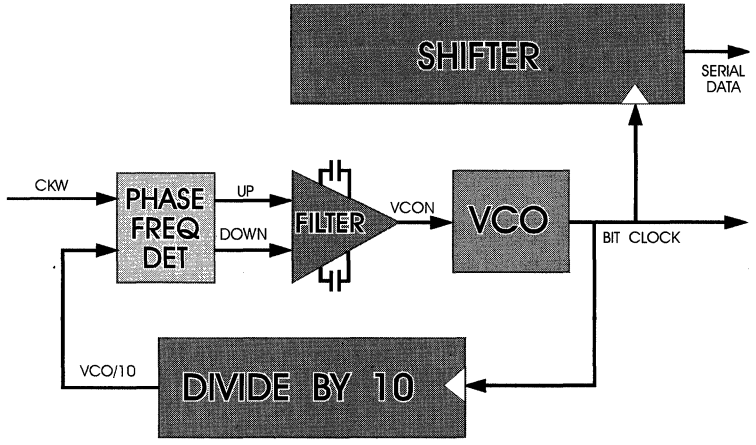


Figure 13. HOTLink Transmitter PLL Block Diagram

HOTLink Transmitter Jitter

The PLL used in a Transmitter application (clock multiplier) is intended to provide a high-speed, stable clock that tracks a low-speed reference (CKW in *Figure 13*). This clock (Bit Clock) is used to run the parallel to serial converter and all of the internal logic in the HOTLink Transmitter.

Jitter is an undesirable and often unpredictable misplacement of any particular transition from its ideal position. Transmitter output jitter can be characterized as Random or Deterministic Jitter, RJ and DJ respectively. It can further be subdivided into Intrinsic Jitter, Transferred Jitter, or Injected Jitter.

To separate the various types of jitter, carefully designed tests were performed on HOTLink parts selected from the full spectrum of manufacturing tolerances. Tests were designed to separate the effects of Power Supply, Clock Sources, and various PLL characteristics. Manufacturing tolerances include variations in all types of resistors used in the design, characteristics of Bipolar and CMOS transistors, and other normal process variations. Environmental effects include V_{CC} variation over at least the full specified range, and ambient temperature variation over the full military and commercial ranges.

Unless otherwise noted in the following text, static variations in power supply levels (4.5V to 5.5V), ambient temperature (−55°C to 125°C), and process variations (within manufacturing tolerance limits) cause virtually no change (within the accuracy of the measurement system) to any jitter tolerance or PLL characteristic. This should be true for any *well-designed* PLL, though it is often not true for all products in the marketplace.

Transmitter Random Jitter

Random Jitter is an undesirable and unpredictable misplacement of any particular transition from its ideal position that cannot be correlated to either data-stream content, or parameters of the hardware. To separate Random Jitter from the other effects, the HOTLink is configured to send various square-wave patterns. This minimizes any possible deterministic (DJ) effects caused by loading or variations in internal circuit delays. The set-up used to measure HOTLink Random Jitter is shown in *Figure 14*. The clock source was a combination of an HP 8656B and HP 8131 generator chosen to give the lowest possible input jitter. Other generators, described later in this application note are also satisfactory for this clock source. The output was measured on both an HP 54720D and a Tektronix 11801 Digital Sampling Scope, each of which have sufficient sampling bandwidth to accurately show the performance characteristics of the device under test.

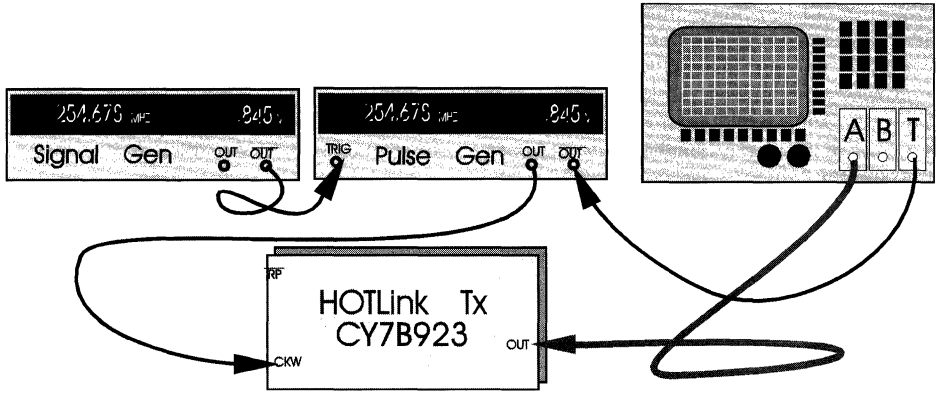
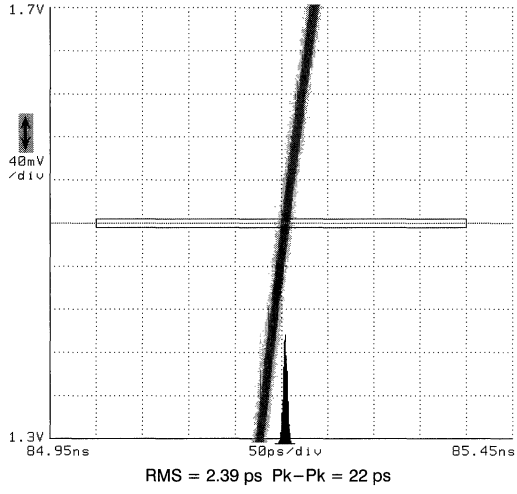


Figure 14. HOTLink Transmitter Random Jitter Set-Up

Random jitter measured in this way is shown in *Figure 15*. In virtual time, the "random" characteristic can be seen in these histograms taken on a Tektronix 11801 Digital Sampling Oscilloscope.

The left histogram in *Figure 15* shows the quality of the signal used to gather the data that follows. The input clock coming from an HP8656B frequency synthesizer, buffered through an HP8131 Pulse generator, has minimal jitter. This TTL input clock source adds a negligible amount of jitter to the output jitter measurements that follow. Other generator/buffer combinations are possible, but if there is any appreciable jitter at this point, it will be impossible to separate the input jitter from the jitter accumulated in the part under test. Other tests (described later) will show the effect of noise at the CKW input.

Cycle-to-cycle jitter out of the HOTLink while sending a "perfect" bit-rate square wave, measured one bit away from the output edge used to trigger the scope, is very small. This is a measure of the jitter that can accumulate between adjacent VCO clocks



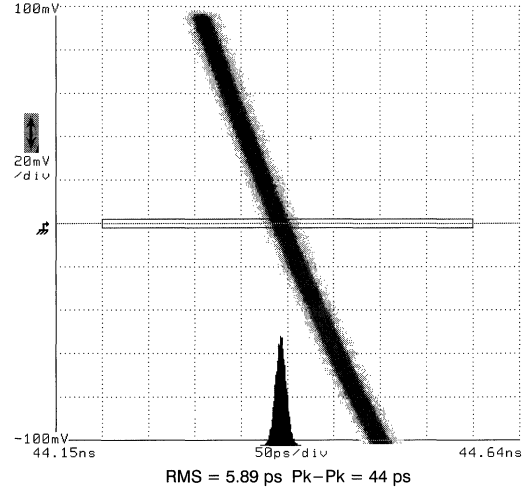
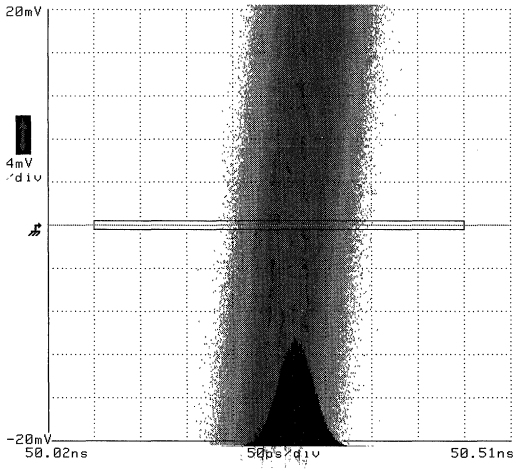


Figure 15. Histograms of CKW Source and OUTA± One Cycle Away

in the multiplier PLL. Since this measurement (right histogram in *Figure 15*) was triggered and measured on the same output, and the scope was not constrained to sample at any particular rate, this photo shows the superposition of all possible bit positions. It also shows the (small) magnitude of Deterministic jitter built into the output circuitry.

The serial outputs of HOTLink are PECL differential signals that must be combined differentially in the front end of the sampling scope to provide an accurate measurement of both the signal transition and any jitter present on those transitions. *Figure 16* displays the differential measurement of the accumulated jitter on the OUTA± outputs of the HOTLink Transmitter. This is a measure of the total jitter accumulation through the entire PLL and output circuit while sending a "perfect" byte-rate square wave.

The wide vertical bar shows the accumulated jitter measured in 100,000 samples of the 0-to-1 transition while being triggered by the CKW reference. The different shades of gray in the vertical bar represent different concentrations of signal samples that occurred at that specific time/amplitude coordinate. The darker the sample point, the more samples that occurred at that point. The very center of *Figure 16* contains a narrow rectangle centered around the



RMS = 19.13 ps Pk–Pk = 164 ps

Figure 16. HOTLink Transmitter OUTA± Rj vs. CKW

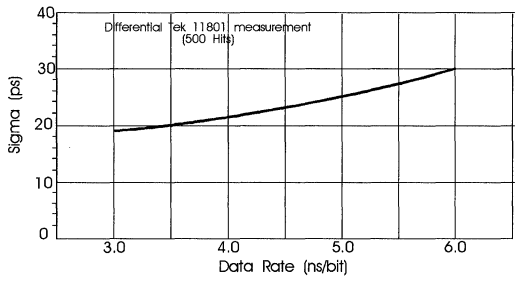


Figure 17. HOTLink Transmitter Random Jitter as a Function of Frequency

HOTLink Receiver threshold region. All samples that occur within this rectangle are plotted in the black histogram at the bottom of the figure. The Gaussian shape of this curve confirms that the jitter is truly random in nature.

While there is a slight increase in output jitter as the operating frequency decreases (see *Figure 17*), there is no appreciable change in HOTLink jitter due to V_{CC}, temperature or process variation.

In contrast to the virtual-time measurements illustrated in the previous figures, real-time measurements allow an insight to the behavior of the HOTLink Transmitter in terms of sequential events. *Figure 18* shows the edge displacement from the ideal location of all sequential rising edges of a continuous data stream. In this sequential, real-time measurement, it can be seen that there is no obvious or repetitive pattern to the jitter, confirming the validity of the virtual-time measurement. A minor pattern is visible in the running-average histogram, that shows a small amplitude, continuous oscillation in the sign of the edge misplacements. Peak-to-peak deviation in the real-time illustration is smaller than that indicated in the virtual-time measurement. This is consistent with the large difference in the number of samples in each, and the fact that many of the extreme excursions that occupy the tails of the distribution may not be PLL variations, but are probably caused by pulse-noise injected into some logical or measurement function.

The pattern of jitter does not change appreciably when the output pattern changes from one cycle-per-byte to one cycle-per-bit (see *Figure 19*). The peak excursion remains about the same, and the distribution is similar.

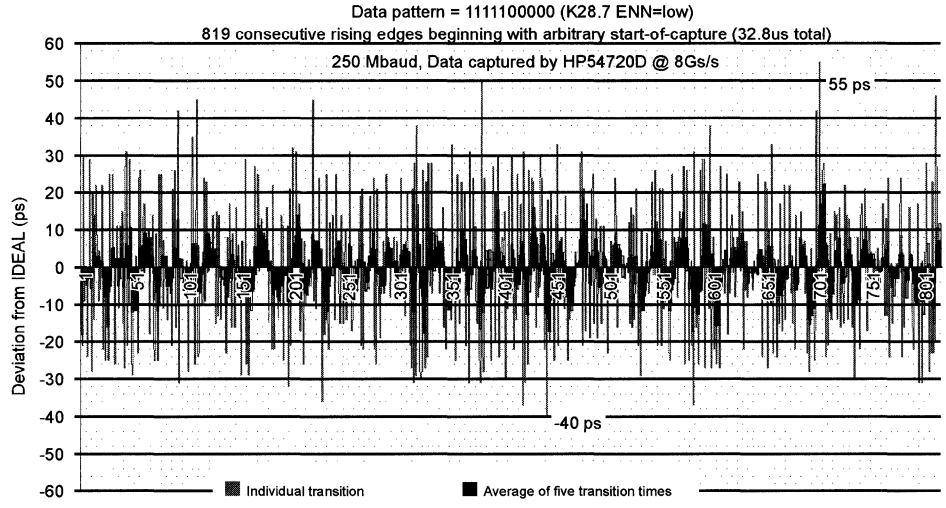


Figure 18. Real-Time HOTLink Transmitter-Output Byte-Rate Jitter

Transmitter Deterministic Jitter

Deterministic jitter is an undesirable and often difficult-to-predict misplacement of any particular transition from its ideal position that can be correlated to the content of the data stream or some characteristic of the circuit or hardware. To measure deterministic jitter attributable to the internal circuitry of the HOTLink Transmitter, the parts were measured in several ways to separate the different possible DJ sources. (For all of the following DJ measurements, repetitive output patterns are averaged to remove RJ effects.)

The basic Deterministic Jitter tests fall into two categories:

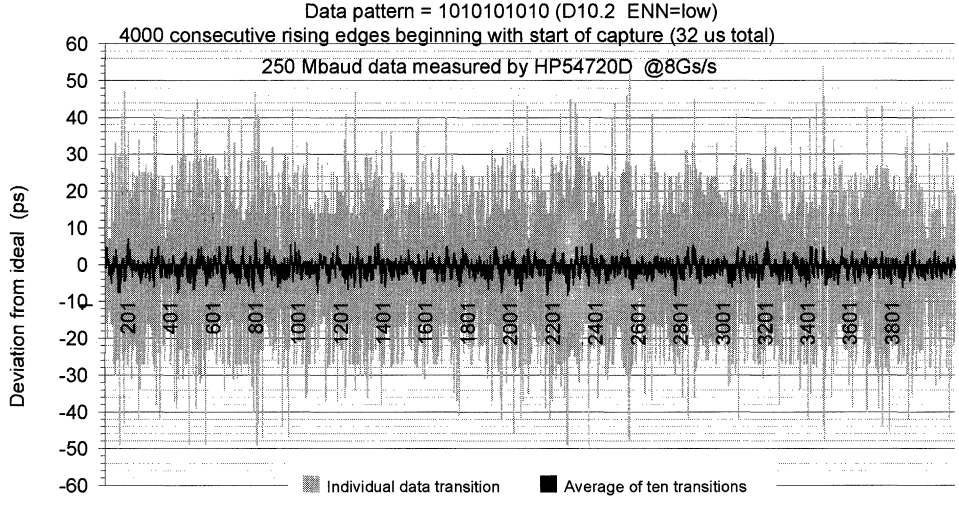


Figure 19. Real-Time HOTLink Transmitter Bit-Rate Jitter Output

1. There could be DJ that causes edges to be misplaced because of variations of internal rising and falling delay or variations of internal delays caused by the spacing between adjacent transitions (e.g., internal logic swing limitations or flip-flop metastable-delay effects).
2. There could be DJ that is caused by some deterministic PLL-multiplier effect which misplaces the internal clock edges from their ideal transition time (e.g., PLL phase corrections might cause bit-0 to be always early, bit-2 always late, and the others on time).

Deterministic Jitter as a Function of Data Pattern

To check for internal delay effects and sensitivity to data content, the HOTLink transmitter was configured to send various data patterns. *Figure 20* shows how the bit position is affected by pulsewidth and pattern-content of the serial bit-stream. This test is performed by measuring the exact timing of a single data transition as other transitions occur at various other bit positions and pulse spacings. By plotting the exact transition time against the expected transition-time, variations in internal delay become apparent. This is a test of the output buffer and circuits in the serial output logic, because it checks the position of a single transition as the pulse width preceding it changes from nine bits to one bit LOW.

HOTLink manages to place any particular edge within ±10 ps of the ideal location regardless of data pattern. In this test it is impossible to separate pulse distortion from slight propagation delay changes. The 50% duty cycle pattern was used as the reference location for the "ideal" pulse position. The absolute position of the rising edge was measured as the ten-bit data pattern was stepped through all nine of the possible pulse widths. Edge placement is not significantly affected by temperature or process variation. The change in edge displacement in the figure above is mostly caused by V_{CC} variation and may be related to output load currents variation.

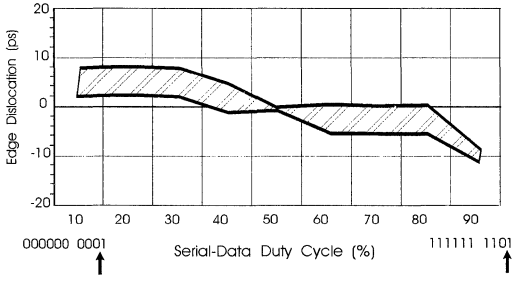


Figure 20. Data Dependent Edge Displacement

Deterministic Jitter Caused By PLL Corrections

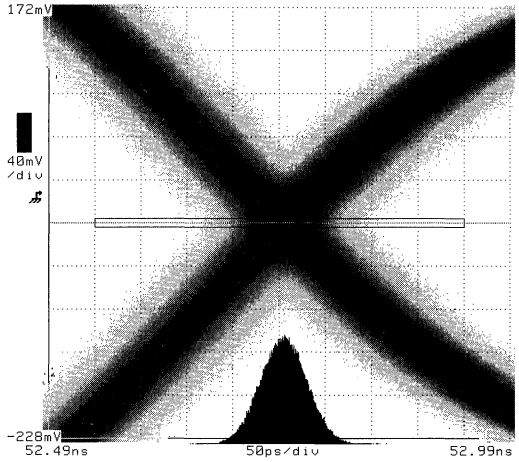
By causing the Transmitter to send an alternating 1-0-1-0... bit pattern and synchronizing the measurement system to the byte rate clock, it is possible to detect misplacement of internal clock transitions. In this test, performed by measuring the exact time of each voltage transition (averaged to remove random jitter effects), and plotting it against its expected transition-time, PLL phase corrections and clock-synchronous noise will appear as fixed, repetitive displacements from the ideal position. The square-wave bit pattern minimizes delay effects which might otherwise be present.

The results of this evaluation showed that there were no deviations observable within the measurement accuracy of the test equipment. Extensive testing showed less than 2 ps deviation from the ideal position of all ten of the output transitions, regardless of frequency.

Total Transmitter Jitter While Sending BIST

As a measure of total transmitter jitter, the tests were repeated while sending the built-in self-test (BIST) sequence. In contrast to the special patterns which were used to measure the various components of overall jitter, this test is a more comprehensive measure of real HOTLink performance, since it would highlight any effect that might have been obscured by the individual jitter-component tests.

The scope photo in *Figure 21* shows the jitter characteristics of the HOTLink output while sending the BIST pattern. This 511-byte pseudo-random pattern includes all of the legal data patterns in the 8B/10B code, and is a good representation of a typical data transmission. The resulting jitter includes all of the random and deterministic jitter accumulated by the clock source, the multiplying PLL, and the internal logic and output circuits.



RMS = 25.1 ps Pk−Pk = 228 ps

Figure 21. HOTLink Transmitter OUTA± Total Jitter in BIST vs. Bit Rate Reference

Transmitter Jitter Transfer Function

PLL output jitter can be affected by the noise characteristics and stability of the clock source used as its reference. The closed-loop transfer function of this type of PLL is a low-pass filter. Noise components below the rolloff frequency (f_{-3dB}) of the PLL will be passed unattenuated and those above f_{-3dB} will be attenuated. By injecting a measurable and controlled amount of noise (jitter) into the clock reference as shown in *Figure 22*, the PLL transfer characteristic can be measured.

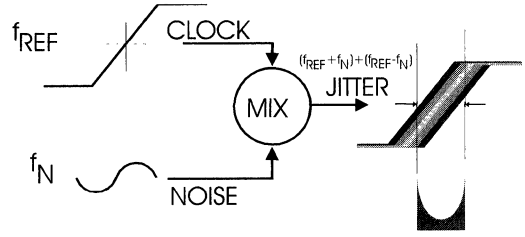


Figure 22. Clock-Jitter Generated by Mixing Noise into CKW

In this configuration (*Figure 22*), the noise source is added to the clock source with a resistive mixer. Because the clock source has a significant ramp rate, the addition of the noise will cause a controlled variation in the effective threshold crossing, thus causing jitter. The noise source can be any controlled source, but for this test, it was a good quality sinewave generated by a stable generator. The amplitude was adjusted to create the desired CKW jitter amplitude (ns Pk−Pk), and the frequency was varied over a wide range while the output jitter was monitored on OUTA±. The graph in *Figure 23* shows the relationship between input and output jitter at various input jitter frequencies as the jitter frequency is increased from about 10 kHz to over 70 MHz.

While the vertical axis of this type of chart usually is expressed as a "gain" term, and uses units of dB (i.e., 20 log out/in), this data is being presented as a pure ratio of input to output jitter. This allows a clearer visualization of jitter magnitude, and shows that for

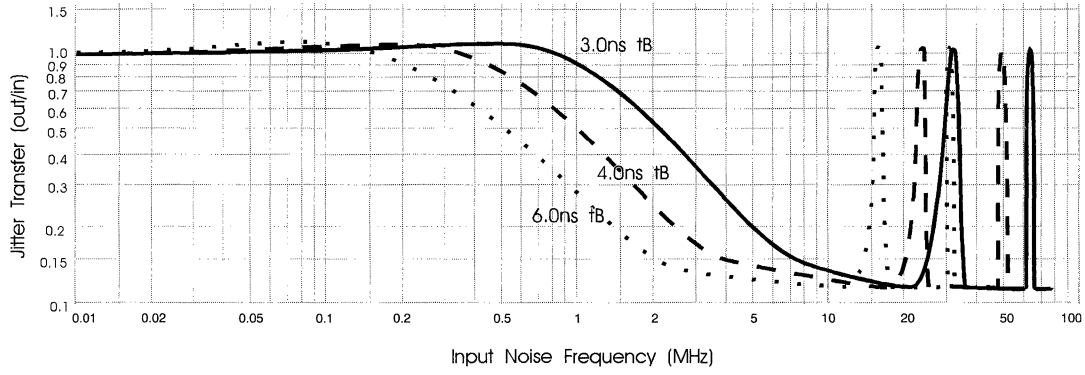


Figure 23. HOTLink Transmitter Jitter Transfer Function

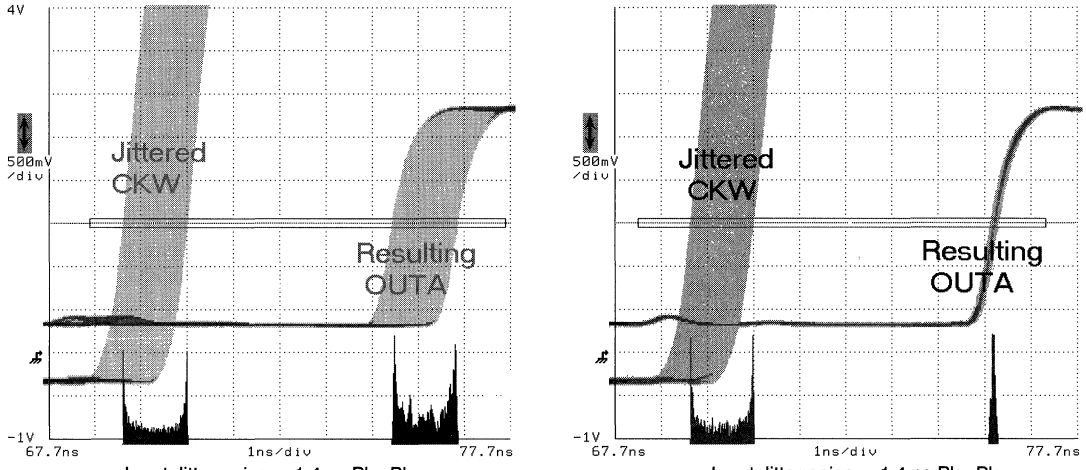


Figure 24. Serial Output Jitter Varies as a Function of Input Noise Frequency

all frequencies of operation and noise, the output jitter falls to approximately the same "noise-floor." By maintaining the vertical log scale, it is obvious that the effect being illustrated, is the same as the closed-loop PLL transfer function described earlier.

As expected, low-frequency jitter passes through the PLL unattenuated. Higher frequencies are attenuated until the jitter frequency approaches the Reference-Clock frequency. This jitter-feed-through peak at about the reference frequency is the result of the sum-and-difference frequencies that naturally result from mixing. A significant frequency component is generated at the "difference" between CLOCK (f_{REF}) and NOISE (f_N) sources. When this "difference" frequency falls within the PLL filter bandwidth, it passes unattenuated to the output of the PLL and appears as jitter exactly as if it was caused by an equivalent low-frequency input-noise source. This effect was enhanced by using a single frequency noise source. The energy at any particular frequency of a wide-band noise source is relatively small, but would feed through in exactly the same way. Narrow-band noise sources that operate synchronously with the HOTLink CKW input rate might cause more of these "mixed-down" frequency components that would also feed through and emerge as output jitter.

The bandwidth of the HOTLink Transmitter PLL varies slightly as a function of the operating frequency as is shown in *Figure 23*. (tB = 1/baud; tB = 6 ns = 160 Mbaud, etc.) This variation is attributable to variations in VCO gain that are a function of operating rate.

The scope traces in *Figure 24* graphically show the jitter feed through from the TTL-CKW input to the PECL outputs of the HOTLink Transmitter. Low jitter frequencies pass through unattenuated, and high frequencies are significantly attenuated. For high-frequency input jitter, the only jitter remaining on the output is roughly equivalent to the intrinsic jitter of an undisturbed PLL.

V_{CC} Jitter Transfer Function

To characterize HOTLink output jitter characteristics in the presence of noise carried on the power supply, the Random Jitter set-up was modified as shown in *Figure 25*.

In this test the power supply was intentionally disturbed. By injecting a measured amount of noise into the V_{CC} pins of the HOTLink Transmitter (using an external driver), the jitter effects of power supply noise could be observed.

As expected, when the power supply is disturbed, the output will contain some additional jitter. In-

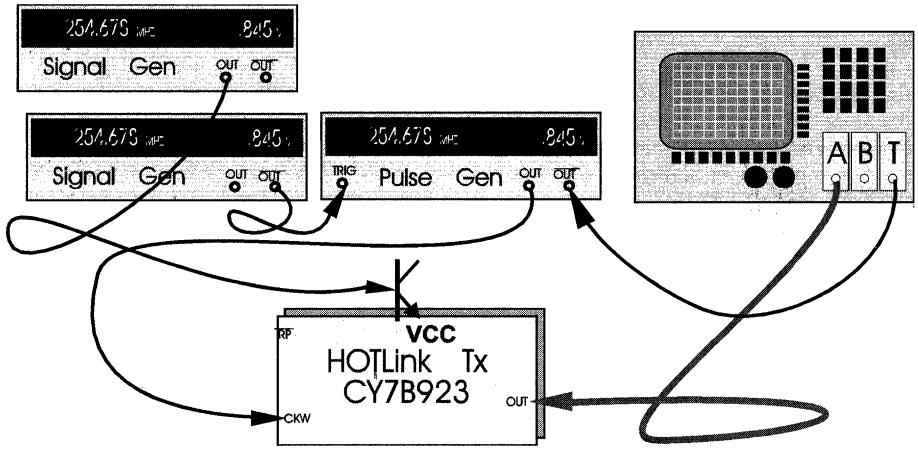


Figure 25. HOTLink Transmitter V_{CC} Coupled Jitter Set-Up

creasing amplitude disturbances cause increased jitter amplitudes. It would normally be difficult to create this much noise on normal system boards because of the normal power supply bypassing that is usually applied to this type of component. Large amplitude V_{CC} spikes are removed by the bypass capacitors.

As the V_{CC} noise frequency is varied across the various frequencies, the jitter out also varies. At low frequencies of noise, the jitter is small, and at high noise frequencies the jitter is also small. At about the PLL roll-off frequency (as measured in the previous analysis), the jitter output increases (see *Figure 26*).

Transmitter PLL Lock Time

Multiplying PLL lock characteristics are mostly a function of the internal loop dynamic characteristics. While the loop is changing the clock period

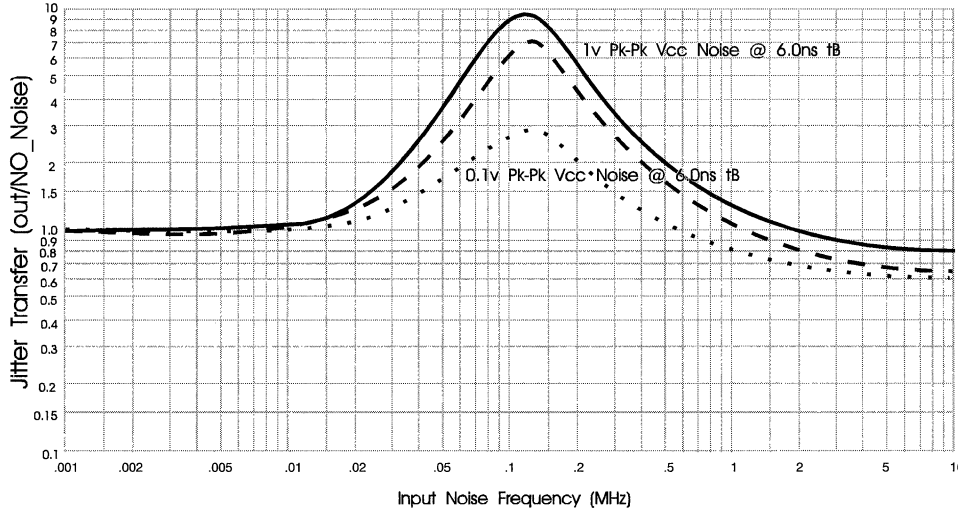


Figure 26. HOTLink Transmitter V_{CC} Coupled Jitter Transfer Characteristic

from one frequency to another, the transition is not monotonic as might be expected, but appears to oscillate about the acquisition-trend line until the frequency falls within the PLL "Lock Range." This effect is a result of the normal characteristics of the Phase Frequency Detector (PFD) and the Loop filter. The PFD output is a continuously varying series of pulses that cause the VCO to change frequency in the desired direction, but the resulting pulse widths are not constant. The graph in *Figure 27* shows each of the Bit-Clock periods during the PLL "re-lock" progression while the loop locks to a higher and a lower frequency.

The Bit-Clock period listed in *Figures 27* and *28* is the period of the internal VCO inside the HOTLink Transmitter. While this signal is not directly observable at a HOTLink Transmitter pin, its period may be directly calculated from measurements made at the serial output (OUTA±) pins. In this test the serial data pattern was set to the equivalent of a K28.7 special code (0000011111), fixing the bit-clock period at one tenth the interval between output rising transitions.

The two locked frequencies were selected to be the minimum and maximum actual operational limits of the part under test. Cycle-by-cycle times were recorded in a continuous stream using an HP54720D. The frequencies used for these illustrations are well outside the datasheet operational limits of HOTLink, but were the actual functional limits of this particular (typical) part. While all parts behave in a similar manner, some have slightly higher or lower operational frequency limits. The acquisition rates vary slightly with temperature, being slowest at higher temperatures.

The PLL behavior is slightly different when the reference clock has been absent for some time (see *Figure 28*). Instead of immediately beginning to acquire the new frequency, there is a time after CKW begins but before there is any change in the VCO frequency. This time is required for the internal control nodes to move from their "ranged-out" levels (resulting from the PLL trying to track to zero or infinity Hz) to within the compliance-limits of the amplifier and VCO. After the change begins, it moves at the same rate as for the previously described cases.

When CKW is removed, the loop immediately begins to slew toward the lowest possible speed. The transition does not have the "jaggies" typical of a loop tracking to another clock rate, because there are no reference clock edges to modulate the PFD output. The frequencies shown in this illustration are well outside the data sheet operational limits of HOTLink, but were the actual functional behavior of this particular (typical) part. While all parts behave in a similar manner, some have slightly higher or lower frequency limits. The acquisition and slew-rates were similar but vary slightly with temperature, being slower at higher temperatures.

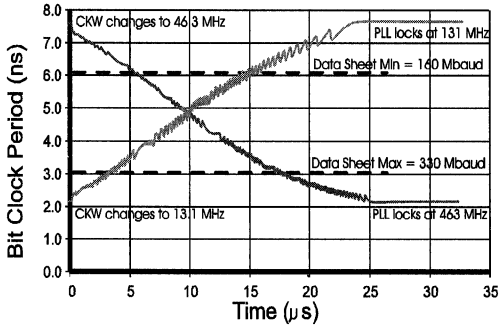


Figure 27. Transmitter PLL Acquisition Characteristic (from Locked to Locked)

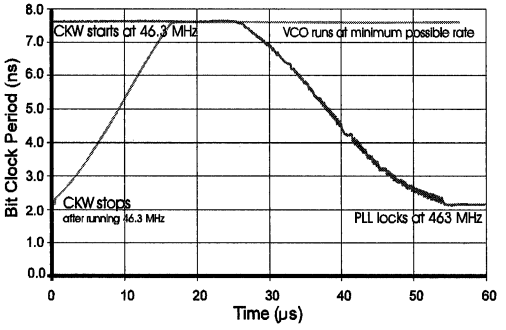


Figure 28. Transmitter PLL Time to Lock (Quiet to Locked)

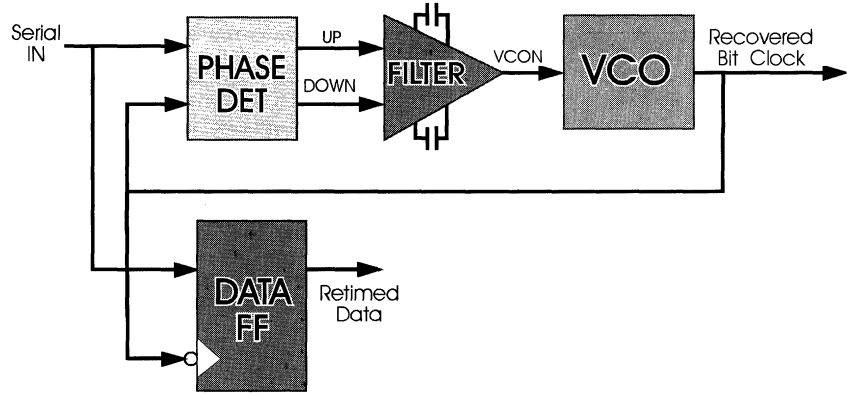


Figure 29. HOTLink Receiver PLL Block Diagram

HOTLink Receiver Jitter

The PLL used to synchronize an internal clock to a received bit stream (i.e., in the HOTLink Receiver) has different requirements than those for a multiplying PLL. This loop is effectively a one-to-one loop where the bit clock (Received Bit Clock, an internal signal) runs at the same rate as the incoming data stream (Serial IN, an external signal). The Received Bit Clock is used to sample the Serial input at regular intervals, thus extracting the serial data (Retimed Data, in *Figure 29*). This same signal runs all of the internal logic for deserializing, framing, and decoding the serial data. Any disturbance that can affect the PLL and the Recovered Bit Clock will affect both the quality of the data recovery and the quality of the byte-rate, data-synchronous clock that is provided to the receiving system.

Receiver jitter affects systems in at least two ways. Jitter tolerance is a major determinant of system margin, and Jitter feed-through can reduce timing margins in the receiving host system.

Jitter feed-through is a function of the PLL filter characteristics, and can be directly measured at the CKR output of the HOTLink Receiver in much the same way used to test Transmitter jitter feed-through.

Jitter tolerance is more complicated, since it is a measure of the Receiver's ability to correctly capture and interpret incoming data, and must be measured indirectly. Jitter tolerance is both a function of the intrinsic jitter in the receive-clock synchronization PLL and the effects of received data upon it. Tolerance is also a function of the precision-timing and alignment of internal clock edges (i.e., the clock edge used in the PLL to synchronize the data, and the clock edge used to sample the incoming data stream). The data-sampling flip-flop set-up/hold timing characteristics and their variation contribute to further jitter tolerance degradation.

To isolate the effects and tolerance limits to various types of jitter, carefully designed tests were performed on HOTLink parts selected from the full spectrum of manufacturing variation. These tests were designed to separate the effects of power supply, data characteristics, external clock sources, and various PLL characteristics. Unless otherwise noted, static variations in power supply levels (4.5V to 5.5V), ambient temperature (−55°C to 125°C), and process variations (within manufacturing tolerance limits) cause virtually no change (within the accuracy of the measurement system) to any of the following jitter tolerance or PLL characteristics.

Static Alignment and Error-Free Window

To maximize jitter tolerance, the receive circuit is designed to sample the incoming data at a point *exactly* half way between the ideal transition times of uncorrupted data. This requires that the PLL track the incoming data and align itself with the "average timing" of the received edges. The precision of this

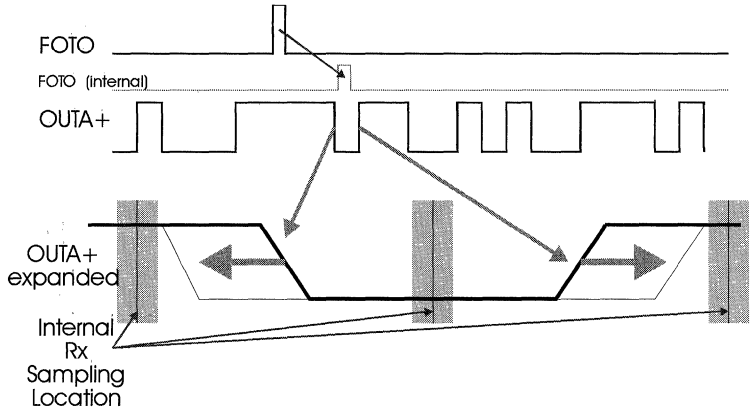


Figure 30. Technique to Measure Static Alignment

alignment is often called "Static Alignment" and should have a magnitude of zero, indicating perfect alignment of VCO and the data and perfect 50% sampling alignment. Using this recovered clock, the incoming data is sampled at the point that gives maximum tolerance to misplaced edges and maximizes the error-free window. Any misplacement of this sampling point will reduce jitter tolerance.

Static alignment of the HOTLink Receiver was evaluated using the technique shown in *Figure 30*. The HOTLink Transmitter and the Receiver under test were configured to send and receive the BIST pattern. Then, by inserting a BIST-synchronous pulse on the FOTO pin (using a generator triggered on the $\overline{RP}$ output of the HOTLink Transmitter), one transition in the transmitted data pattern was varied to find the maximum "misalignment" possible before the onset of an RVS error indication. This configuration allows the receive PLL to have about 3000 "ideal" transitions (i.e., the total number of transitions in the 511 byte BIST loop) and only one misplaced edge. Shorter patterns modified in this way (e.g., a single data byte with byte-synchronized FOTO pulses having a single misplaced transition) give an erroneous result. The very large phase error which occurs in one of the ten bit positions will be averaged out by small-compensating phase-adjustments during the other nine bit-times. The BIST pattern test allows the PLL phase-correction response from the single-edge error to settle out before the next error appears so that the averaging effect does not color the data-capture results.

Data transitions can be misplaced from their ideal position by almost half of a bit-time without erroneous sampling by the data recovery flip-flop. The data characterization summary in *Figure 31* indicates that the HOTLink Receiver will accept misplaced edges to within about 250 ps of the half-bit point. The center of the small error region where data is not sampled correctly (at approximately 180 ps after the ideal mid-bit point) is the actual PLL static alignment position. The width of the error region (about 150 ps) is attributable to both the sampling flip-flop metastable region, and the internal PLL clock jitter.

This data alone implies that any data edge could fall anywhere within a bit time (minus about 500 ps) and still be decoded correctly. This is almost correct, except for the effect of receiver clock jitter caused by the various types of incoming jitter.

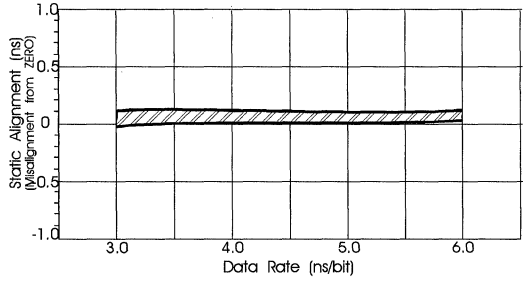


Figure 31. HOTLink Receiver Static Alignment as a Function of Frequency

Duty Cycle Distortion Jitter Tolerance

The characteristics of some types of interconnect circuits cause Duty Cycle Distortion which the receive system must tolerate. DCD jitter alters the placement of all transitions in the data stream by about the same amount (in alternating directions) regardless of the bit pattern being sent. For small amounts of jitter, this alternating error tends to cancel out, and the loop behaves normally while recovering data without error.

As the magnitude of jitter increases, phase correction pulses from adjacent misplaced edges will begin to interact. Each correction pulse has some finite duration, usually a significant percentage of the expected bit time, and is proportional to the magnitude of the edge misplacement. Since jitter is also expressed as a percentage of a bit (usually a large percentage) the interaction between jitter magnitude and phase correction pulse width will determine DCD jitter tolerance. When adjacent phase corrections interact, they sum in unexpected ways which affect the resulting correction response. When these interactions are rare or small, there is no apparent effect. If the interactions affect most of the phase correction events, the PLL stability, predictability, and output jitter will be affected and data will not be captured correctly.

Figure 32 shows HOTLink Receiver DCD jitter tolerance. This test was performed by carefully corrupting the link between a HOTLink Transmitter and Receiver with increasing magnitudes of DCD (See Jitter Generator circuit and description *Figure 49*). Using the BIST test capability included in the chips, DCD tolerance limits were declared to have been exceeded when the RVS output of the Receiver indicated approximately one error every ten seconds (i.e., BER$\leq\approx 4x10^{-10}$ at 250 Mbaud). Slight differences in jitter tolerance were found between parts from different process corners, but no appreciable variation was found for V_{CC} or temperature variation. The DCD tolerance characterization data shown above varies by less than 5 percentage points across the full process spread (e.g., from 1.42 ns to 1.39 ns out of a 3.0 ns bit time). The threshold of failure is very abrupt. At the jitter levels shown in *Figure 32*, changes in jitter amplitude of less than ±100 ps make the difference between almost-perfect data reception, and almost-total corruption.

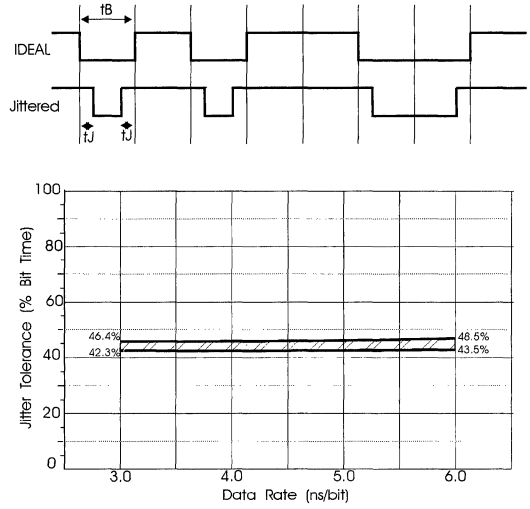


Figure 32. Duty-Cycle-Distortion Jitter Tolerance as a Function of Data Rate

In contrast to the predicted jitter tolerance that comes from the Static Alignment test, and the DDJ tolerance (see following text), DCD tolerance at first appears to be much smaller. This apparent reduction in jitter tolerance is entirely due to PLL and Phase-Detector effects, and do not result from any anomaly in the data recovery path. The data can be recovered correctly at the levels of edge misplacement that are found at the limits of DCD tolerance but not above. By carefully approaching the limit, it can be seen that the PLL loses lock at the jitter magnitudes shown in *Figure 32*, and then regains it at slightly higher jitter levels, but with a massive clock jitter, often slipping bits as the jitter goes through the "magic point," destroying any data recovery possibility. The recovered clock shows almost no jitter feed-through when DCD is present and remains below the "data-corruption" threshold, as will be shown later (*Figure 46*).

Fortunately, most transmission links don't include large amounts of DCD. The most common contributors are mismatched output loads on differential or single-ended PECL outputs, and improperly designed or operated optical interface modules. Single-ended PECL outputs can change the effective delay of the driver by about ±0.5 ns. Differen-

tial outputs are typically more symmetrical. Optical-to-electrical (receiver) interface modules running with extremely high or low light levels can have non-linear and asymmetrical delay characteristics that affect the pulse symmetry of the received output used by the PLL data recovery circuits. The optical emitter in an electrical-to-optical interface module also has non-symmetrical turn-on and turn-off characteristics which are normally compensated by careful design of the drive electronics. At the limits of performance, optical modules can add more than ±1 ns of DCD.

Data Dependent Jitter Tolerance

The characteristics of some types of interconnect circuits cause Data Dependent Jitter which the receive system must tolerate. The same "correction-pulse" interaction that limits DCD tolerance also affects DDJ tolerance. Since the collisions between adjacent correction pulses occur at a much less frequent and regular rate, the effect is smaller. The "clock-jitter" that results from these corrupted corrections reduces the jitter tolerance to less than the ideal maximum that the Static Alignment test might predict.

Figure 33 shows HOTLink Receiver DDJ jitter tolerance where the DDJ was generated by an artificial

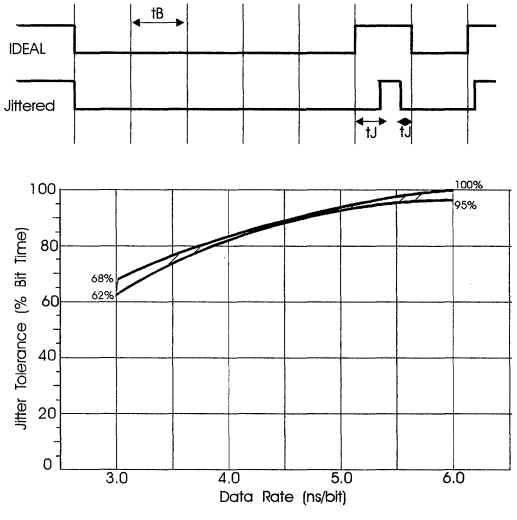


Figure 33. Data-Dependent-Jitter Tolerance as a Function of Data Rate

generator. This test was performed by carefully corrupting the link between a HOTLink Transmitter and Receiver with increasing magnitudes of DDJ (see Jitter Generator circuit and description in *Figure 50*) while sending a continuous BIST pattern. Errors were most typically associated with the long running bit pattern included in a K28.5 bit pattern, and the same tolerance was observed while receiving *only* corrupted K28.5s. The worst DDJ peak always follows the 1111101 and the 0000010 contained in the special characters. Using the BIST test capability included in the HOTLinks, DDJ tolerance limits were declared to have been exceeded when the RVS output of the receiver indicated approximately one error every ten seconds (i.e., BER $4x10^{-10}$ at 250 Mbaud). Slight differences in jitter tolerance were found between parts from different process corners, but no appreciable variation was found for V_{CC} or temperature variation. The DDJ tolerance characterization data as shown in *Figure 33* varies by less than 5 percentage points across the full process spread (e.g., from 2.04 ns to 1.86 ns out of a 3.0 ns bit time). The threshold of failure is very abrupt. At the jitter levels shown above, changes in jitter magnitude of less than ±100 ps make the difference between almost-perfect data reception, and almost-total corruption.

Interconnect Link Jitter Tolerance

The tolerance to synthetic-DDJ shown in *Figure 33* is slightly worse than that found when the jitter is natural-DDJ. The variation is caused by unintentional DCD introduced by the test system used to create a stable and repeatable test pattern at all frequencies over which HOTLink might operate. Wire transmission line jitter is dominated by DDJ caused by the variation in attenuation as a function of frequency. Higher frequencies are attenuated more than lower ones. This rising attenuation-with-frequency characteristic of wire links causes the wider pulses (i.e., multi-bit one or zero strings) to have a higher amplitude than the shorter pulses since the higher frequencies (those attenuated the most) are required to make the fast edges and narrow pulses, while the wider pulses contain more low-frequency components. This variation in amplitude results in variations in pulse placement, since the edge rate is almost constant and the variation in amplitude causes variations in the time at which a transition will cross the receiver threshold.

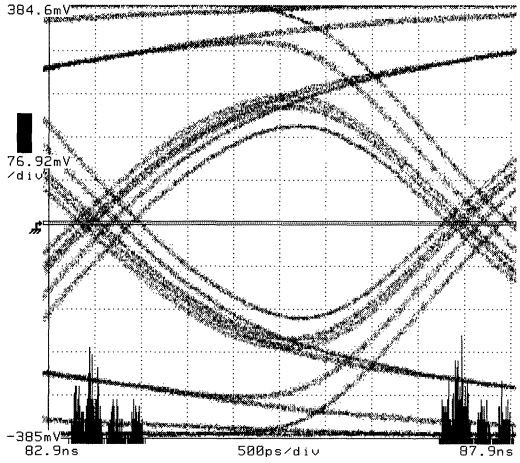


Figure 34. DDJ Characteristic of K28.5 at 250 Mbaud after 250 ft. RG−59

This effect is most visible when a single, worst-case data byte is measured. *Figure 34* shows the edge misplacement caused by the different-length pulses in a continuous K28.5 pattern (i.e., 11000001010 00111110101...). When the data is more normally distributed, it becomes more difficult to see the distinct pulse positions, and the jitter just merges into a continuous "uncertainty-zone" (see *Figure 35*).

Using actual data and real transmission lines, the HOTLink tolerance to DDJ appears to be a more constant function of bit rate than *Figure 33* shows. If about 500 ps of clear eye-opening can be maintained, the data will be recovered correctly, regardless of the data rate. However, recovered clock jitter increases with increased DDJ (see *Figure 47*).

In wire transmission links, the accumulation of DDJ determines the maximum distance over which data can be reliably communicated. The characteristics of the chosen media determines the useable distance. The total attenuation of the line is rarely sufficient to limit the maximum useable distance, even though the data bits that are incorrectly interpreted will have minimal amplitude at the time of the error. This loss of amplitude is a result of the variation in peak voltage attained during any particular pulse.

HOTLinks have been designed to offer more than 20 dB of attenuation margin between the transmitter output and the receiver input. Typical maximum-distance links have less than 10 dB of high frequency attenuation due to the transmission line and interconnect components. The remainder of the interconnect budget can be used to compensate for the difference between high and low frequency attenuation of the wire transmission line. Compensated wire links have been built that operate reliably over more than double the distances shown in *Figure 36*.

Fiber optic links, in contrast to the wire links described above, are limited by optical attenuation, chromatic dispersion, and the resulting Random Jitter in the optical-electrical converter. At the limit of operational optical margins, the low light levels into the receiver and the dispersion from the fiber combine to create misplaced data transitions. These displacements are usually random, but in the case of some optical modules, can also include significant Duty Cycle Distortion.

Peak random jitter tolerance *should* be approximately the same as the Static-Alignment limits described above (*Figures 30* and *31*). The simplest way to generate random jitter involves a long piece of fiber optic cable, and appropriate fiber optic interface modules. As fiber length increases, adding chromatic dispersion (i.e., pulse distortion caused by the variations in propagation delay through the fiber, as

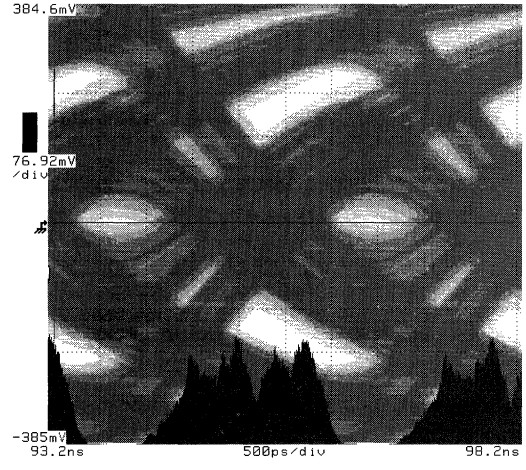


Figure 35. BIST data at 370 Mbaud after 250 ft. of RG−59 coax (BER<4.5x10^{-11} with <700 ps eye opening)

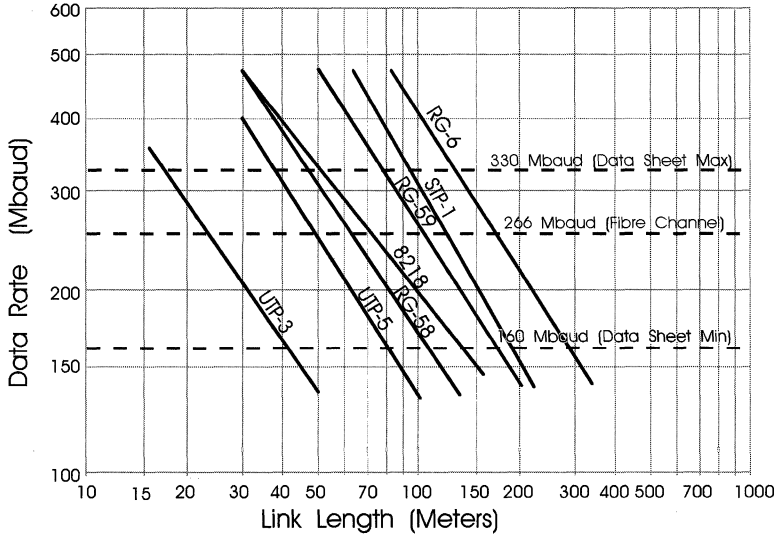
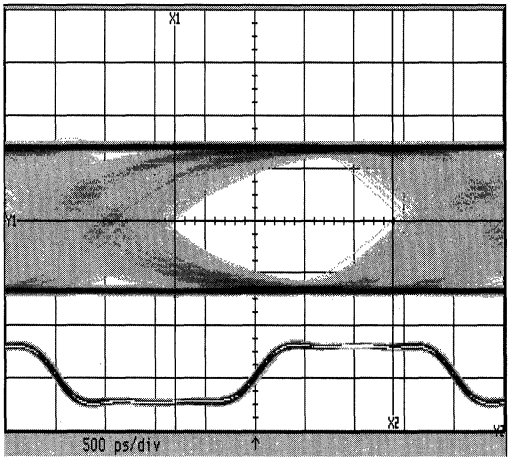


Figure 36. Maximum Data Rate vs. Uncompensated Wire Length (BER < 3 x 10^{-12})

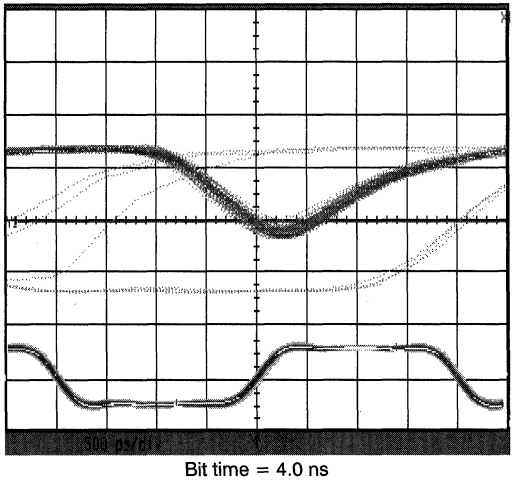
a function of optical wave-length) and attenuation, the jitter out of the optical-to-electrical converter will increase. There is a limit to attenuation, beyond which the fiber optic receiver cannot recover the data correctly. Attenuation alone, without the effects of long fiber optic cable, often causes significant DCD in the link. This DCD will obscure the real random jitter behavior of the receiving PLL.

The random jitter output of a 5-km piece of 62.5 multi-mode fiber is shown in *Figures 37* and *38* and



Bit time = 4.0 ns
Eye opening = 2.185 ns (apparently)
BER = 1 x 10^{-9}

Figure 37. Random Jitter out of Fiber-Optic Link Triggered by Bit Clock



Bit time = 4.0 ns
Eye opening = < 100 ps
BER = 1 x 10^{-9}

Figure 38. Random Jitter out of Fiber-Optic Link Triggered by RVS & Bit Clock

illustrates a typical problem that occurs when trying to measure random jitter and jitter tolerance. These photos were taken at the limit of frequency/length as indicated by BIST errors appearing on RVS. The first "eye-diagram" (*Figure 37*) was taken using the traditional infinite-persistence scope measurement, where the scope is triggered by a pristine bit-clock. The trigger-clock, shown below the eye-diagram for reference, is arbitrarily placed with respect to the jittered data trace. This is the resulting display of an HP54720D at 8 Gs/s after about four hours of jitter accumulation (approximately 30,000 traces). It would appear that the jitter tolerance of the receiver is only about 45% (i.e., 4.0 ns − 2.19 ns) at the measured BER. *This conclusion is incorrect.*

Figure 38 offers another view of the same link and error rate, when triggered by the error event and shows the *actual* eye opening. This view, triggered by the pristine bit-clock qualified by RVS (ANDed), shows that when the HOTLink indicates an error event, the "eye" is actually fully closed. This photo displays only those traces that contained an error event, about one every four seconds at 250 Mbaud. It is impossible to determine from these photos exactly where the PLL and the data sampling flip-flop have placed the bit boundaries, but it is obvious that if the transition doesn't cross the threshold, the data is lost. (The "ghost" traces that appear in the photo are parts of other error-traces where the eye-closure occurred at some other bit position beyond the limits of the screen.) The discrepancy between these two figures is caused by the triggering and display characteristics of the scope. Even though there are over 30,000 patterns displayed on the first one (*Figure 37*), it just happened that none of the error bits were captured. This could have been because of the relative rarity of the events, and the trigger hold off caused by the scope processing that occurs between measurements.

Receiver Data-Phase Acquisition Time

To measure the HOTLink Receiver response to phase-hops in the incoming data stream, it is necessary to produce a data stream that has a controlled phase change. It is possible to use the two selectable inputs of the HOTLink Receiver to switch between two identical, but skewed, data streams. The data stream used for these tests comes from a HOTLink Transmitter using a good quality clock source. The HOTLink BIST function provides a convenient source of repeatable data and is accompanied by a convenient trigger pulse in the $\overline{RP}$ output that occurs once per BIST loop. The Receiver BIST comparator can be used to determine whether the receiving PLL has maintained phase lock without slipping by monitoring its RVS output. This output will pulse only if there is an error in the received data pattern.

In the test set-up shown in *Figure 39*, the input to the INB+ pin of the Receiver is skewed with respect to the INA± input using the precision skew capability of the Colby delay generator, which can add delay up to 10 ns in 1 ps increments. A carefully placed control pulse (i.e., inputs are changed only when both inputs will be staying at the same logic level for a few bit times to insure that the change does not affect the serial data stream), which is a BIST-synchronous control signal (i.e., the pulse is triggered by $\overline{RP}$ which occurs once in each BIST loop), switches the receiver input between the two data streams. As expected, when the A/$\overline{B}$ input switches between these two streams, no errors are indicated if the skew is small. When the skew is increased, and approaches almost half of a bit time (i.e., 135 to 150 degrees as seen by the PLL Phase Detector) errors are indicated by pulses on RVS. These errors are caused by "bit-slip" in the PLL as it reacquires the new data stream.

By triggering the HP54120D on the signal that changes data streams, it is possible to observe the real-time behavior of the receiving PLL. The scope can be programmed to measure either clock period, or propagation delay between two channels. The former will show each clock period as the loop acquires the new data stream. The latter set-up will show the more traditional phase-alignment measurement that defines Phase-Lock-Loop acquisition characteristics.

Measurements were taken with various amounts of phase difference between the two input channels. The figures that follow show the characteristics of the HOTLink Receiver with phase errors less than 180 degrees and with phase errors at as close to 180 degrees as possible. The first kind illustrates typical link performance. The second kind shows the worst case phase acquisition characteristic.

In the test set-up shown in *Figure 39*, the HOTLink Receiver input is switched to one of the inputs, allowed to stabilize there for a few byte times, and

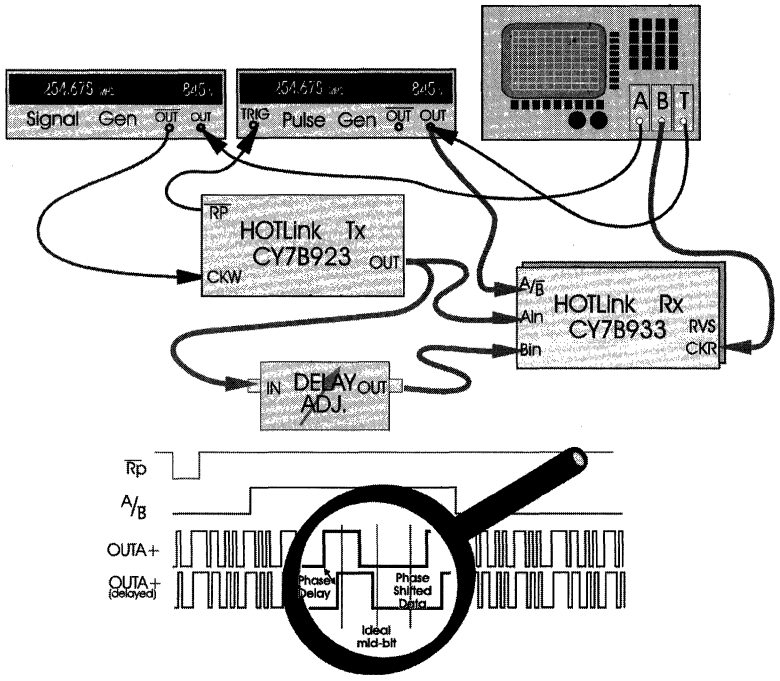


Figure 39. Set-Up to Measure HOTLink Phase Acquisition Characteristics

then switched back. The second switch is an equal phase offset, but opposite sign. During the time when the PLL is trying to regain phase alignment with the incoming data stream, it adjusts the period of the VCO, and thus the output clock of the HOTLink Receiver. As illustrated by the data shown in *Figure 40*, the phase correction begins immediately after the change in data stream. Since the phase error is less than 180 degrees, the correction is always in the expected direction. When the new data stream "lags" the current PLL position, the clock is stretched for a few cycles until it realigns with the incoming data. Likewise, when the new data stream "leads" the current PLL phase, the clock is shortened for a few cycles until it realigns with the incoming data.

The change between any pair of clock (CKR) periods is small, and the maximum deviation usually varies by less than ±1 ns midway through the seven to ten byte-times required to realign the clock. The magnitude of change that can be accommodated without error varies slightly with frequency, and the time needed to resume normal clock periods varies by one or two byte times. There is little or no correlation between settling time and the sign of the phase-change, data-speed, process-corner, V_{CC}-level, or ambient-temperature. For all frequencies, it seems that any phase change that is less than a half-bit time (less about 500 ps) will be accommodated without data corruption. The Byte-Clock adjustment shown in *Figure 40* is the accumulated sum of the ten Bit-Clock periods that combine to makeup the Byte-Clock adjustment, each of which was probably much smaller.

When the phase change is carefully adjusted to the 180 degree position, the correction behavior changes. The correction can be in either direction, since both have an equal capability to realign the PLL clock phase. One direction will cause a bit slip since the decoding logic will find the data appearing one bit earlier or later than expected. The other direction might not slip, but will probably still indicate

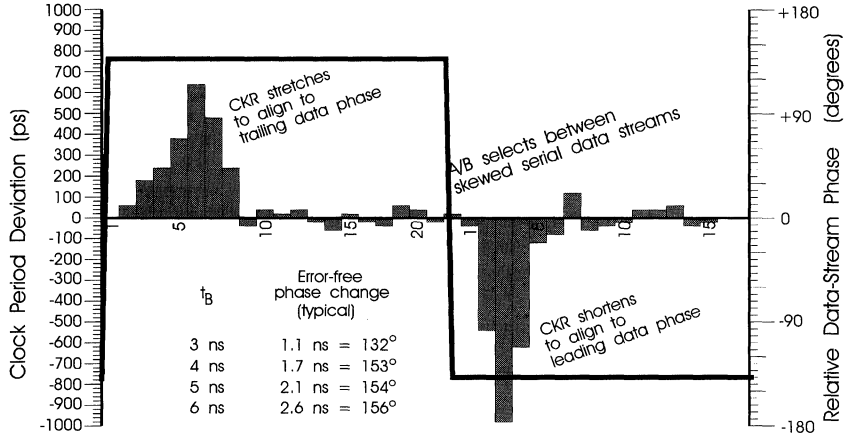


Figure 40. Phase Hop of less than 180 degrees without Data Corruption

a corrupted byte because of a metastable response from the data sampling flip-flop.

Additionally, the phase correction does not start immediately after the change in incoming data phase (see *Figure 41*). The time it *might* take cannot be calculated, because the loop is operating outside its linear response region, and will assume some metastable behavior that could theoretically take forever to clear. It takes several byte times before the PLL accumulates enough error information to cause it to realign itself. When the data has exactly 180 degrees phase offset to the PLL VCO, the Phase Detector may have either no phase-correction effect or a small reverse phase-correction effect, in contrast to its normal, increasing-correction with increasing-error, linear-phase-correction response to smaller phase errors. Once it begins to change, the PLL completes the phase hop in about the same way as the earlier example showed, although over a slightly longer duration. Perhaps counter to intuition, the quieter the received data stream, and the cleaner the VCO clock, the longer this "hang time" will become. (Products with "jitter problems" will never exhibit this "hang phenomenon.") Any jitter or frequency deviation between the incoming data and the VCO provides a tie-breaker and gives enough error information to allow the Phase Detector to begin its change. Once the relative-phase has moved only a little bit, it becomes obvious to the Phase Detector that the error is large and requires a large correction. Complete phase alignment is not

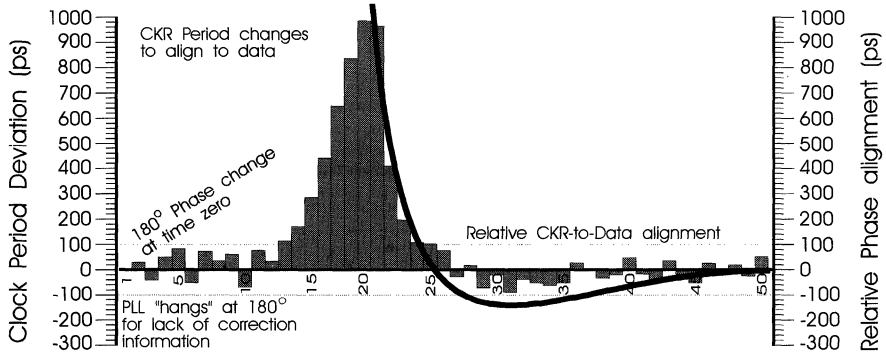


Figure 41. Phase Hop Timing with Exactly 180-Degree Phase Difference

achieved until several byte times after the CKR output has resumed its normal period.

This final alignment time is immaterial for most data-communications systems, since the receiving system will have long since resumed correct data recovery because of the wide jitter tolerance of the receiver. As was shown in the limited phase-step experiment, the HOTLink Receiver will recover the *bit-stream* correctly when the input transitions are more than about 500 ps away from the mid-bit point. However, the *data* must be "framed" to be interpreted correctly, and the time necessary to accomplish framing (HOTLink requires one or two K28.5 characters to frame, depending on current framer mode) will depend on the protocol being used, and how often SYNC characters occur.

Receiver Data-Frequency Acquisition Time

Two serial data streams rarely operate at exactly the same frequency, so the PLL must first acquire the new frequency before it can align the clock to the phase of the new data. The frequency offset that must be accommodated is different for each standard system, but is usually a few hundred parts per million (PPM) variation from a specific frequency. Fibre Channel, for instance specifies a maximum frequency offset of ±100 PPM (±0.01%). HOTLink is specified to accommodate frequencies of ±0.1%, but will typically accommodate more. The chart in *Figure 42* is an illustration of the behavior of a HOTLink receiver as it switches between two data streams that are offset from the local REFCLK frequency by ±0.2%. The acquisition time and the effective clock-period transient time is equivalent to that seen when the receiver is only adjusting for phase differences.

This is typical of the HOTLink acquisition behavior at all operating frequencies, and doesn't vary significantly across process, V_{CC}, or temperature variations. The exact transient size and duration will vary from event to event because of the statistical nature of the "first-change." The excursion could be either to a shorter or a longer clock period depending upon the perceived phase of the new data when the change occurs. Likewise, the time to achieve phase alignment will vary slightly depending on the probability of having a perfect 180 degree phase alignment after the change. None of the clock period excursions measured during this test exceeded ±1.0 ns.

When the data stream is not within the frequency tolerance limits of the receive PLL, the HOTLink automatic frequency-range-control mechanism will affect the transient behavior of CKR. This function continuously monitors the frequency of the VCO and compares it to the frequency of REFCLK. When they are different by a sufficient amount, and for a sufficient time, internal logic will force the VCO to align to the REFCLK frequency. This automatic mechanism insures that the absolute frequency of CKR is never far from its ideal period, and that

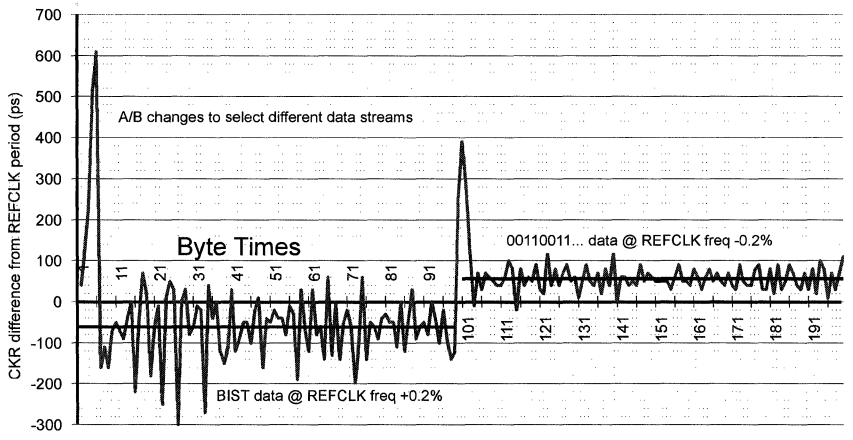


Figure 42. Frequency Hop Within ±0.2%

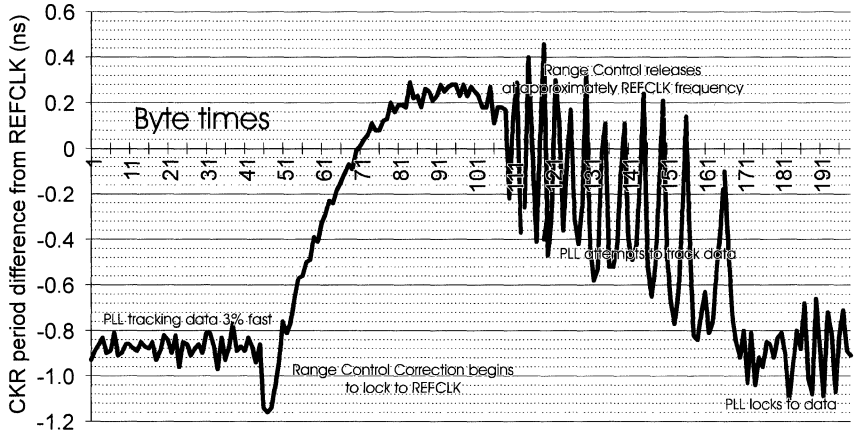


Figure 43. Frequency Acquisition from/to +3.0% Changes CKR Period

when "good" data returns, the PLL will be able to rapidly align to it, and begin correct data recovery.

This test is similar to the previous one, except that the data is not switched externally, and the recorded transient is only the result of the internal re-lock to REFCLK behavior. In this case (shown in *Figure 43*), the applied data stream was offset from REFCLK by about 3.0% (well beyond the data sheet limit of 0.1%). The HP54720 was "glitch triggered" when RVS was HIGH for >60 byte times. (RVS-HIGH for 64 byte times is the PLL out-of-lock indication, since normal data will not yield continuous error indications.)

For the first few bytes (out to about Byte-time 45 in *Figure 43*), the average period of CKR is about 3% faster than the expected 30 ns which indicates that HOTLink has been successful in acquiring the data frequency. When the built-in automatic range control is asserted, there may be a momentary transient in the CKR period caused by the phase and frequency of the PLL relative to the instantaneous bit-stream phase. Next, the VCO will be pulled to the frequency by the internal range-control logic (from about Byte 45 to about Byte 110 in *Figure 43*). Finally the PLL is released to track the incoming data, whereupon it might immediately return to the previous frequency (the frequency of the incoming bit-stream, if any), or as in this illustration, hunt around for an indeterminate time (maybe an indefinite time) until it again finds a signal within its acquisition and tracking range. The exact PLL behavior will depend on the frequency, transition density, timing characteristics and stability of the applied data stream.

CKR period excursions are slightly larger when this range control mechanism is applied, but still under about ±1.2 ns. The period of CKR is the sum of all Bit-Clock periods that occur between CKR transitions.

Receive PLL Jitter Transfer Function

PLL jitter, and consequently recovered clock jitter, can be affected by the noise characteristics and stability of the incoming data stream. The closed-loop transfer function of the PLL is a low-pass filter. Noise components below the natural frequency (f_n) of the PLL will be passed unattenuated and those above f_n will be attenuated. By injecting a measurable and controlled amount of noise (jitter) into an otherwise stable data stream as shown in *Figure 44*, the PLL transfer characteristic can be measured.

In this configuration, the noise source is added to the data-clock source by a resistive mixer, similar to that used for transmitter jitter-transfer testing. The mixer output drives the external bit-rate clock input of a high speed data generator. The Microwave Logic GigaBERT 1400 can run with clock rates above 1 GHz, and can send serial data from an internal memory using this clock. By jittering the external clock, it is possible to create a controlled seri-

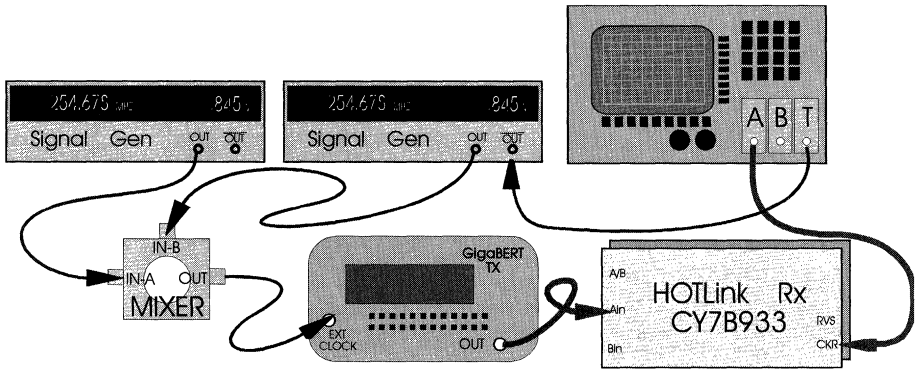


Figure 44. Data-Jitter is Generated by Mixing Noise into Serial-Data-In

al data stream with single frequency jitter noise. The amplitude of input jitter was adjusted to create the desired data jitter amplitude (ns Pk−Pk), and the frequency was varied over a wide range while the jitter was monitored on the CKR output.

Direct jitter generation is difficult to manage because of the need for a single frequency noise source superimposed on an otherwise perfect data stream. Most jitter generators seem to generate either multiple frequency noise sources or have significant DCD and DDJ. The method described for creating jitter suitable for Transmitter jitter-testing creates significant DCD which is ignored by the transmitter PLL, since it only responds to the rising edges of its reference input. Because the receiver responds to both edges of the pulse, this DCD affects the results in undesirable ways. The graph in *Figure 45* shows the relationship between input and output jitter at various input jitter-noise frequencies.

As expected, low frequency noise passes through the PLL filter unattenuated and higher frequencies are attenuated as theory would predict. Also as expected, the apparent bandwidth of the PLL filter varies as the transition density of the data stream varies. For the highest possible transition density (e.g., a 1010101... data stream) the natural frequency is highest, and for lower transition densities it is proportionally lower. The information shown here is characteristic of the HOTLink while receiving normal data. In this case the data was the BIST pattern.

Effective loop-bandwidth varies as a function of data rate, as shown in *Figure 45*. This variation is caused by various gain changes within and between

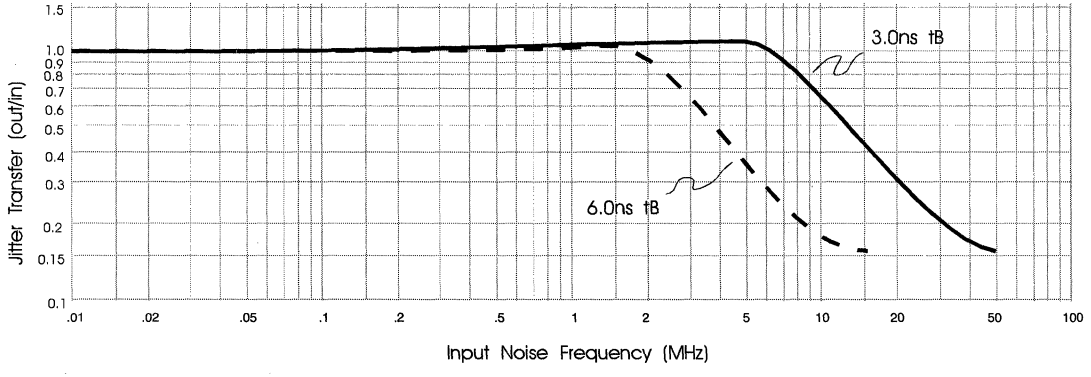
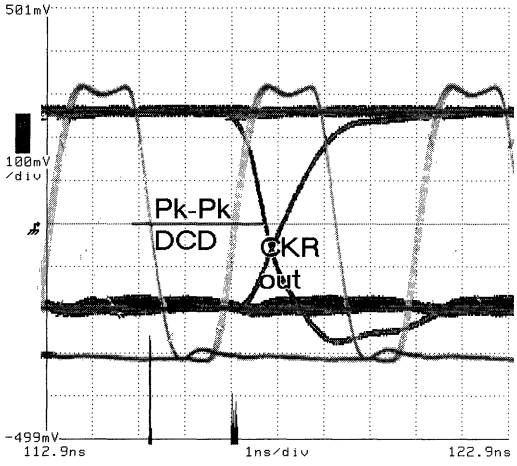


Figure 45. HOTLink Receiver Jitter Transfer Function (BIST Data)

the PLL component blocks. Some blocks have analog gain variations as a function of frequency, and others have a constant output response regardless of operating frequency. The behavior shown in *Figure 45* is unaffected by temperature, V_{CC} variation, and variations in manufacturing tolerance.

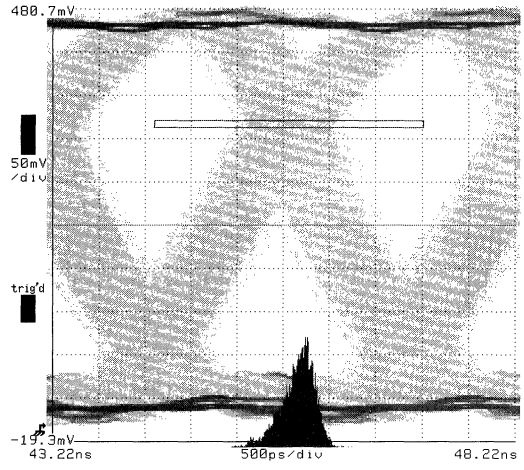
The Receive PLL transfer function is not sufficient to determine what the actual jitter out of the HOTLink Receiver might be. Different types of jitter have different transfer characteristics.

DCD-type jitter causes essentially no output jitter for input jitter magnitudes up to the point where the data is corrupted. The waveforms in *Figure 46* illustrate the jitter feed-through characteristics of the HOTLink Receiver. The input waveform is a continuous stream of 1-0-1-0-... bits that have been artificially distorted with the DCD Jitter generator described later (*Figure 49*). The 4.0 ns bits have been narrowed by about 1.96 ns (see the twin-peak histogram in *Figure 46*), and the CKR output shows less than 100 ps of jitter as illustrated by the darker trace superimposed on the input jitter waveform (note that the two traces have different vertical scales, but the same time scale).



250 Mbaud, Data = BIST,
DCD = 1.94 ns Pk–Pk

Figure 46. CKR Output Jitter as DCD Corrupted Data is Being Received



370 Mbaud DDJ in = 2.18 ns Pk–Pk,
CKR Jitter = 1.31 ns Pk–Pk

Figure 47. CKR Jitter Output as a Function of DDJ Input

When DDJ is applied to the data input, CKR jitter will increase. The illustration in *Figure 47* shows that when DDJ approaches maximum tolerable levels, the CKR output jitter increases appreciably. The test shown in *Figure 47* was performed using the same maximum tolerance jittered data shown in *Figure 35*. This 370-Mbaud signal (well beyond the datasheet limit) was generated using the BIST sequence transmitted through 250 feet of RG59 coaxial cable at 370 Mbaud, while operating with a received BER of $<4.5 \times 10^{-11}$. (The measurement in *Figure 47* is triggered by the pristine-bit-clock, which results in copies of the byte-rate TTL clock displayed at bit-clock intervals.)

This jitter feedthrough is partly caused by the low-frequency characteristic of the jitter, which is determined by its data content, and partly bacuase the actual PLL failure mode (as opposed to Data failure mode) is the same for DDJ as for DCD. In either case, when any data-pulse falls below the DCD pulsewidth limit, the PLL drops some of its tracking and locking information. In a normal data stream this loss is not regular, and causes minimal disturbance. The main effect is to increase jitter on the CKR output.

Summary

The following summary data is representative of the sample tested and described in this report. This evaluation included parts from across the full manufacturing spread, which were tested over the full range of temperature, voltage and frequency of operation. This data is representative of HOTLink in-system performance, but because of the small sample size tested, it cannot necessarily be assumed to be worst case.

Table 4. Summary of HOTLink Jitter Characteristics

Parameter	Characteristic		Condition
Tx Cycle-Cycle Random Jitter	< 6 ps RMS	< 50 ps Pk−Pk	
Tx Input-Output Random Jitter	< 20 ps RMS < 22 ps RMS < 30 ps RMS	< 175 ps Pk−Pk < 190 ps Pk−Pk < 250 ps Pk−Pk	(330 Mbaud) (250 Mbaud) (160 Mbaud)
Tx Data Dependent Edge Displacement		< ±10 ps Pk−Pk	
Tx PLL Deterministic Edge Displacement		< ±2 ps Pk−Pk	
Tx Total Transmitted-Data Jitter	< 26 ps RMS < 28 ps RMS < 36 ps RMS	< 230 ps Pk−Pk < 250 ps Pk−Pk < 300 ps Pk−Pk	(330 Mbaud) (250 Mbaud) (160 Mbaud)
Tx Closed-Loop Bandwidth (3 dB)		1.5 MHz 0.6 MHz 0.3 MHz	(330 Mbaud) (250 Mbaud) (160 Mbaud)
Tx Re-Lock Rate (Locked to Locked)		> 11 MHz/µs > 9 MHz/µs	Typical Hot
Tx Crash Rate (From CKW Stop)		> (45 MHz +19 MHz/µs) > (21 MHz +16 MHz/µs)	Typical Hot
Tx Lock Time (Quiet to Locked)		< 45 ms < 60 ms < 80 ms	Typical (160 Mbaud) Typical (330 Mbaud) Hot (330 Mbaud)
Rx Error-Free-Window (Static Alignment)		> t_B − 250 ps	**Note:** t_B = 1/baud rate (ns)
Rx Random Jitter Tolerance (BER<1x10^{-12})		> t_B − 500 ps	
Rx DCD Tolerance (BER < 1x10^{-12})		> 0.42 x t_B	
Rx DDJ Tolerance (BER < 1x10^{-12})		> 0.62 x t_B > 0.82 x t_B > 0.95 x t_B	(330 Mbaud) (250 Mbaud) (160 Mbaud)
Rx Total Jitter Tolerance (BER < 1x10^{-12})		> t_B − 500 ps	
Rx Input-Output Random Jitter	< 39 ps RMS < 25 ps RMS < 24 ps RMS	< 224 ps Pk−Pk < 180 ps Pk−Pk < 148 ps Pk−Pk	(330 Mbaud, no jitter BIST) (250 Mbaud, no jitter BIST) (160 Mbaud, no jitter BIST)
Rx CKR Cycle-Cycle Peak Jitter (does not include reframing CKR-stretch)		< 100 ps < 300 ps < 0.7 x t_B < 1.0 ns < 1.5 ns	(No input jitter, single data) (No input jitter, random data) (Worst case input DDJ) (Data Phase Hop only) (Loss of Lock)
Rx CKR Maximum Instantaneous Offset Freq.		< REFCLK ±5%	(Unstable, range control active)

Table 4. Summary of HOTLink Jitter Characteristics (continued)

Parameter		Characteristic	Condition
Rx CKR maximum continuous offset freq.		< REFCLK ±0.25%	(Stable, range control inactive)
Rx Run-Length Limit (without cycle slip)		> 200 t_B > 200 t_B > 200 t_B	(330 Mbaud) (250 Mbaud) (160 Mbaud)
Rx Phase Acquisition Time (BER < $1x10^{-12}$)		< 60 t_B < 250 t_B	(typical, <180 degree hop) (includes 180 degree hop)
Rx Frequency Acquisition Time (BER < $1x10^{-12}$)		< 50 t_B < 700 t_B	(delta−freq ≤ ±0.2%) (delta−freq > ±0.2%)
Rx Closed-Loop Bandwidth (3 dB)		9.0 MHz 4.5 MHz 2.5 MHz	(330 Mbaud) (250 Mbaud) (160 Mbaud)
Rx REFCLK Re-Lock Rate (Locked to Locked)		> 2 MHz/µs	
Rx Lock Time (REFCLK Quiet to Locked)		< 200 µs +2 MHz/µs	
Rx Crash Rate (from REFCLK & DATA stop)		> 80 ps/µs	

Hints to Improve Measurement Accuracy

- Use differential scope inputs instead of single-ended measurement systems to remove common-mode amplitude variations from timing jitter. Minor variations in power supply levels that are passed through to the complementary PECL outputs are ignored by the differential receiver, and so should be removed from the measurement. Systems with only single-ended scope inputs should carefully monitor V_{CC}-coupled signals, since a few millivolts of vertical shift can result in several picoseconds of apparent delay variation. Faster edges and minimal loading can minimize the problem, but not eliminate it.

- Random jitter measurements should be taken at the approximate center of the differential swing to minimize "scope arithmetic" and round-off errors that obscure actual performance.

- Bypass PECL load circuits to remove "load-ringing" effects. Power supply and PC board impedance adds directly to the impedance of the termination circuit. Careful attention to power supply bypassing minimizes load related errors.

- AC coupling of input, output, and measurement signals cause unexpected problems if the wave form is non-repetitive, not DC balanced, or if the signaling rate changes. The components used for blocking the DC voltage in the signal will exhibit impedance variations because of their reactive nature. They almost always have non-monotonic transfer functions, and often have self resonant characteristics that are not well documented. High quality DC-blocking modules from HP and other sources are typically specified to be effective over a very wide frequency range (e.g., HP 11742 Blocking Capacitor is useful at 0.01 through 26.5 GHz), but the more common "capacitor soldered on a board" is usually unsuitable for critical measurements.

- Simplified, high quality PECL measurements are possible using the connection shown in *Figure 48*. This a derivative of the standard 80Ω/130Ω Thévenin termination for PECL in which the lower 50Ω of the 130Ω is provided by the scope input impedance. By using low impedance, pas-

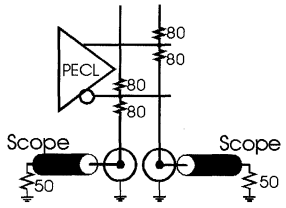


Figure 48. PECL Scope Probe

sive probes to maintain the full input bandwidth of the scope, and by separating the scope probes from the loads, a more representative measurement is possible. This connection yields a probe with an approximate attenuation of 2.6:1, instead of the more usual 10:1 probes. For critical voltage measurements, each such connection must be calibrated, because the actual attenuation factor will depend on the actual values of resistor used for the PECL termination. Since most AC measurements are differential and use only relative voltage levels, this connection is preferred to more expensive probe configurations. Of course, good low-capacitance layout and good quality 50Ω cables and connectors are required to maintain the bandwidth of the measurement system. When the scope is not connected to the test points, a substitute 50Ω resistor should be connected to allow the PECL outputs to operate correctly.

Test Equipment

Relevant Characteristics of Measurement Equipment

Good quality, high-bandwidth, measurement equipment is mandatory to determine the actual performance of the HOTLink and the systems used to test it. To gain an accurate insight into 300-MHz transmission lines, and the picosecond variations which characterize the components that define the limits of operation, it is necessary to use test systems capable of making accurate measurements up to multiple Gigahertz. The list that follows (and the short listing of their relevant attributes) are not the only applicable measurement systems, just the ones used by this design team.

HP54720D High-speed, Real-time, digital sampling scope

Sample Rate = 8 Gigasamples/second

Trigger Jitter < 10 ps

Bandwidth = 2 GHz

1 GHz on each channel with 54721A Input module

2 GHz on single channel with 54722A Input module

This high-performance scope offers the opportunity to observe the actual wave shape with its "real-time" capability. In contrast with the more traditional sampling scope, this instrument will record the signal on its inputs at 125 picosecond intervals until its input buffers are full. The 54720D has the ability to place the triggering event at the beginning, middle, or end of the stored waveform, which allows it to capture random and non-repetitive events.

Tek 11801A Digital Storage Oscilloscope

with SD−22, 12.5-GHz Sampling heads for precision low-impedance measurements

with SD−14, 3.0-GHz Sampling heads for low-load, high-impedance measurements

and DL−11, 5-GHz Delay Line for measurements at the time of the trigger event

Trigger Jitter < 3 ps

Bandwidth > 20 GHz, bandwidth on each channel limited by the sampling head. (SD−22 or SD−14)

This high-performance scope has sufficient bandwidth to observe the actual performance of the PECL outputs of HOTLink. Lower bandwidth scopes and probes often give an erroneous impression of the voltage waveform being measured. The 11801A is best used for measuring repetitive waveforms, since it only accumulates a "dot" for each trigger. Accumulated over

time, this is sufficient for observing repetitive wave forms, and its color-graded histogram ability is very useful for capturing jitter performance.

HP 8560A Spectrum Analyzer

50 Hz to 2.9 GHz

Used for monitoring jitter transfer tests and various clock source attributes to assure the accuracy of the bench setup. The displays that appear on a spectrum analyzer are often ambiguous, since frequency, phase and amplitude variations all cause similar indications. This is a fun instrument to use, but must be interpreted with care. It usually gives more information than can be fully understood, but does offer another view of the system under test from the frequency domain.

HP 54610 500-MHz, 2 channel oscilloscope

This is a small, relatively portable bench scope (i.e., about one cubic foot and can be carried with one hand, in contrast to the other scopes which require a dedicated cart) used for monitoring the function of various bench set-ups and the functionality of the part under test. It has sufficient bandwidth to give an accurate picture of the circuit under test, but is too slow to give accurate results in the previously described precision tests. These scopes are typically used for setting up the various generators, clock sources, and data generators, and for cross-checking the validity of many of the measurements. They were not used to gather actual data, but offer sufficient performance to see that the set-up is working as expected.

Clock Sources

Crystal oscillators are typically used in operational systems because of their stable, predictable, low noise characteristics (as well as their low cost). They were found to be unsuitable for the previously described tests, because of their low-frequency delay and wander characteristics. These unrepeatable effects obscure the jitter characteristic being measured. In operational systems, these effects will cause *no* reduction in link performance, and will merge into the unmeasurable, insignificant background characteristics of the system. To gather the precision information described in this application note, several clock and data sources were used. The list that follows (and the short listing of some relevant attributes) are not the only applicable clock sources, just the ones used by this design team.

RF Generators

HP 8656B Generator 0.1–990 MHz

HP 8647 Signal Generator 250 kHz–1000 MHz

Used as frequency reference generators because of their spectrally clean output, and their high frequency function. They generate small, ground referenced sine waves with great accuracy and are easily programmable from the panel or using a GPIB controller. These generators are typically used to trigger high-performance Pulse Generators, which produce the required levels and edge rates. The generators themselves have acceptable stability and jitter performance for most AC and functional evaluations, but are not sufficient for jitter related tests.

When triggered by a stable source, the jitter performance of the generator improves to almost that of the triggering reference.

Clock Generators

HP 8131A 500-MHz pulse generator

Pulse generators are used to generate the PECL and TTL clock and data sources for testing HOTLink products. The HP8131 can be used by itself or triggered by an RF source. It offers two independent channels with complementary outputs for each.

Wavetek 178 Function Generator
0–50 MHz Function generator

The Wavetek 178 is convenient for generating low frequency signals such as Receiver REFCLK and swept frequency-range tests. It has the capability to generate various wave shapes and can sweep its output fre-

quency across a wide range. It has good stability and is relatively "clean," but exhibits about 200 ps of low-frequency jitter.

Colby Instruments Pulse Generator PG−1000A

The Colby pulse generator is a very stable oscillator that is mechanically tuned, and offers very good spectral purity and good control. It suffers from slight frequency drift until it is fully warmed-up. The design of the instrument is very modular, and offers many specialized controls and options to meet various voltage translation and buffering needs.

Pattern Generators

Microwave Logic GigaBERT − 1400 TX

1.4 GHz max. clock rate

< 2 ps RMS clock jitter, < 20 ps Pk−Pk

No jitter added to output when divided by N to create Bit or Byte Clock

This instrument is actually a very high quality clock generator, packaged with a bit-rate data generator. It can be used for generating bit-clock inputs without the need of an external oscillator trigger source. It was used for many of the bit-rate referenced tests described in this application note by programming it to the required pattern.

Translators and Delay Generators

Colby Instruments
Custom clock buffer and translator box

This general-purpose translator box was used to convert between differential PECL and both true ECL (−5.2V referenced) and "zero-crossing" signals used in various tests. It can accept single-ended signals and return differential outputs with extremely fast edges and no appreciable increase in jitter noise. The inputs all include high quality transmission line terminators that simplify most bench configurations.

Colby Instruments Programmable Delay Line PDL−30A

This general-purpose, mechanical delay generator is capable of generating a repeatable and stable delay up to about 10 ns in increments as small as 1 ps. It is most useful for adjusting mismatched delay lines, and for creating desired skews between various signals. It is essentially a 50Ω transmission line that can be mechanically adjusted in small increments to change the delay. It is programmable by an external keyboard with a digital readout of programmed delay.

Home-Brew and Non-Commercial Test Equipment

Synthetic-DCD Jitter Generator

Duty Cycle Distortion (DCD) can be generated by the circuit shown in *Figure 49*. This circuit uses the stages in a 10H116 (ECL triple-differential amplifier) to perform

- Differential-PECL-input buffering
- Ramp generation
- Threshold shifting
- Level restoration
- Differential PECL output buffering

In this circuit the Transmitter data stream is fed through the Jitter Generator while the Receiver monitors and checks for correct operation. As the control voltage (Vj) input is varied between the 10KH V_{IL} and V_{IH} levels, the duty cycle of the data stream is corrupted in a repeatable and measurable manner. Either of the Vj inputs can be independently adjusted, or they can be differentially driven to get different jitter effects.

The first differential stage of the 10H116 is used as a differential-ramp generator with controlled output impedance and symmetrical rise and fall times. The series Resistor and Capacitor to ground are adjusted to provide a relatively long voltage transition ramp that can be used to manipulate the edge transition timing. The ECL output termination resistors

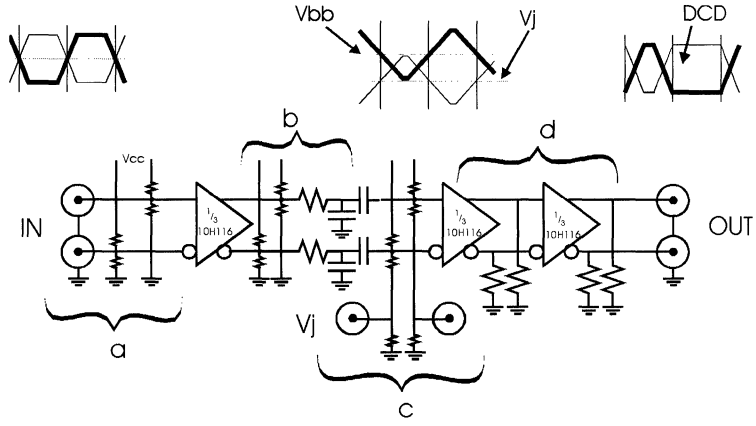


Figure 49. Duty Cycle Distortion Jitter Generator Schematic

shown at the outputs of each differential stage are part of the normal PECL output loads, and can be either the parallel terminations shown at (a) or the single pull-down shown at (d).

The R–C ramp generator at must be tuned to each data rate, to insure that 100% voltage swing is maintained for the narrowest pulses expected. If the Ramp is too long, it will be possible to raise Vj above the level of some data bits, thus "losing" data.

The second differential stage of the 10H116 serves as a voltage comparator that translates the differential, artificially extended voltage-ramps back to PECL swings. The differential (or single-ended) control voltage (Vj) level modifies the restored DC levels of the AC coupled ramps. By adjusting the DC levels at the input of stage two, the average (DC voltage component) of each ramp can be independently adjusted. This adjustment moves the "crossing voltage" which the differential inputs of stage two converts to changes in the timing of the data bit. Additional DC filtering may be required between the Vj input and its input to (d) to insure that high-frequency, single-ended noise does not corrupt the data flow.

The third differential stage of the 10H116 is used to restore crisp-edged, full-swing levels to the serial data, and to drive the subsequent transmission line. In some cases, the PECL output terminations of this stage are provided by the transmission line terminations.

Synthetic-DDJ Jitter Generator

Data Dependent Jitter (DDJ) that approximates the natural effect of long wire-transmission lines, can be generated by the circuit shown in *Figure 50.* This circuit uses the stages in a 10H116 (ECL triple-differential amplifier) to perform

- Differential-PECL-input buffering
- Ramp generation
- Threshold shifting
- Level restoration
- Differential PECL output buffering

In this circuit the Transmitter data stream is fed through the Jitter Generator while the Receiver monitors and checks for correct operation. As the control voltage (Vj) input is varied to cause variations in the "data-corruption" ramps, the data stream is corrupted in a repeatable and measurable manner.

The first differential stage of the 10H116 is used as a differential-ramp generator with controlled output impedance and symmetrical rise and fall times. The series Resistor and Voltage-variable Capacitor (c) are adjusted to provide a relatively long voltage

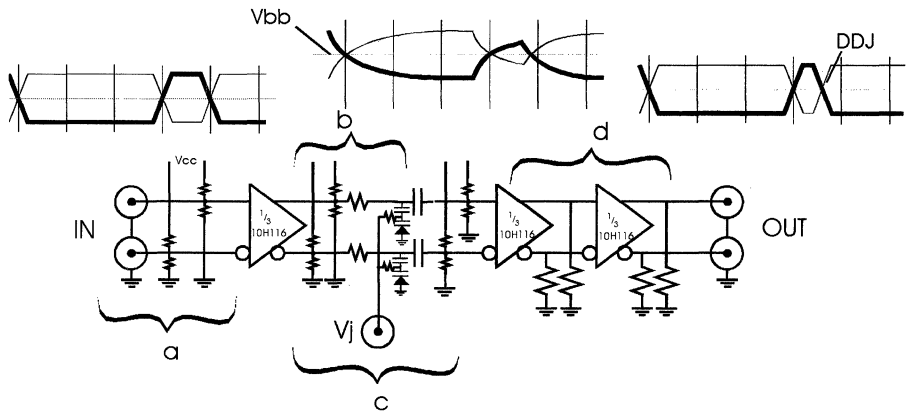


Figure 50. Data Dependent Jitter Generator Schematic

transition ramp that can be used to manipulate the edge transition timing. The ECL output termination resistors shown at the outputs of each differential stage are part of the normal PECL output loads, and can be either the parallel terminations shown at (a) or the single pull-down shown at (d).

The R−C ramp generator at (c) must be tuned to each data rate, to insure that the ramp covers the same number of bits for each speed. If the Ramp is too short, the full spread of pulsewidth dependent jitter will not be generated.

The second differential stage of the 10H116 serves as a voltage comparator that translates the differential, artificially extended voltage-ramps back to PECL swings. The differential restoration resistors put the degenerated waveforms at the optimal voltage so that the inputs of the receiver gate can make a proper logical translation.

The third differential stage of the 10H116 is used to restore crisp-edged, full-swing levels to the serial data, and to drive the subsequent transmission line. In some cases, the PECL output terminations of this stage are provided by the transmission line terminations.

Fiber-Optic Test Bed

The set-up that was used for testing fiber-optic interface capabilities of HOTLink is shown in *Figure 51*. It consists of a HOTLink Evaluation card, several lengths of fiber-optic cable, and appropriate measurement equipment.

A 3-km piece of fiber-optic cable, with only a single splice in it, was used to generate chromatic dispersion. The shorter pieces of fiber, with two connectors between every 500 meters, and the optical attenuator were used to add connector attenuation. The optical splitter and power meter were used to insure repeatability of the measurements. The limits of distance and speed were mostly set by the optical interfaces used, and by the number of connectors in the link.

Coax Test Bed

The set-up used to test wire links is shown in *Figure 52*. It consists of a HOTLink Evaluation Board with suitable connectors and a length of the cable to be used for testing. Various cable types have been tested for speed and distance characteristics. The HOTLink BIST function and the Evaluation Board error indicator combine to offer a clear and unambiguous system to determine the quality of an inter-

connect link, and its suitability to perform at a specified rate.

HOTLink Evaluation Board CY9266–C, CY9266–T, and CY9266–F

The HOTLink Evaluation Card was designed to facilitate early HOTLink system evaluation without expensive or hard to find test equipment. These cards (shown in *Figure 53*) have convenient interfaces for user data and control signals, using either the 48-pin connector used on the IBM OLC–266 card, or a 60-pin card edge connector.

The CY7B923 and CY7B933 include an exhaustive Built-In Self-Test function that can be used to effectively test link performance. It can also be used as a controlled and predictable data source, and as a grader for received data. The receive comparator assures correct functionality of the HOTLink Transmitter, the internal logic in the HOTLink Receiver, and the interconnect link that joins them. These are the essential components of a Bit-Error-Rate tester, except for the reporting mechanism. To fill this need, the Evaluation Cards include a PLD programmed to be a two-digit accumulator and display driver. The Error Display will show the number of Error Bytes received during the BIST sequence, by counting the HOTLink RVS outputs.

BIST

HOTLink Transmitter and Receiver include a comprehensive link test function, as part of the functionality of the basic chips. When the HOTLink Transmitter $\overline{\text{BISTEN}}$ is enabled, the part creates a continuous 511 byte (2^9-1 bytes) pseudo-random stream of 8B/10B-encoded data patterns which the HOTLink Receiver checks byte-by-byte. The 256 possible data patterns are sent once each, and the 12 Special Characters and the 4 specified error codes are sent sixteen times each (except C0.0 which is sent only 15 times) for a total of another 255 data patterns. For a complete list of codes used in the 8B/10B encoder and the Special Character and Error Codes, see the CY7B923/933 HOTLink Tx/Rx Data Sheet.

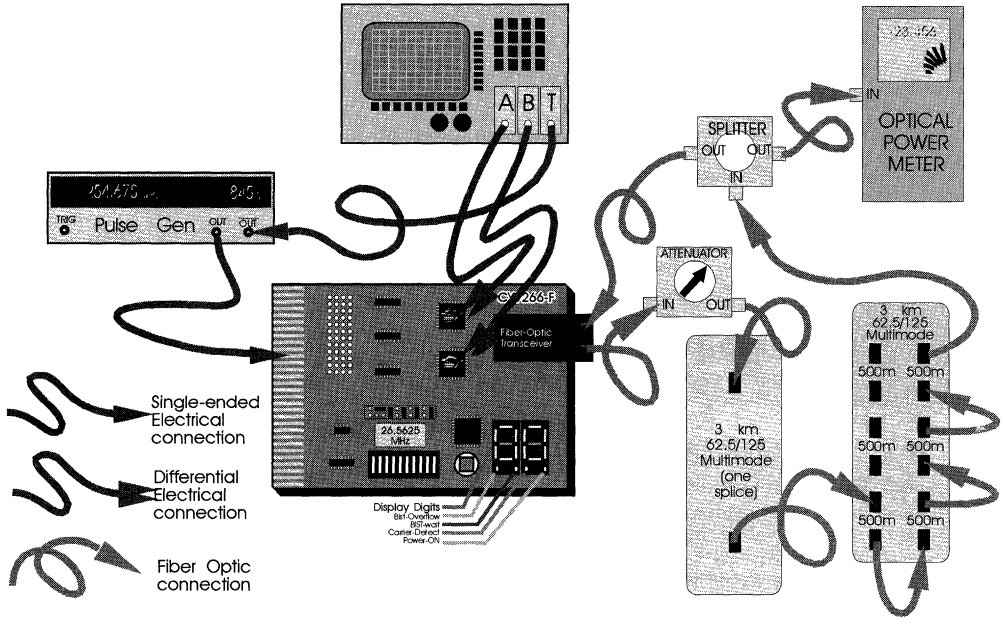


Figure 51. Fiber-Optic Test Bed Facilitates Random-Jitter Testing

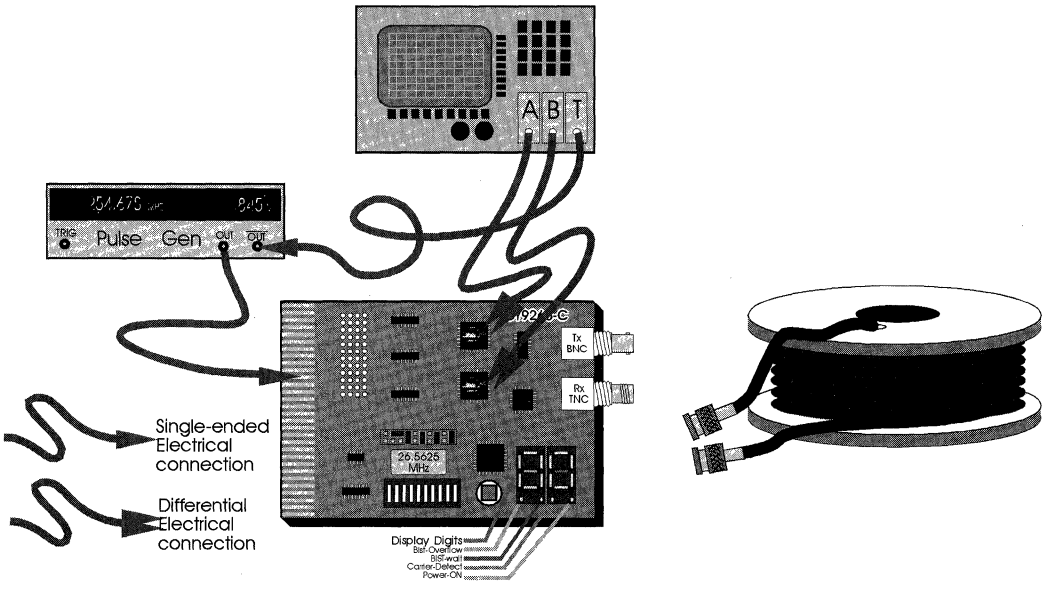


Figure 52. Coax Test Bed to Test for Deterministic Jitter

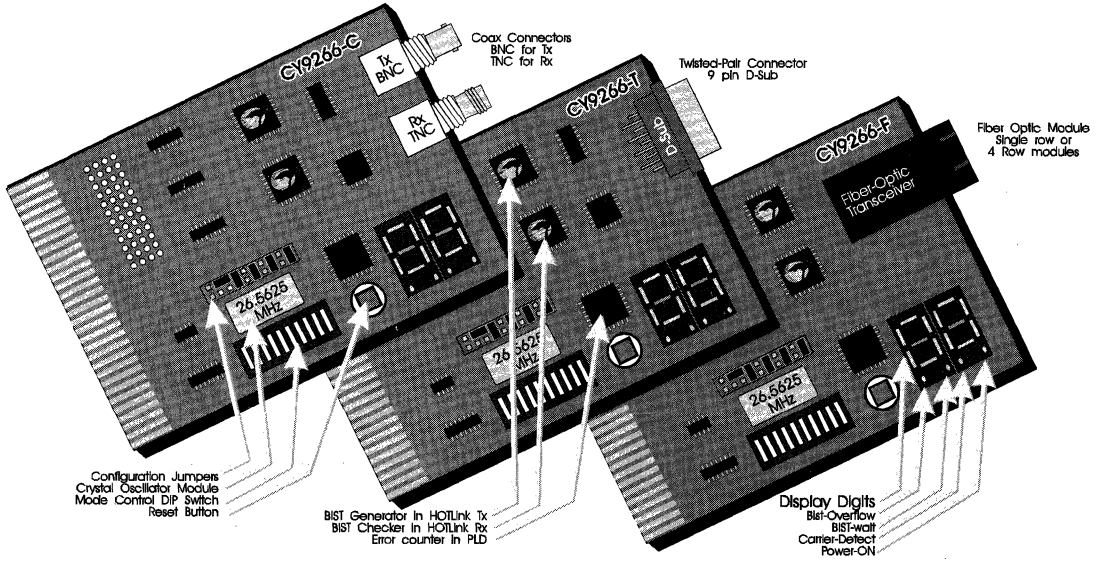


Figure 53. HOTLink Evaluation Boards Form the Core of a Comprehensive Evaluation System

If errors are discovered in the received sequence, received running disparity, or received transmission codes, they are flagged by the RVS output of the HOTLink Receiver. A full discussion of the BIST function of HOTLink is contained in the "HOTLink Built-In Self-Test (BIST)" application note.

For Further Information

HOTLink User's Guide

Hewlett-Packard Catalog
Hewlett-Packard Test & Measurement Division
Mail Station 51LSJ P.O. Box 58199
Santa Clara, CA 95052-9943
(800) 452-4844 or (408) 553-7271

Tektronix Catalog
Tektronix
26600 Southwest Parkway P.O. Box 1000
Wilsonville, OR 97070-1000
(800) 426-2200 or (503) 627-1916

Microwave Logic
285 Mill Rd
Chelmsford, MA 01824
(508) 256-6800

Colby Instruments, Inc.
1810 14th St,
Santa Monica, CA 90404
(310) 450-0261

Understanding Bit-Error-Rate with HOTLink™

Understanding Bit-Error-Rate

The concept of an error rate for digital systems may seem somewhat foreign to many digital designers. The message has always been that digital circuits always switch to either a one or a zero, and that if the circuit doesn't do it correctly then it must be broken.

The real world is quite different. Typical computer networks lose or corrupt packets, disk and tape storage require re-reads of data (or even error correction), and large DRAM memory arrays may have bits corrupted by α-particles and require ECC correction. These random events occur regularly in these computer systems, and the necessary error detection and recovery mechanisms are planned for in their design. Under conditions that can cause these types of errors, the system's performance is determined both by the circuit design, and by *probability*.

Serial data communications systems, such as those based on HOTLink™, must also deal with probabilistic forms of errors. The amount of error detection and recovery built into the system is often determined by the tolerance of the system to bit errors, and how often these errors occur. In these types of systems the errors are (for the most part) caused by either intrinsic or extrinsic noise sources that can affect any or all parts of a data link. The measurement and specification of a bit-error-rate (BER) exists as a way to quantify the susceptibility of a digital link to these noise factors.

Bit-Error-Rate Definition

Bit-error-rate is the relationship of the number of bits received incorrectly, compared to the total number of bits transmitted. This relationship is shown in *Equation 1*.

$$BER = \frac{\#\ of\ bits\ in\ error}{\#\ of\ bits\ transmitted} \qquad \text{Eq. 1}$$

This simple relationship is the basis for all BER measurements and specifications. It assumes that all transmitted bits were sent error free.

BER is usually specified as a number times 10 raised to a large negative exponent. Common requirements for serial links are generally in the range of $1x10^{-6}$ to $1x10^{-15}$.

BER numbers by themselves do not represent any period of time. They are only a ratio of numbers of bits sent and received. A specific BER, when related to time, can yield an MTBF (mean time between failure) for a serial link. This relationship is shown in *Equation 2*.

$$MTBF_{(hours)} = \frac{1}{BER \times bits\ per\ hour} \qquad \text{Eq. 2}$$

HOTLink operates at bit rates of 160 Mbits/sec to 330 Mbits/sec. An operating BER of 10^{-12} for a 330 Mbit/sec data stream would have an MTBF of 0.84 hours. This is equivalent to detecting an average of one bit in error for every 0.84 hours of operation. This same link at the same BER, but operating at 160 Mbits/sec, would detect an average of one bit in error for every 1.74 hours of operation.

Link-Based Errors

The BER for a specific link is not based on the HOTLink components used at either end of the link. A HOTLink Transmitter connected directly to a HOTLink Receiver (when operated within their datasheet parameters) has a BER of zero. As other components are added to the link (transformers, transmission lines, opto-electric transceivers, connectors, optical fiber, etc.) the link BER begins to

grow. These components add distortion to the transmitted signal. This distortion can come in many forms, including attenuation, dispersion, increased jitter, and DC offset. The unpredictable element that is also added is susceptibility to noise.

Sources of Errors

In a communication link, errors are generally separated into two categories: intrinsic and extrinsic. Intrinsic errors are those caused by the components used to create the link. Extrinsic errors are those caused by external influences that affect the operation of the link.

Intrinsic Errors

Intrinsic errors are those errors due to the design, components, and implementation of a link. These errors can be caused by internal noise sources (i.e., thermal noise), poor electrical connections, and (with some systems) receiver sampling errors.

Optical Links

Optical links are often used in areas where strong electrostatic and electromagnetic fields are present, to limit the number of errors caused by these extrinsic noise sources. In the absence of these noise sources, many users are surprised to find that optical links are often more error prone than an electrical or copper based link. These errors are due to the physical components used to make the link (optical driver, optical receiver, connectors, optical fiber, etc.) and not to the serializer and deserializer components used at the ends of the link.

Optical fibers, even the best ones, contain numerous impurities and flaws. As light strikes these minute flaws it gets vectored off at different angles or absorbed in the cladding. This is not generally a problem for short links, but long ones contain many such flaws. These flaws work to both reduce the amount of light that reaches the receiver (attenuation), and to spread out the transmitted pulsewidth (dispersion).

Each optical connector also causes signal loss and pulse degradation similar to the flaws inside the fiber. Here the main loss mechanism is back reflection and attenuation due to contamination, cleaving faults, or poor polish of the fiber end. These types of signal degradation are translated into increased jitter by the opto-electric receiver. This jitter (within certain limits) does not increase the BER of a link. As long as the opto-electric receiver's output jitter remains within the receiver's (deserializer) jitter tolerance, the link should remain error free.

One of the largest causes of random or noise-induced errors is the optical receiver. Here light received from the fiber is converted to an electrical signal through a transimpedance amplifier. This amplifier must respond to current changes in the PIN photodetector of less than 1 μA to detect the presence or absence of light. This low signal-level makes the receiver preamplifier susceptible to thermal and shot noise, and converts these into random jitter. This random jitter has a Gaussian distribution and is directly influenced by the signal-to-noise ratio (SNR) of the optical link.

The optical receiver is also quite sensitive to external EMI sources. External static discharges or power supply transients often make their way to the optical receiver where they manifest themselves as erroneous bits.

Electrical Links

Electrical or *copper* based links are also subject to errors, however errors in these types of links are (in almost all cases) due to extrinsic sources. While the components used to make an electrical link are still sources of noise in a system, the amplitudes of these noise sources are tens of dB below any of the electrical thresholds used in the receiver.

The one possible exception to this deals with an improperly installed or maintained system. If low quality components are used in a non-benign environment (corrosive atmosphere, salt spray, etc.) it is possible for the interconnections and even the cable itself to degrade. The galvanic action of dissimilar metals in such an environment can generate significant noise in the system.

Transmitter (Serializer)

In a communication link the transmitter is generally never considered to be a source of errors in the link. This is due primarily to the pseudo-synchronous

nature of its design. In the case of HOTLink, the transmitter operates fully synchronous to its internal synthesized bit-clock. So long as the clock, incoming data, and power, meet their specified parameters the part should not generate any errors.

The one exception to this is the possibility of disturbances at the subatomic level. While it is theoretically possible for SEU (single event upset) to occur due to α, β, or some other subatomic particle emission, this event is not expected. High-reliability design practices, coupled with the robust nature of BiCMOS circuitry used to make HOTLink, make this highly improbable.

Receiver (Deserializer)

The HOTLink Receiver is based on a high-reliability fully differential analog PLL (phase-locked loop). It is designed to remove all intrinsic error sources from the receiver, and to block many of the extrinsic error sources.

As long as the HOTLink Receiver is presented with valid power and data (meeting its datasheet requirements), it is effectively error-free in operation just like the HOTLink Transmitter. As with any electronic component, it may be susceptible to SEU phenomena, however none have ever been observed.

For electrical connections where no external receiver preamplifier is present, the receiver sensitivity may also have an effect on the link BER. The HOTLink Receiver typically will only require 10 mV of differential signal (50 mV worst case) at the receiver input for proper operation. These enhanced low-amplitude inputs of the HOTlink Receiver permit operation with much longer external cables, or cables having much more equalization present, at very low bit-error-rates.

Extrinsic Errors

Extrinsic errors are those caused by external or outside influences. These errors are caused by things like spikes, sags, and surges in the power mains, electrostatic discharges, RF emissions, and cable/connector vibrations.

Power Supplies

In some cases normal power-supply noise and ripple is grouped in with extrinsic sources of errors, however a good design will place this as part of the intrinsic errors. Power-supply noise becomes extrinsic when externally generated noise is allowed to *pass through* the power supply and reach the serializer, deserializer, and media driver/receiver. These external noise sources can be as small as an ESD discharge from someone touching a cabinet, or as large as a lightning strike. Depending on the characteristics of the noise source (and how much is allowed to reach the serial-link components), it may be able to induce link errors.

Many standard appliances operate with motors that generate very strong noise fields. Some examples of these are electric drills, vacuum cleaners, mixers, etc. Basically anything using a motor that contains brushes. As these appliances operate they radiate strong RF fields, and reflect large amounts of RF energy back into the power mains. Limiting the effects of such power-coupled sources usually involves various types of power filters or conditioners on the front-end of the system power supply.

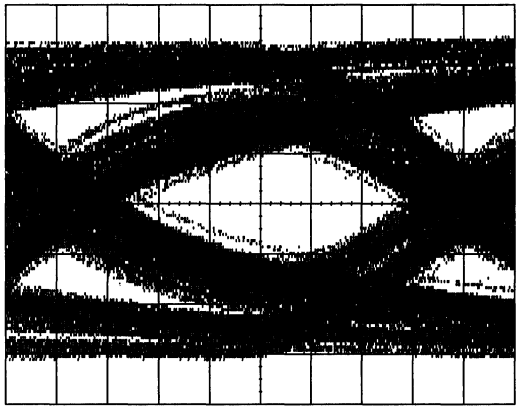
Optical Links

Optical links are fortunate in that the fiber-optic cables themselves are immune from externally generated noise. The weak link in an optical connection is the susceptibility of the receiver to external noise. In many cases the largest cause of noise for an optical receiver is the optical transmitter mounted directly adjacent to it. This requires careful layout and isolation techniques to keep the noise generated in the optical driver from affecting the sensitive optical receiver.

Electrical Links

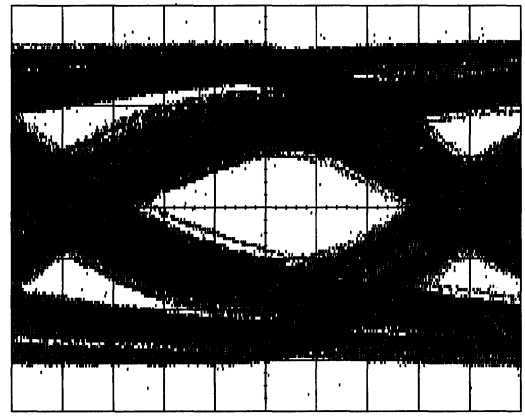
Electrical links are in some ways at a disadvantage when compared to optical links in that they *are* affected by external electromagnetic fields. Just how much they are affected is based on many different characteristics. These are primarily the cable-type used, the data rate, and the strength of the external field.

Cypress has tested multiple types of copper media (different impedances and diameters of coaxial and



Ch. 1 = 200.0 mV/div Timebase = 500 ps/div

Figure 1. Eye Pattern without Forced Noise



Ch. 1 = 200.0 mV/div Timebase = 500 ps/div

Figure 2. Eye Pattern with Forced Noise

twisted-pair cable) to determine how far a reliable link can be operated. What was learned was that the higher-impedance and lower-attenuation cables allowed error-free communication for the greatest distances.

Some of these links were also tested in the presence of an uncalibrated noise source (i.e., an electric drill). This testing, while not directly quantifiable, does allow numerous observations to be made as to how a copper-based link responds to external noise.

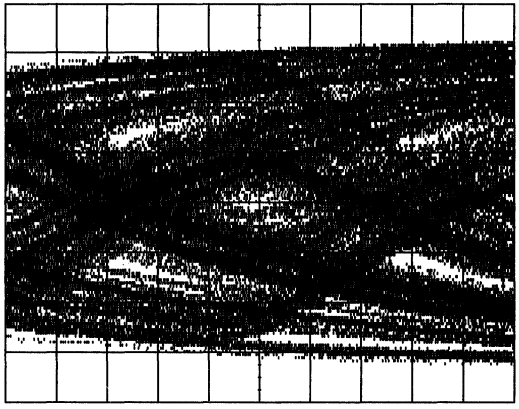
The first observation was that short copper-based links ($\leq$100m), when implemented with shielded cables (coax or STP), are relatively immune to the noise generated by the noise source. *Figure 1* shows the "eye" at the end of a 91.2m (300-foot) piece of RG59 coaxial cable running the HOTLink BIST (built-in self-test) at 25 MHz with normal office electrical noise present. At this distance there is significant ($\leq$30%) jitter present in the link, and the eye (as viewed on a digital sampling scope) is reasonably open (see the Cypress Semiconductor application note "HOTLink Design Considerations" for an explanation of jitter and eye patterns).

For noise testing, a small number of turns (six) of the cable were tightly wrapped around the body of an electric drill to maximize the noise coupling. The eye pattern with the noise generator enabled is shown in *Figure 2*. Under these conditions the eye becomes a bit fuzzy around the edges, but the center remains mostly open.

This "fuzz" is in fact multiple sample points created when the external noise caused the received signal to move from its normal position. Rather than being just a single dot on the screen, each of these points is actually part of a continuous waveform. Because of the random nature of the noise source (relative to the scope trigger and serial data) and the repetitive sampling used to display a signal, it is not possible to view the actual altered waveform.

Even with this strong of a noise source, the HOTLink Receiver detected no errors during the 15-minute period of this test. This does not mean that such a link would remain error free indefinitely, just that the SNR in this configuration is sufficiently large that most received pulses still fall within the normal range of the receiver for a correct 1 or 0 to be detected.

As the cable gets longer the signal continues to degrade and the eye closes. This closure is not a linear function; it is more logarithmic in nature. At 121.4m (400 feet) the eye (for this cable-type and data-rate), as shown in *Figure 3*, is effectively closed ($\leq$5% eye opening). Under these conditions the HOTLink Receiver (in the absence of strong external noise sources) will still correctly detect the data as an error-free stream. Now however, when the noise source is enabled, the receiver detects multiple and near continuous errors.



Timebase = 500 ps/div Ch. 1 = 100.0 mV/div

Figure 3. Error Free Eye Pattern at Maximum Cable Length without External Noise

Jitter

A popular misconception is that the reason for the detected errors in a communications link is the jitter accumulation in the link. While jitter definitely does play a part in determining the BER for a system, it alone does not cause errors.

The link measurement in *Figure 3* shows a very large amount of jitter present, yet the link operates error free. A link of this type can meet a BER of 10^{-12} (or better) as long as the external noise remains controlled. In a similar fashion, a link measuring minimal jitter (<10%) could become unusable if presented with a strong enough noise source.

Specifying BER

The BER for optical links is usually specified as a transfer function relative to signal-to-noise ratio. This is due to the way an optical signal is modified as it moves down a fiber. This specification does not take into account any of the extrinsic noise sources that can effect the opto-electric converters that are part of the link, and assumes that all errors are due to the pulse degradation and how the signal is interpreted by the opto-electric receiver.

For copper cables it is a bit more complex. The specification is still based on SNR, but now is a set of N curves in N-dimensional space. These curves must take into account such things as the launched power, the spectral content of the source signal, the type of shield on the cable, the receiver sensitivity, and how much (if any) equalization is present. Unlike optical cables, the BER specifications for copper links must take into account extrinsic noise sources because these are the primary cause of bit-errors in an electrical link.

BER Floor

A bit-error-rate floor is that point in a link where the BER is limited by something other than the SNR. This occurs in links when no increase in launched power into the cable or optical fiber will yield an improvement in the BER.

For electrical cables the BER floor sits at the point where the eye effectively closes and signal transitions can no longer be properly detected. In these cables, the shape of the eye is determined only by the frequency characteristics of the signal launched into the cable and the cable's attenuation characteristics (and any signal conditioning if present).

Figure 4 shows the BER floor for Type-1 shielded twisted-pair (STP) cable when used with HOTLink. This testing was performed on four different CY9266–T HOTLink Evaluation Boards, under room temperature conditions, with no cable equalization or special conditioning of the environment (see also the "CY9266 HOTLink Evaluation Board User's Guide" for additional information on the CY9266). All areas under the curve allow normally error-free link operation, with all detected errors due to extrinsic noise sources. All areas above the curve identify where the link will operate with near continuous errors, regardless of the presence or absence of external noise sources.

This same curve is plotted with the frequency axis on a logarithmic scale in *Figure 5*. Now the portion of the curve determined by the cable characteristics is effectively a straight line. This shows that the transfer function for the BER floor relative to frequency is actually an exponential function. Two other limits actually exist in the BER floor for HOTLink. These are the upper and lower frequency limits of the HOTLink Transmitter and Receiver circuits.

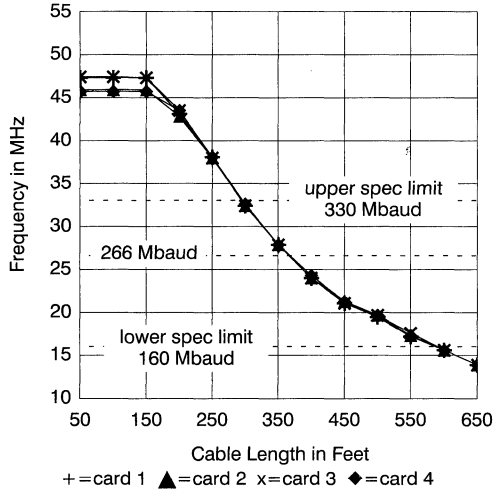


Figure 4. BER Floor for Type-1 STP Cable, Linear Frequency Scale

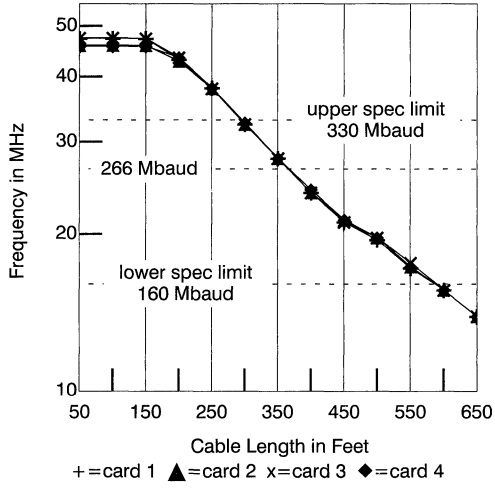


Figure 5. BER Floor for Type-1 STP Cable, Log Frequency Scale

HOTLink is a trademark of Cypress Semiconductor.

The upper frequency limit can actually be identified in *Figures 4* and *5* as the flat horizontal section between 50 and 150 feet. In this area the operating limit is not due to the cable, but is instead due to characteristics of the phase-locked loops in the transmitter and receiver.

The lower frequency limit (not directly identifiable on the graphs) is that frequency below which the HOTLink Transmitter and Receiver cannot remain in a proper phase-lock to communicate valid data. For those parts used in this evaluation this is somewhere around a 13-MHz byte-clock rate (130 Mbits/second).

Conclusion

The key observations for bit-error-rate measurements with HOTLink are:

- The HOTLink Transmitter and Receiver have an intrinsic error rate of zero.
- Optical links suffer primarily from intrinsic noise sources in the optical transmitter and optical receiver, and extrinsic sources in the optical receiver.
- Electrical links suffer primarily from extrinsic noise sources.
- The exceptional BER floor of HOTLink is due primarily to the very high jitter-tolerance of the receiver and low jitter generated in the transmitter.

Driving Copper Cables with HOTLink™

Overview

The HOTLink™ family of data communications products are designed to support communication over both optical and copper cables. Each media type has specific cost, bandwidth, emissions, and distance criteria. This application note covers the methodology and evaluation of various forms of attachment to copper media. It is expected to be used in conjunction with a companion application note titled "HOTLink Design Considerations."

Primary Topics

The primary topics covered in this application note are:

- Transmission lines
- Copper cable types
- Direct coupling
- Capacitive coupling
- Transformer coupling
- Quantitative Interface Comparison

Introduction

The electromagnetic spectrum covers all wavelengths from near zero through infinity. This includes all radio, microwave, light, x-ray, and cosmic-ray wavelengths. *Table 1* lists the classifications of those frequencies and wavelengths usually associated with data communications over copper media. Communication links based on HOTLink products utilize frequencies in the HF, VHF, and UHF bands.

Copper cables (or circuit board traces) are used to move electromagnetic energy from one place to another. With slow signal-switching speeds (and short interconnect distances), a signal placed on one end of the cable will eventually show up at the other end. Systems of this type are seen and used in homes and offices every time a light switch is opened or closed. Here the primary concern is delivering energy to a load.

In high-speed communications systems, many other concerns exist. Not only must energy be delivered to the communications link receiver, but the signal delivered must arrive with minimal distortion. Delivery of electromagnetic energy with minimal (or controlled) distortion requires the proper use of transmission lines.

Transmission Lines

In the most general sense, a transmission line is any closed system for directing electromagnetic energy. (While antennas may also direct electromagnetic energy, they are not part of a closed system and are thus not considered transmission lines.) Any transmission line meets the following three criteria:

- Has a system of material boundaries
- Has a start and end point
- Capable of directing electromagnetic energy

Table 1. Electromagnetic Band Classifications

Band	Band Name	Frequency Range	Wavelength Range	Common Uses
ELF	Extremely Low Frequency	30 Hz–300 Hz	10 Mm–1 Mm	Commercial AC Power Distribution
VF	Voice Frequency	300 Hz–3 kHz	1 Mm–100 km	Analog Telecommunications
VLF	Very Low Frequency	3 kHz–30 kHz	100 km–10 km	Voice and Music Reproduction, Submarine Communications, Sonar
LF	Low Frequency	30 kHz–300 kHz	10 km–1 km	Commercial AM Radio, Shallow-to-Medium Depth Sounders
MF	Medium Frequency	300 kHz–3 MHz	1 km–100 m	Commercial SW Radio, Amateur Radio, Marine Radiotelephone
HF	High Frequency	3 MHz–30 MHz	100 m–10 m	Commercial SW Radio, Amateur Radio, Citizen Band Radio
VHF	Very High Frequency	30 MHz–300 MHz	10 m–1 m	VHF Television Broadcast (Channels 2–13), FM Radio, Amateur Radio, Cordless Telephones
UHF	Ultra High Frequency	300 MHz–3 GHz	1 m–10 cm	UHF Television (Channels 14–83), Microwave Ovens, Aeronautical Radionavigation
SHF	Super High Frequency	3 GHz–30 GHz	10 cm–1 cm	Microwave Communications, Marine Radar, Aircraft Tracking and Radar
EHF	Extremely High Frequency	30 GHz–300 GHz	1 cm–1 mm	Space Communications, Radio Astronomy

Electromagnetic energy moves along a transmission line as an electromagnetic wave, composed of electric and magnetic fields. These waves and fields travel (or propagate) down a transmission line at a finite rate, determined primarily by the dielectric in the transmission line.

Transmission lines generally fall into two different types, based on the orientation of the electromagnetic fields as they propagate down the transmission line. All dual-conductor transmission lines (coaxial, twisted-pair, twinaxial, microstrip, stripline, etc.) propagate their electromagnetic energy with both the electric and the magnetic fields oriented perpendicular to the direction of propagation. This is known as Transverse Electric Magnetic (TEM) mode. *Figure 1* shows a graphic representation of these fields within a coaxial cable.

Single-conductor transmission lines (also known as waveguides) propagate their energy in multiple modes known as either TE (Transverse Electric field) or TM (Transverse Magnetic field). In these modes, one or the other of the fields is oriented parallel to the direction of propagation.

Both TEM and TE/TM transmission lines have cutoff frequencies—points in the electromagnetic spectrum where the transmission modes change. For TEM transmission lines the cutoff frequency determines the upper frequency limit for TEM

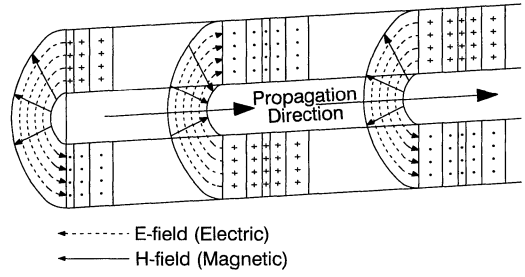


Figure 1. Electric and Magnetic Fields for TEM Mode in a Coaxial Transmission Line

propagation. Signal components higher than the cutoff frequency will propagate in TE/TM modes.

For TE/TM (waveguide-type) transmission lines, the cutoff frequency determines the frequency *below* which energy cannot propagate. This cutoff frequency is determined by the physical dimensions of the waveguide, and is calculated using *Equation 1.*

$$f_{(c)} = \frac{300,000km}{2 \times Wall\_Width} \qquad \text{Eq. 1}$$

Applying this equation to the data rates used with HOTLink shows that such a structure would be very impractical. It would require a cross-sectional width of near 5 meters to propagate the low-frequency signal components (33 MHz) of even the highest operating data-rate (330 Mbps) of HOTLink. Because of this restriction (and others) all remaining discussion will only deal with TEM-type transmission lines.

TEM Transmission Line Characteristics

The conductors used to form a transmission line have numerous distributed parameters that determine its operation and characteristics. These distributed parameters include the series inductance (L) of the conductors in the transmission line, the shunt capacitance (C) between the conductors, the series resistance (R) of the conductors, and the shunt conductance (G) between the conductors. Because these properties remain constant per unit length of the transmission line, they are referred to as distributed properties. These parameters are functions of the diameter and spacing of the conductors and the dielectric constant of the spacer used between them. A schematic equivalent of these elements in a balanced (two-wire) transmission line is shown in *Figure 2.*

Transmission lines are usually characterized by two parameters: characteristic impedance (Z_O) and velocity of propagation (V_P). Proper determination of these values is imperative to allow the transmission line to be used correctly.

Characteristic Impedance

The characteristic impedance identifies the impedance seen by a source when driving a transmission line terminated at the load-end in a pure-resistance equal to the characteristic impedance. While this appears to be a circular definition, it is valid. If the load end of the transmission line is terminated in an impedance other than the characteristic impedance of the line, the source end of the line will see an impedance different than either that of the load or the characteristic impedance of the line. Because this characteristic impedance is generally unaffected by frequency, a transmission line terminated in its characteristic impedance has the same load characteristic of a fixed resistor.

In most transmission lines the series-R and shunt-G values are usually very small and have minimal effect on the impedance of the line. This means that the characteristic impedance is determined almost entirely by the series-L and shunt-C shown in *Figure 2.* This relationship is shown in *Equation 2.*

$$Z_O = \sqrt{\frac{L}{C}} \qquad \text{Eq. 2}$$

Velocity of Propagation

In space an electromagnetic wave travels at nearly 300,000,000 meters per second (speed of light). Moving this same signal through a transmission line

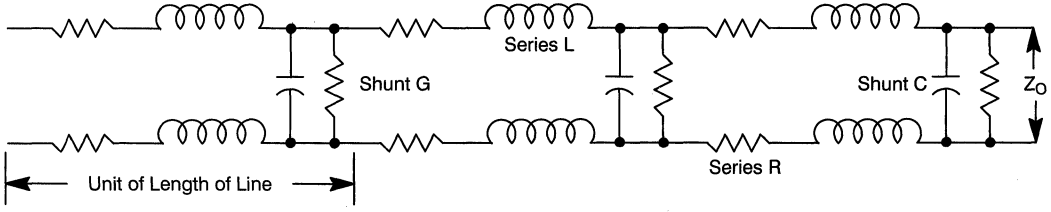


Figure 2. Equivalent Circuit of a Transmission Line

with a vacuum for the dielectric separator between the conductors allows the wave to propagate at or near this same rate.

Real transmission lines are seldom found with a vacuum dielectric. Instead, various non-conductive materials are used to maintain the spacing between the two conductors of the transmission line. These separators all have different dielectric constants, and all of them slow down the propagation of the signal. The rate the signal propagates, relative to the speed of light, is known as the Velocity of Propagation (V_P) and is usually expressed as a percentage (sometimes expressed as a propagation delay in time per unit distance). This velocity difference may be calculated using *Equation 3*, where ε_r is the relative dielectric constant of the transmission line.

$$V_P = \frac{1}{\sqrt{\varepsilon_r}} \qquad \text{Eq. 3}$$

For this calculation to work, the entire electromagnetic field must propagate in the dielectric. Many transmission lines are structured such that some of the field propagates in the dielectric, while other parts propagate in the surrounding air. For transmission lines of this type the equation must be modified to account for the mixed dielectrics.

TEM Transmission Lines

TEM Transmission lines may be grouped in any number of different ways: by length, by construction, by dielectric, by usage, etc. For operation with HOTLink they are generally split into two categories: unbalanced (single-ended) and balanced (differential) transmission lines.

Unbalanced Transmission Line

Figure 3 shows a driver/receiver combination used in an unbalanced transmission line. In this configuration, a single driver sources and sinks current into the transmission line with the return path provided by a common ground.

In this configuration, other communications paths can share the common ground. This allows for fewer wires in a cable, and fewer contacts in a connector. The main problems suffered by this type of transmission line are susceptibility to external noise, crosstalk, ground potential differences, and limited noise margin.

In an unbalanced transmission line, the electromagnetic field necessary for signal propagation exists between the driven line and the ground path. The receiver operates by comparing the amplitude of the received signal relative to ground or some other reference.

Balanced Transmission Line

Figure 4 shows a driver and receiver configured for use in a balanced transmission line. In this configuration, two drivers source and sink complimentary signals into the two wires of the transmission line. These signals need to be matched in amplitude, and must be 180° out of phase with each other for the transmission line to work properly.

In this configuration, a common ground is not always necessary. Since there is no ground requirement, the sensitivity to ground potential differences is greatly reduced. All that is required is that the signals remain within the input (common-mode) range of the receiver.

Susceptibility to crosstalk is also greatly reduced. The construction of a balanced transmission line requires that the two conductors be in close proximity to each other (without an intervening ground or power plane). This means that any transients induced in one conductor of a balanced transmission line will have the same (or nearly the same) transient (with the same magnitude and phase) induced

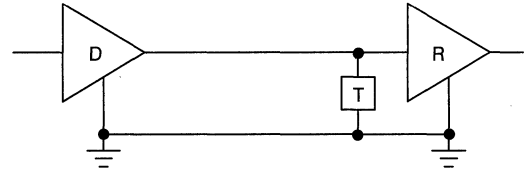


Figure 3. Unbalanced Transmission Line

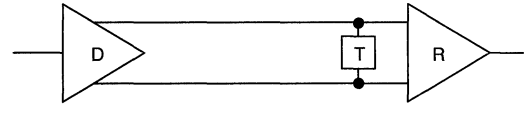


Figure 4. Balanced Transmission Line

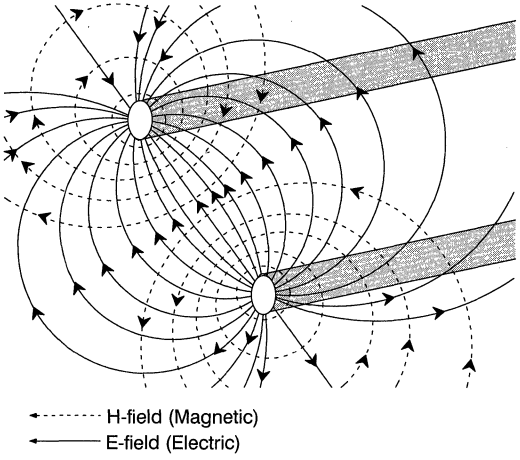


Figure 5. Electric and Magnetic Fields in a Balanced Transmission Line

in the other conductor. This crosstalk is, in effect, a form of common mode noise that (within limits) is rejected by the differential receiver.

In a balanced transmission line, the electric and magnetic fields exist *between* the two driven lines—there is no dynamic current flow in any present ground path. These fields are shown in *Figure 5*. The receiver is implemented as a differential amplifier that operates by comparing the amplitude *difference* between the two received signals.

HOTLink Usage of Transmission Lines

When driving transmission lines with HOTLink, the first selection criteria is usually how far the signals must travel. For very short interconnects, the transmission line is often created using circuit board constructs that allow the high-speed signals to be routed across a card or backplane. For distances greater than a meter, cables of various configurations are generally used instead.

Circuit Board Transmission Lines

Figure 6 shows the cross-sectional construction of the two primary types of circuit-board-based transmission lines. While other configurations are possible, the stripline and microstrip constructions follow standard circuit board manufacturing flows, and thus see the largest industry usage.

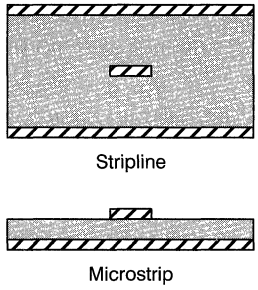


Figure 6. Circuit Board Transmission Lines

These types of transmission lines are used to route high-speed signals from a few centimeters to around a meter of circuit board. They are often routed through connectors as well as backplanes. Because of the relatively short distances used with these types of transmission lines, they are usually considered to be lossless.

Microstrip Transmission Line

Microstrip transmission lines are characterized by having a single strip-conductor spaced above a ground plane by a dielectric. This dielectric is usually the same material used for the remainder of the circuit board.

The key to using such a construct as a transmission line is stability of dimensions. Three dimensions determine the characteristic impedance (Z_O) of the transmission line as shown in *Figure 7*: the width of the trace, the thickness of the trace, and the height of the dielectric.

With standard circuit boards the thickness of the trace is determined by the weight of copper specified for that specific (strip) layer. Standard thicknesses are usually specified in ounces; i.e., 1-ounce

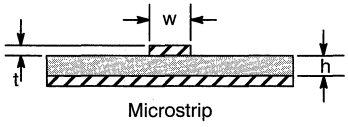


Figure 7. Microstrip Dimensions

copper yields a trace 0.0356 mm (0.0014″) thick. The width of the trace is specified in the artwork used to generate the circuit card, while the height of the trace from the ground plane is determined by the thickness of the laminate specified for the board construction.

A close approximation of the characteristic impedance of a microstrip transmission line may be calculated using *Equation 4*, where ε_r is the relative dielectric constant of the board and w, h, and t are the dimensions shown in *Figure 7*.

$$Z_O = \frac{87}{\sqrt{\varepsilon_r + 1.41}} \ln\left(\frac{5.98h}{0.8w + t}\right) \quad \text{Eq. 4}$$

This equation is an approximation and is not accurate for all ratios of width-to-height-to-thickness. Per experimental observation it does remain accurate (±5%) for width-to-height ratios between 0.1 and 3.0 if the dielectric constant remains in the 1−15 range (Reference 2).

The transfer function for Z_O versus trace width for a microstrip transmission line is shown in *Figure 8*. All curves are based on standard FR4/G10-type laminate with 1-ounce copper. Varying the copper thickness has the least effect on the trace impedance. Going to 2-ounce copper will lower the trace impedance from 1−5%, while changing to 0.5-ounce copper will raise the impedance a similar amount.

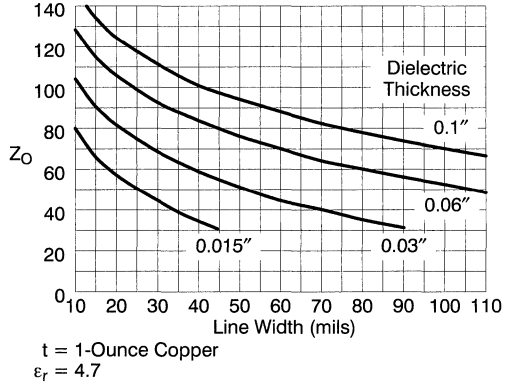


Figure 8. Calculated Impedance vs. Trace Width for Microstrip Transmission Lines

Because of the variation in trace widths caused by etching, it is not advisable to use line widths under 10-mils for controlled impedance transmission lines. As the trace widths get smaller, the variation in line width has a much larger impact on trace impedance.

In a transmission line of this type some of the electromagnetic field propagates in the air above the strip conductor, while the remainder propagates through the circuit board dielectric. Because of this mixed medium, the V_P calculation for a microstrip transmission line (shown here in *Equation 5*) is different from that in *Equation 3* (Reference 2).

$$V_P = \frac{1}{\sqrt{0.475\varepsilon_r + 0.67}} \quad \text{Eq. 5}$$

Stripline Transmission Line

Stripline transmission lines are characterized by having a single strip-conductor spaced between two ground planes by a dielectric. This dielectric is usually the same material used for the remainder of the circuit board.

Just as with a microstrip line, the key to using a stripline construct as a transmission line is stability of dimensions. Three dimensions determine the characteristic impedance (Z_O) of a stripline transmission line as shown in *Figure 9*: the width of the trace, the thickness of the trace, and the height of the dielectric.

A close approximation of the characteristic impedance of a stripline transmission line may be calculated using *Equation 6*, where ε_r is the relative dielectric constant of the board and w, h, and t are the dimensions shown in *Figure 9*.

$$Z_O = \frac{60}{\sqrt{\varepsilon_r}} \ln\left[\frac{4h}{0.67\pi w(0.8 + \frac{t}{w})}\right] \quad \text{Eq. 6}$$

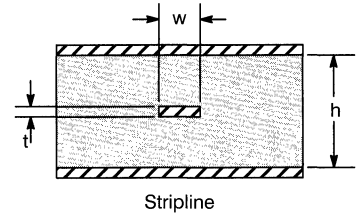


Figure 9. Stripline Dimensions

This equation is also an approximation and is not accurate for all ratios of width-to-height-to-thickness. Per experimental observation it does remain accurate (±5%) when w/(h−t)<0.35 and t/h<0.25 if the dielectric constant remains in the 1−15 range (Reference 2).

The transfer function for Z_O versus trace width for a stripline transmission line is shown in *Figure 10.* All curves are based on standard FR4/G10-type laminate with 1-ounce copper. Varying the thickness of the buried copper trace has the least effect on the trace impedance. Going to 2-ounce copper will lower the trace impedance from 3−8%, while changing to 0.5-ounce copper will raise the impedance from 1−5%.

Unlike a microstrip transmission line, where part of the electromagnetic field propagates in air, in a stripline transmission line the field is bounded by the ground planes and must remain within the circuit board dielectric. This means that the V_P for a stripline transmission line is determined only by the dielectric constant and thus follows the calculation in *Equation 3.*

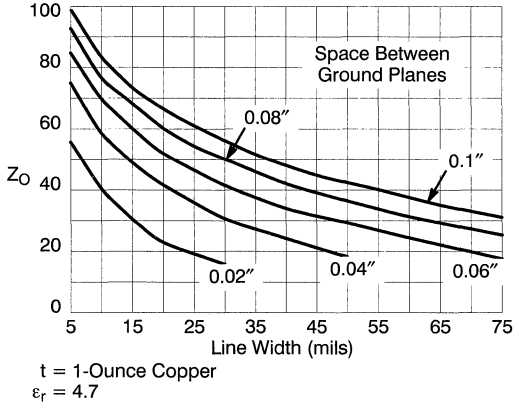


Figure 10. Calculated Impedance vs. Trace Width for Stripline Transmission Lines

Other Circuit Board Concerns

When building microstrip transmission lines, interaction with the circuit board manufacturer is a must. To insure a constant dielectric thickness, the user should verify with their board manufacturer that a double-sided laminate is used (versus the B-stage or pre-preg layers) for this part of the circuit board. These "fill" layers in multilayer circuit boards cannot maintain the same dimensional stability between the strip-trace and the ground (or power) plane.

Verification of the relative dielectric constant should also be done. While often approximated at 4.7 for G10/FR4 substrates, this value can range from 4 to 6.

Unlike a microstrip transmission line, which may be forced onto a two-layer laminate for its construction, a stripline transmission line must, by its very nature, be composed of two separate circuit boards that are then laminated together in a multilayer assembly. This makes it much more difficult to control the dielectric height specification. The multilayer construction also raises the assembled board cost. This often limits the use of stripline transmission lines to those areas of a design that require the additional shielding provided by the embedded strip construction.

Care must also be exercised in the placement position of the strip conductors relative to any significant discontinuities in the ground planes. As a general rule the strip should remain at least 5(w+h) away from the discontinuity for microstrip, and 5(w+ h/2) for stripline; e.g., don't route these types of transmission lines along the edges of cards.

Table 2 lists the relative dielectric constants for a number of common circuit board substrates. This dielectric constant alone determines the V_P (and the propagation delay) for the transmission line.

While it is theoretically possible to create a balanced transmission line on a circuit board, such construction is both difficult and costly due to the geometries involved. Almost all circuit-board-

based transmission lines are unbalanced; i.e., transmitted as a signal relative to ground.

Table 2. Properties of Circuit Board Substrates

Material	Dielectric Constant	Prop Delay (ps/cm)	
		Microstrip	Stripline
G10/FR4	4.7	56.8	72.3
Mylar	5	58.2	74.5
Alumina	9.9	77.2	105
Teflon	2.1	43.0	48.3

For those cases when the added noise immunity or other signal characteristics of a balanced transmission line are desired, the circuit may employ two unbalanced transmission lines that are then examined by the receiver differentially.

If matched delays are necessary in a system (for clock traces, pseudo-differential signals, etc.), do not attempt to route some of the signals as stripline and others as microstrip. The V_P calculations for each of these transmission lines are approximations based on specific dimensions that can vary significantly over manufacturing runs. By selecting either stripline or microstrip for both transmission lines, the manufacturing variations present should affect both transmission lines in similar amounts and thus have minimal effect.

Copper Cable Transmission Lines

Copper cables are generally used either for difficult signal routing (may even be used on a circuit board) or when long distances are involved. They have the advantage of being available in many configurations, with tightly controlled impedances, and allow communications at high bit-rates over hundreds of meters.

All copper cables fall into two categories: coaxial and parallel pair. The principal selection criteria between these two cable types is if the signal is to be transmitted unbalanced (coaxial) or balanced (parallel pair). The cross sectional construction of these two cable types is shown in *Figure 11*.

Coaxial Cables

Coaxial cables are composed of two concentric conductors, maintained at a fixed spacing by a dielectric separator. This type of cable may only be driven in an unbalanced or single-ended form. They are available in flexible, semi-rigid, and rigid configurations, in diameters from 0.25 mm up to around 10 cm. Commercial cables are available in many impedances ranging from 32Ω to 125Ω, with the primary standards being 50Ω, 75Ω, and 93Ω.

The 50Ω standard was developed for the military services in the early 1900s for use in radio broadcast. They needed a cable to feed vertical ground plane omnidirectional antennas which, by construction, had a 50Ω impedance. The 75Ω standard was adopted by the video and telecom industries because this impedance is the most efficient (considering only the voltages, currents, and powers to be driven) for transmission. The 93Ω standard was developed for the instrumentation industry to address their need for a low capacitance cable. Many other construction variants exist (triax, quadrax, etc.) that differ primarily in the number and usage of the outer cable shield (Reference 3).

The characteristic impedance of a coaxial cable is determined by the ratio of its inner conductor to outer conductor diameters as shown in *Figure 12*, and

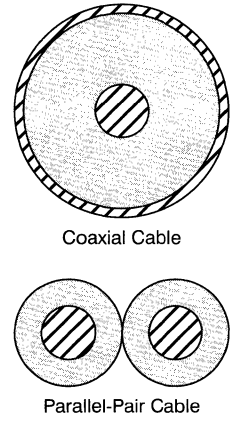


Figure 11. Copper Cable Cross-Sectional Constructions

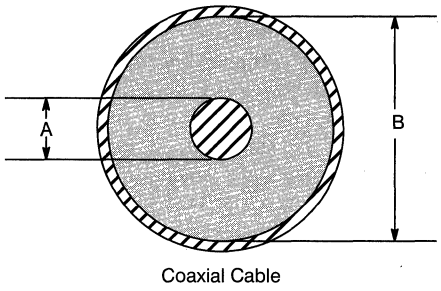


Figure 12. Coaxial Cable Critical Dimensions

the dielectric constant of the spacer material. This relationship is shown in *Equation 7* (Reference 4).

$$Z_O = \frac{138}{\sqrt{\varepsilon_r}} \log_{10} \frac{B}{A} \qquad \text{Eq. 7}$$

The entire electromagnetic field in a coaxial cable propagates through the dielectric (see *Figure 1*). This means that the V_P for a coaxial transmission line is determined only by the dielectric constant and thus follows the calculation in *Equation 3*. A comparison of the propagation velocities of common coaxial cable dielectrics is given in *Table 3* (Reference 5).

Table 3. Propagation Velocity of Dielectrics

Insulation Type	ε_r	V_P Coaxial (%)	Prop Delay (ns/m)
PVC (Standard)	4–6	50–41	6.7–8.2
PVC (Premium)	3–5	58–45	5.8–7.5
Polyethylene	2.27	66	5.02
Polypropylene	2.24	67	4.99
Cellular Polyethylene	1.5	82	4.08
FR Polyethylene	2.5	63	5.27
FEP/TFE Teflon	2.1	69	4.83
Cellular FEP	1.4	85	3.94

Parallel-Pair Cables

Parallel-pair cables are formed from two conductors, each having the same diameter, maintained a fixed distance apart from each other. This distance separation is usually maintained by the insulation around the individual conductors, but other types of spacers are also used.

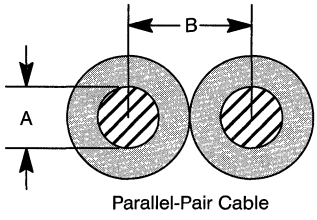


Figure 13. Parallel-Pair Cable Critical Dimensions

While individual coaxial cables may only be driven in a single-ended (unbalanced) connection, parallel-pair cables may be driven either single-ended or differentially. What surprises many people is that the characteristic impedance for the cable is *different* depending on how the line is driven.

Equation 8 (along with the dimensions shown in *Figure 13*) is the standard equation used to calculate the Z_O for a parallel-pair transmission line. What is not usually identified is that this equation is only valid for differentially driven cables. When the exact same cable is driven single-ended (i.e., one line of the pair is a signal ground), the cable impedance is about 25%–35% lower (Reference 6).

$$Z_O = \frac{276}{\sqrt{\varepsilon_r}} \log_{10} \frac{B}{A} \qquad \text{Eq. 8}$$

Equation 8 also makes the assumption that the entire electromagnetic field propagates through the dielectric. Except for those transmission lines that are either air dielectric (open wire) or a specialized construction, the propagation will actually be split across multiple dielectric types and *Equation 8* will not be as accurate.

The V_P of a parallel-pair cable is also usually calculated using *Equation 3*, however the accuracy of this equation (because of the mixed dielectric) will vary depending on cable construction. It will usually be slightly faster than the calculation, which assumes only the physical (non-air) dielectric.

In theory, in a balanced transmission line the electromagnetic fields created around the two parallel conductors are equal in magnitude, but opposite in phase. The total field around such a transmission line has a net field-strength of zero; i.e., the fields

cancel each other out and no energy is radiated. In actuality the two fields do not quite cancel. To do so would require both conductors to occupy the same physical space. To keep radiation to a minimum, the distance between conductors should be kept to no more than 1% of the signal wavelength (Reference 4).

Current balance is also important to minimize radiation. Because the fields generated are based on the currents present in the two conductors, any difference in the magnitude or phase of the driven signals will generate a different electromagnetic field. This difference, because it is not canceled out by the opposing field on the other conductor, radiates energy. This mismatch can be a significant contributor to EMI in a system.

Care must also be exercised in routing the conductors of balanced transmission lines to make sure that adjacent objects do not induce an unbalance into the system. If one of the two conductors is routed close to a ground or other conductor, the shunt capacitance can unbalance the line currents and increase radiation.

Two primary techniques are available to help reduce the interference affects of parallel-pair transmission lines, both from a radiation and from a susceptibility standpoint. The first of these is to twist the two conductors together at a controlled number of twists per unit length. In such a construction, the conductors must radially remain at the same centerline spacing throughout the twists to maintain the transmission line characteristic impedance. Average twist densities are from 1 to 0.1 twists per centimeter.

Twisting the lines together allows magnetic field cancellation and minimizes the affects of other nearby conductors. While the shunt capacitance will still exist, it is now applied in nearly equal amounts to both conductors, maintaining the field balance.

This same twisting also improves immunity to crosstalk in a system. With a true parallel-wire system, the currents induced by the fields present around an adjacent conductor are not always of the exact same magnitude on both conductors of a parallel-pair (due to the physical spacing between the conductors). The twists present in a twisted-pair cable tend to bring both conductors into the same proximity of the noise generating conductor. This not only maintains the field balance in the cable, but also keeps the noise pickup truly common-mode, which can then be canceled by the receiver differential amplifier.

Twisted-pair cables also offer significant immunity to external e-fields (electric) and h-fields (magnetic). Because the signal wavelength is significantly longer than the twist-length on the cable, an external electromagnetic field's influence is spread across each propagating wave in multiple twists of the cable, each of which presents an opposite field intensity. These opposing fields tend to cancel out the affect of the external field.

The other method used to limit interference on parallel-pair conductors is shielding. A shield is an additional conductor surrounding both signal conductors in the parallel-pair. The purpose of this shield is two-fold: to constrain the electromagnetic fields generated by the transmission line, and to isolate external fields from this same transmission line.

Shields

Shields are used to keep what's outside out and what's inside in. How effective they are depends on their construction and how they are used in the system. *Figure 14* shows the construction of a number of different types of cable shields. Shields of these types operate as an electrostatic or Faraday shield. This means that they can block e-fields (electric) but offer only minimal protection from external h-fields (magnetic).

In *Figure 14* the part identified as the cable core could be any of the previously described cable types. In the case of coaxial cables the core, in its simplest form, would consist of a single conductor surrounded by its dielectric spacer, with the shield being the ground return conductor of the transmission line. Other constructions of transmission line cables can actually have multiple shields. In these configurations the cables are usually identified by the names triax (a center conductor, its ground, and an overall isolated shield) and quadrax (a shielded parallel- or twisted-pair cable with an overall isolated shield).

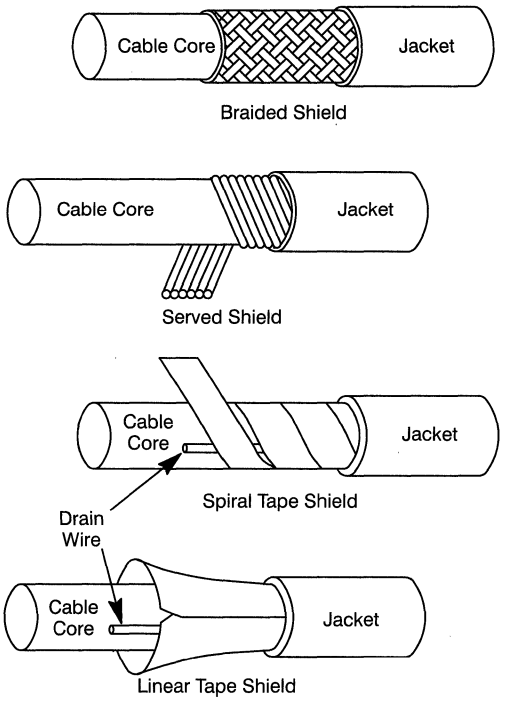


Figure 14. Cable Shield Constructions

A perfect shield would be a seamless metallic tube running the length of the transmission line. Construction of this type is actually used for some forms of coaxial cable known as *hardline*.

For flexible cables, a compromise must be made. This compromise trades off shielding effectiveness for cable flexibility. Now instead of the shield being completely seamless, it has multiple seams that allow the cable to bend. These shields are made of either braided or spirally wrapped (served) layers of fine-gauge copper (sometimes aluminum if used as a secondary shield) wire, or spiral or linear-wrapped metallic tape.

Braided shields consist of multiple groups of 34- to 40-AWG copper wire, braided together in a circular fashion around the core section of the cable. These strands may be bare copper but are often tin or silver plated. Shields of this type are rated in terms of braid coverage; i.e., how close to a seamless tube they get. Because of the high-frequencies present in a HOTLink-based serial connection, shield coverage should be a minimum of 85%. As a rule of thumb, if any dielectric is visible through the braid, there is insufficient coverage.

Served shields consist of the same fine-gauge copper wire wrapped in a continuous spiral around the cable core for the length of the cable. These strands may be tin plated, but are generally not silver plated. Cables of this construction should never be used for frequencies above 10 MHz because the spiral-wrap construction contains many long spiral gaps (especially near cable bends) that will leak EMI.

Metallic-tape shields are often used for high-frequency signaling because of the high degree of shielding coverage they provide. The metallic tape is made from either thin aluminum foil, or a plastic strip that is coated with aluminum on one or both sides. To allow termination of the shield at either end of the cable, and to make sure that each wrap of the shield tape is shorted together, these cables usually include an uninsulated drain wire that is in direct contact with the tape shield for its entire length.

Shields are often combined for even better shielding. Often a tape-shield will be covered by a braided shield. In this configuration the drain wire is eliminated because the braided shield performs the same function.

Shield Transfer Impedance

One of the best ways to judge a shield's effectiveness is by its transfer impedance. This is a specification that relates how currents on one surface of a shield generate a voltage drop on the *other* surface of the shield. It is usually specified in mΩ/meter of cable. The effectiveness of any shield is directly proportional to its transfer impedance. As the term impedance implies, this is a frequency sensitive parameter.

Because of their high DC resistance, aluminum-based tape shields do not fare very well in this measurement. Braided and served shields do much better due to their low-resistance copper construction. The best results are achieved by the combination of tape and a braided or served shield. *Figure 15* shows how shield construction effects transfer impedance.

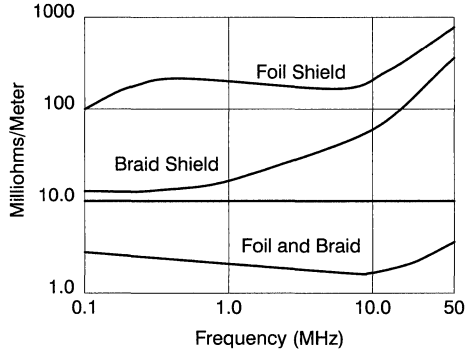


Figure 15. Shield Transfer Impedance

Electromagnetic Compatibility

Shields are also necessary in many systems to allow equipment to meet various national and international electromagnetic compatibility (EMC) requirements. EMC deals with how much electromagnetic energy a piece of equipment is allowed to radiate, as well as how much external energy it must tolerate. Specific limits for both of these are set by a number of different international governing bodies. In the United States the limits for compatibility are set by the Federal Communications Commission (FCC) in Part-15 of their regulations. In Europe, the Common Market countries are now governed by a single EMC Directive in standards EN55022, EN55014, and EN60555-2, developed by CENELEC (Committee for European Electrotechnical Standardization). These standards deal with any digital equipment operating with any clocks or switching present at greater than a 9-kHz rate, and cover all frequencies up to 40 GHz.

For digital equipment, different limits are set for both radiated emissions as well as susceptibility depending on the target customer for the equipment. Equipment intended only for use in an industrial or business environment is classified as Class-A, while equipment that may be used in the home is classified as Class-B. The radiated emission limits for Class-B are shown in *Figure 16* (Reference 9).

Under both of these classifications, it is necessary to test up to the 5th harmonic of the highest frequency clock present in the system. For HOTLink-based systems this could require testing up to 1.7 GHz.

Coupling to Copper

There are three primary ways of coupling HOTLink to copper media: direct coupled, capacitor coupled, and transformer coupled. Each of these methods has different bias and termination requirements for the high-speed ECL signals.

Direct Coupling

Direct coupling is where a DC path exists between the HOTLink Transmitter and Receiver on the high-speed serial interface. This coupling is used for those cases where both the transmitter and receiver operate from the same power supply and are in (relatively) close proximity to each other.

There are many subsets within this direct coupled area. These are differentiated by how far the signal must travel and the quantity of loads present.

Direct Coupled: <3 cm Length

For link distances under 3 cm, the serial signals do not have to be treated like transmission lines. In these cases all that is necessary is to bias the ECL signals so that they may properly switch. Because the transmission distance is so short, the signal may be assumed to be digital in nature. This allows the analog transmission concerns of longer distances to

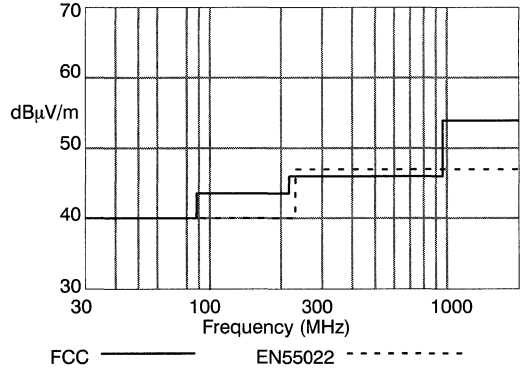


Figure 16. FCC and EN 55022 Class-B Emission Limits at 3 Meters

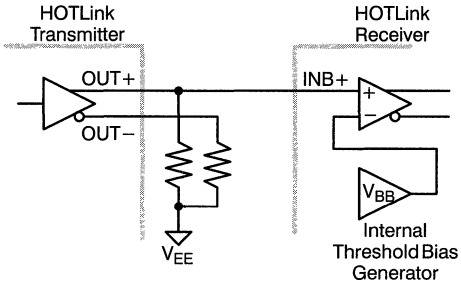


Figure 17. Single-Ended Connection

be minimized. A single-ended connection schematic is shown in *Figure 17*, while a differential connection is shown in *Figure 18*. Typical values for the pull-down loads are from 250Ω to 510Ω.

Because the HOTLink Receiver does not provide an external V_{BB} reference, a single-ended connection may only be implemented using the INB+ input of the receiver. A differential connection may be implemented using either of the INA± or INB± differential inputs.

The ECL bias in both of these configurations is implemented with a single pull-down to V_{EE} on each driver output. While this bias configuration does generate more jitter than either a Thévenin or Y-bias, the amount is well under the jitter tolerance limits of the HOTLink Receiver for all supported frequencies.

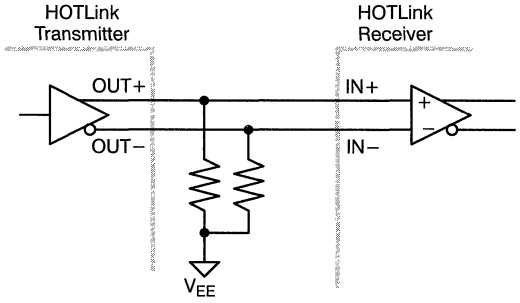


Figure 18. Differential Connection

Direct Coupled: From 3 cm to 1 m Length

Once the length of the connection becomes longer than 3 cm, the connection *must* be treated as a transmission line. This requires a termination network at the end of the transmission line. Because the connection is DC coupled, the termination network may also be used to bias the ECL output.

Unlike the previously described bias-only pull-down load, the network here must actually match the impedance of the transmission line. If it does not, a portion of the signal delivered into the transmission line is reflected off the termination and returned to the source. The amount of the reflection is determined by the voltage-reflection coefficient of the load, ρ_L, which is calculated using *Equation 9*.

$$\rho_L = \frac{\textit{reflected voltage}}{\textit{incident voltage}} = \frac{R_L - Z_O}{R_L + Z_O} \qquad \text{Eq. 9}$$

Since this type of connection is only terminated at the destination, any signal reflected from the load will be returned to the source. However, because the source is *not* impedance matched to the transmission line ($\rho_L \cong 1$), a large portion of the reflected signal it sees will again be reflected back down the transmission line.

A reflection of this type will continue to travel back and forth between the two ends of the transmission line, being attenuated in amplitude both by the transmission line losses (very low for these short lines) and by the amount of signal absorbed in the terminations.

Figure 19 shows a single-ended connection using a Thévenin bias network. This network is sized for termination to V_{CC} − 2V of a 50Ω transmission line, and should be changed if other impedance transmis-

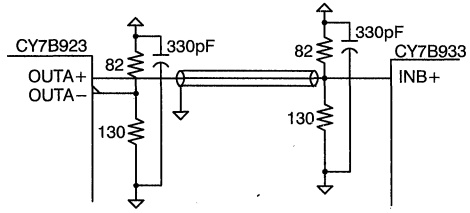


Figure 19. Direct-Coupled, Single-Ended Interface

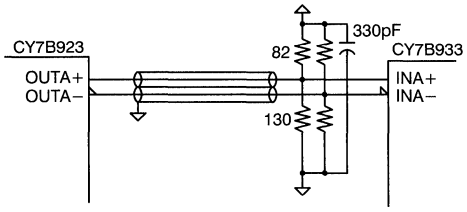


Figure 20. Direct-Coupled, Differential Receiver Interface

sion lines are used. A similar network is added to the OUTA− driver to keep a matched load on the differential driver. While shown in the schematic as a coaxial line, this would in most cases be implemented either as microstrip or stripline. Just as in *Figure 17*, the INB+ receiver is used for the single-ended connection.

When implemented with two transmission lines (as shown in *Figure 20*), the signals may be examined differentially by the receiver. While not a true balanced transmission system, this configuration doubles the noise immunity of the single-ended configuration.

This type of connection is often called a balanced transmission line, but it is not. What actually exists are two single-ended (unbalanced) transmission lines that are examined differentially. Because the electromagnetic waves propagate independently down the two transmission lines, it is very important to make sure that both lines are the same electrical length from the driver to the receiver to allow the two signals to arrive in the same phase relationship they were sent.

Direct-Coupled Bus

A common usage for HOTLink is as a data-mover on a backplane. In this configuration, the HOTLink Transmitters and Receivers are used to replace some of the wide buses on the backplane, along with their associated drivers, receivers, and connector pins. This usually provides a lower cost, lower power, and more reliable solution than the parallel interface it replaces.

Single-Ended Bus

A bus of this type utilizes the wired-OR capability of ECL outputs to allow multiple sources on a common bus. Transmission line terminations are still necessary, and in fact must now be placed at *both* ends of the transmission line. *Figure 21* shows a sample configuration of a single-ended multiple source and destination bus.

In this configuration, a HOTLink Transmitter and Receiver are located on a card plugged into a backplane. All the receivers are enabled at all times, and the transmitters are controlled using the FOTO signal such that only one of them is allowed to transmit at a time. Because of the single-ended operation of the bus, the INB+ input of the receiver should be used for serial data input.

The transmission line must be terminated at both ends to allow signals to be driven at any point along the transmission line. When the signal is launched into the line it effectively splits, with part of the signal traveling in each direction on the line. When the signal reaches the end of the transmission line it is absorbed into the termination networks. This double termination places a higher current burden on the driver. It sees two 100Ω transmission lines in parallel, which present a load of 50Ω.

The complementary output of each differential driver must also see the same load as the true output to provide a balanced load for the driver. This requires adding a 50Ω Thévenin bias network for each driver present.

While implemented here with a 100Ω transmission line, other impedances may also be used. The lowest recommended transmission line impedance is 50Ω. This presents a 25Ω effective load on each attached driver.

The physical implementation of a single-ended bus does have a few limitations. One of these is how many drivers/receivers can actually be on the bus. This is not a driver current limitation (HOTLink input currents are $\ll$ 1 mA), but is instead due to capacitive and stub effects. Each card on the backplane adds from 3-pF to 10-pF of capacitance to the bus. This added capacitance slows down the rising

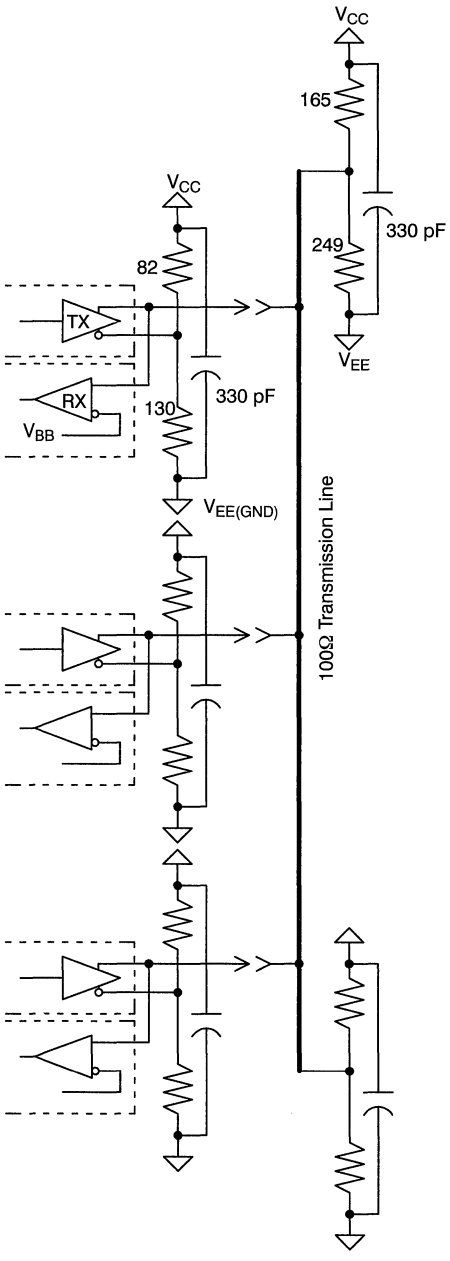


Figure 21. Single-Ended, Multi-Source Bus

and falling edges of the signals. When operating in a single-ended environment, the maximum number of driver/receiver pairs should be limited to 20.

The physical placement of each driver/receiver pair is also critical to proper operation. Due to the construction of a backpanel and its associated cards, each driver/receiver pair also adds a stub to the transmission line. The longer each stub, the more reflection/distortion it will cause on the backplane. These reflections are limited by placing the driver/receiver directly adjacent to the board/backplane connector. The signal route from the connector to the driver/receiver pair should be kept to no more than two centimeters in length.

Differential Bus

A single-ended bus of this type may be reliably used when the system noise is understood and within the margins of a single-ended ECL connection. For systems with more loads, more noise present, or those that may be exposed to large external noise sources, the bus may be implemented in a differential form. This is not a true balanced transmission line because two separate transmission lines are used; i.e., they do not share a common electromagnetic field.

In the single-ended bus implementation, bus access is controlled using the HOTLink Transmitter FOTO pin. The FOTO pin was designed to disable the light output of optical modules by driving a differential logic-0 (OUT+=LOW, OUT−=HIGH) when the FOTO input is HIGH. Because the OUT− pin is still sourcing current when FOTO is HIGH, access for a differential bus must be controlled externally. This requires the addition of an external ECL multiplexer or differential driver with output disable capability, as shown in *Figure 22*. This driver operates by effectively disabling *both* sides of the differential driver from a single control input.

The biggest problem in implementing such a structure is that true differential ECL multiplexers are rare, and those capable of disabling both outputs are fewer still. This function may be created from separate gates (requires two ECL gates for each differential driver present). Being separate gates, these drivers also do not maintain the close current balance normally present in a true differential driver.

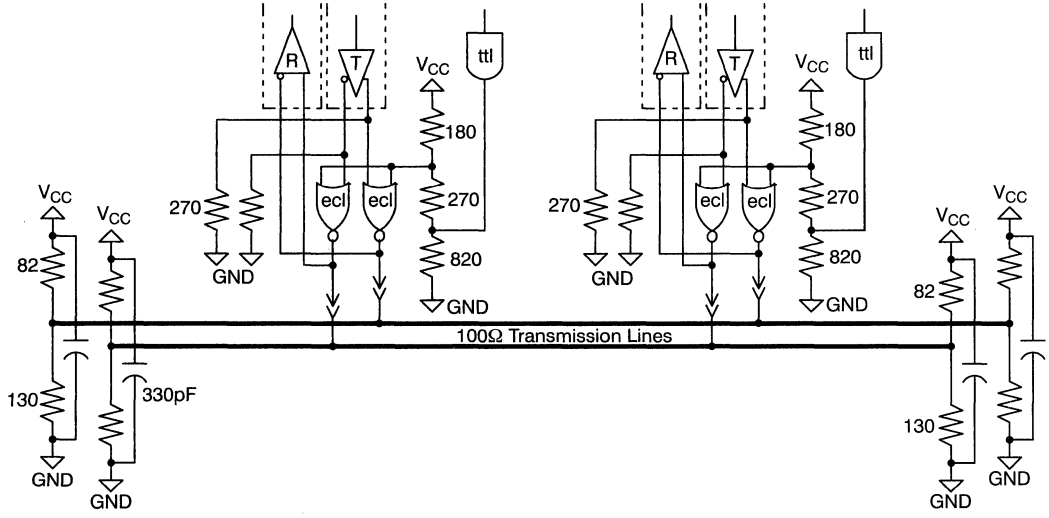


Figure 22. Differential, Multi-Source Bus

To keep delays and currents as matched as possible both gates should be in the same physical package.

These ECL parts are operated in PECL mode; i.e., they use the same V_{CC} and ground as the HOTLink Transmitter and Receiver. Unlike the HOTLink Receiver A/$\overline{B}$ select pin (an ECL input), which may be controlled from a TTL environment using only two external resistors, these external ECL parts must use a three resistor divider. The third resistor is necessary to limit the V_{IH} of the ECL input to no more than V_{CC} – 0.6V.

Some care must be exercised when selecting these external ECL parts. Because of the switching speeds present on the serial interface (>150 MHz) these parts must be 100K ECL or faster. In addition, because the connections between the HOTLink transmitter and these parts are effectively single-ended connections, the external ECL gates must also be temperature compensated to maintain noise margins.

One final concern deals with drive current. Unlike the HOTLink Transmitter, which can drive 25Ω loads, most ECL drivers can only handle 50Ω loads. If the backplane transmission line impedance is less than 100Ω, special bus drivers (e.g., F100123) or drivers with multiple outputs (e.g., F100313 with outputs tied in parallel) must be used to provide the necessary current. If these parts have differential outputs, the unused (complement) outputs should be attached to bias networks to provide a similar load as that seen by the used (true) output of the driver.

Capacitive Coupling

Capacitive coupling may be used for those connections where some reference difference may exist between the source (transmitter) and destination (receiver). This difference may be planned (e.g., true ECL communicating with PECL), or merely anticipated (e.g., possible ground or V_{CC} differences). In both of these cases the capacitor is used to block the DC signal component while allowing the AC components to propagate to the receiver.

This capacitively coupled interface is not recommended for cabling systems that leave a cabinet or extend for more than a few meters. This is primarily due to

- Limited voltage breakdown of the coupling capacitors under ESD situations
- ESD susceptibility of the receiver due to transients induced in the cable

- Limited common-mode rejection at the receiver end

In a capacitive-coupled system, such as that shown in *Figure 23*, a bias network is still necessary at the driver to allow the output to switch. The preferred location for the DC-block capacitors is adjacent to the transmitter, immediately after the output bias network. This location is necessary due to the reactive nature of capacitors.

At the receiver end of the transmission line, the line must be terminated in its characteristic impedance. This is implemented using the two 50Ω resistors in *Figure 23*.

In addition to terminating the transmission line, the receive end must perform a DC restoration to place the received signals within the normal operating range of the HOTLink PECL receiver. This is done using a voltage-divider network.

In this configuration, the receiver reference point is set slightly different from that of a standard ECL receiver. Part of this is due to the HOTLink Receiver being designed for operation at +5V rather than −5.2V or −4.5V. The other is that the HOTLink Receiver has a wider common-mode range than standard 100K ECL parts. To allow operation over the widest range of signal conditions the external bias network on the receive end of the transmission line is set to the center of the HOTLink Receiver 3V common-mode range at V_{CC} − 1.5V.

While it is possible to bias and terminate the differential inputs with two Thévenin networks, this should not be done. The tolerance differences, even using 1% resistors, are enough to introduce offsets of >50 mV between the inputs. This offset will lower the system noise margin and increase the duty cycle distortion (DCD) jitter in the link. The bypass cap is used to keep the bias point stable by supplying current during any minor transients.

The transmission line in *Figure 23* is shown as two 50Ω unbalanced transmission lines. If the interconnect is implemented using microstrip, stripline, or coaxial cables, this is the type of connection that actually exists. In this dual-unbalanced connection, the same equal-length restrictions of direct-coupled interfaces still exist.

By replacing the two unbalanced transmission lines with a single balanced transmission line (unshielded twisted-pair, shielded twisted-pair, or twinax), it is possible to remove most of the equal-length concern of the conductors in the transmission line. In this configuration, the transmitter and receiver circuits remain the same, but the mode of propagation is now balanced (i.e., conductor-to-conductor, ground path not required).

A capacitively-coupled link may also be operated using a single piece of coaxial cable, but only with single-ended drive and reception. This requires giving up half of the received signal amplitude (only one driver is used), and connecting the INA− receiver input directly to the reference voltage.

DC-Block Capacitor

While the desired affect of a DC-block capacitor is to block all DC and pass all AC signal components (without loss), real life components don't operate in this fashion. Instead, a real capacitor blocks *most* of the DC, and passes frequency selective amounts of the AC signal components.

An equivalent model of a real capacitor is shown in *Figure 24*. In addition to the pure capacitance C, a number of parasitic resistive and inductive elements

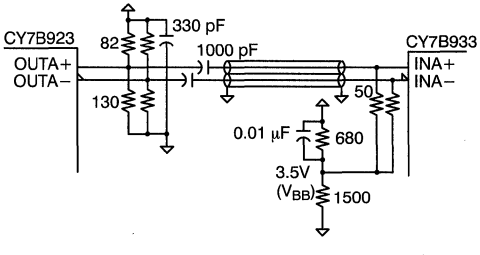


Figure 23. Capacitive-Coupled, Copper Interface

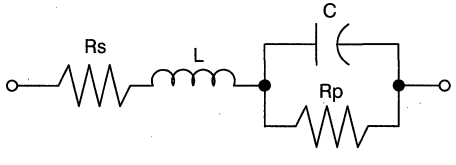


Figure 24. Capacitor Equivalent Model

are also present. These parasitic elements determine the amount of leakage current, the ESR (equivalent series resistance), and where (in terms of frequency) the capacitor stops acting like a capacitor, and starts acting like an inductor. This frequency point is called the series-resonant frequency of the capacitor.

The very small amount of DC current passed through a capacitor is called leakage current. For most designs this leakage is so small that it will be undetectable relative to the AC signal components. The amount of AC signal passed varies with frequency, and is limited on the low end of the frequency spectrum by capacitance, and on the high end by parasitic inductance. This gives a capacitor a passband characteristic.

The amount of AC signal that is passed is controlled by the reactive characteristics of the capacitor, relative to that of the attached transmission line. For those frequencies below the series-resonant frequency of the capacitor, the reactance can be calculated using *Equation 10*. To allow efficient signal transfer, the X_C should be kept below 1Ω for the frequencies of interest.

$$X_C = \frac{1}{2\pi fC} \qquad \text{Eq. 10}$$

Because the reactance of a capacitor varies greatly with frequency, placement of such a component between the receive end of the transmission line and its termination network is not recommended. This is due to the reflections that would be caused by not terminating the transmission line in its characteristic impedance at all frequencies.

Placing such a capacitor directly adjacent to the driver removes much of this reflection problem. The reflections will still occur, however, they are absorbed as part of the rise and fall times of the source signal.

Good low-loss, RF-grade capacitors should be used for this application. These parts are available in many different case types and voltage ratings. The capacitors used must be able to withstand not just the voltage of the signals sent, but any DC difference between the transmitter and receiver and the maximum ESD expected. A typical 1000-pF 50-WV C0G/NP0 capacitor would be available in an 0805 surface mount case size (0.08″L x 0.05″W x 0.02″H). For on-board applications a 50-WV rating should be sufficient. While capacitors with much higher breakdown voltages are available, both cost and space make their use prohibitive. This same 1000-pF C0G capacitor at 5-kV breakdown is almost a half cubic inch in size (Reference 7).

Transformer Coupling

Transformer coupling is the preferred method for attachment to copper cables that extend for more than a few meters, or are operated between enclosures. Transformers have multiple advantages in copper-based interfaces. They provide:

- High primary-to-secondary isolation
- Common-mode cancelation
- Balanced-to-unbalanced conversion

The transformer is similar to a capacitor in that it also has passband characteristics, limiting both low and high frequency operation. Proper selection of a coupling transformer allows passing of the frequencies necessary for HOTLink serial communications.

The configuration shown in *Figure 25* uses only a single transformer, and either 150Ω twinax or twisted pair as the transmission line. This can be done because the transmission system remains balanced end-to-end. Here the primary functions of the transformer are to provide isolation and common-mode cancelation.

In a single transformer configuration the transformer should be placed at the source end of the cable. Unlike the HOTLink differential receiver, which has a full 3V common mode range, an ECL output

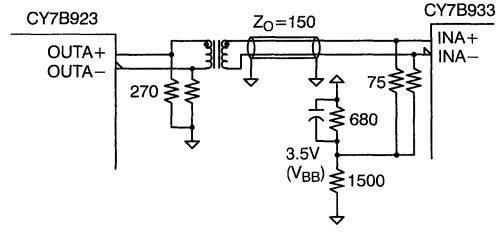


Figure 25. Transformer-Coupled, Copper Interface

(when sourcing a zero or LOW-level) will respond to high-going signals picked up on the transmission line. If a shield is present, it should be grounded at one or both ends to an earth or chassis (not signal) ground.

The transmitter shunt-bias network shown in *Figure 25* was selected to provide the maximum signal amplitude into the transmission line, rather than the most symmetrical edges. This configuration gives the highest signal-to-noise ratio at the receiver, but has different slopes of the rise and fall times at the transmitter.

These asymmetric rise and fall times do not add to the system jitter. Instead, the true and complement outputs combine in the transformer to provide a single signal with symmetrical rise and fall times. This insures matched transmission line currents for balanced transmission lines. This bias arrangement also the has the advantage of delivering the entire transmitter output signal swing into the transformer, rather than part into the transformer and part into the bias network. In a standard Thévenin bias or bias to V_{TT}, the source signal amplitude divides across the load (transformer) and the bias network, causing a significant amplitude loss.

This transformer-coupled configuration has many similarities to the capacitively coupled interface. It still provides DC isolation between the HOTLink Transmitter and Receiver, and requires the V_{BB} bias (DC-restoration) and termination network at the receiver.

In *Figure 26* a second transformer is added to the transmission system at the destination end of the cable. This configuration allows use of either balanced or unbalanced (coaxial) transmission lines. The configuration shown here is a 75Ω coaxial cable system. Here, the first transformer is used for balanced-to-unbalanced conversion, while the second transformer provides unbalanced-to-balanced conversion. With transformers at both ends of the cable, much larger amounts of common-mode noise may also be handled.

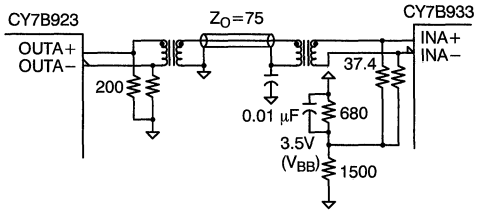


Figure 26. Dual Transformer-Coupled, Copper Interface

The size of the transmitter bias resistors are reduced here to handle the larger current requirements of the load. When driving a common load from a differential source, each driver sees a load impedance of half the actual load present. With a 75Ω cable present each driver sees a 37.5Ω load.

Quantitative Interface Comparison

The transformer-coupled interface is the only one recommended for all cable lengths and types. This configuration operates equally as well with very short (<1 meter) lengths as it does with tens or hundreds of meters. Numerous configurations of transformer coupling and biasing were evaluated to determine both how to best configure a HOTLink-to-transformer interface, and to find out how cable impedance affects these configurations.

Test Equipment

The following equipment was used for the different evaluations:

- HP54100D 1-GHz Bandwidth Digital Sampling Scope
- HP8091A Rate Generator
- HP10240B DC Blocking Capacitor
- HP54002A 50Ω Pods
- Philips PM8919/09 500Ω 10:1 Probes (1.5-GHz Bandwidth)
- Pulse Engineering Transformers
- Cypress CY9266−C HOTLink Evaluation Boards

The primary goals of this testing were to determine how ECL operates when driving transformers, and what cable/coupling methods provide the best signal characteristics.

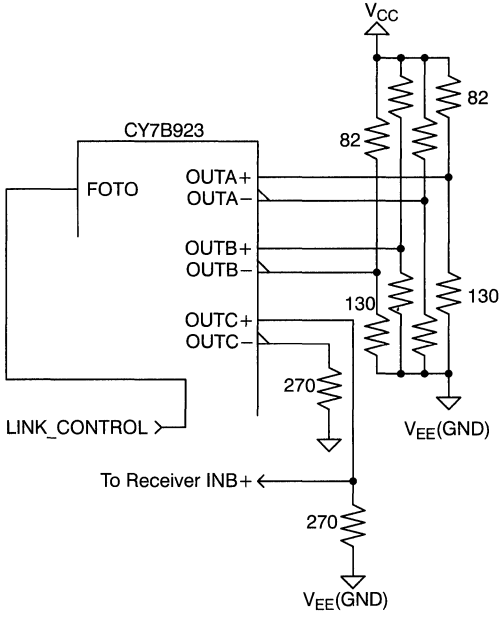


Figure 27. Baseline Test Configuration

To get a good baseline for the following measurements, a HOTLink CY7B923 Transmitter was connected as shown in *Figure 27*. Measurements were made at the OUTB+ pin of the transmitter with the CY7B923 receiving a 25-MHz TTL clock. This clock is up-multiplied by ten inside the HOTLink Transmitter to generate a serial bit-time of 4 ns.

The baseline waveforms for this configuration are shown in *Figure 28*. The top trace shows the TTL-level clock into pin 21 of the transmitter, while the lower trace shows the PECL-level signal on pin 28. Both enable signals on the transmitter ($\overline{ENN}$ and $\overline{ENA}$) are disabled, causing the part to generate a continuous stream of alternating disparity K28.5s. This pattern is good for evaluating serial links because it contains the four combinations of 1s and 0s necessary to test the characteristics of an 8B/10B code.

At this resolution it is difficult to see any real detail other than amplitude and period. To see the critical edge jitter it is necessary to zoom in on the rising and falling edges of the data. This is shown in *Figure 29*.

Figure 28. Baseline Clock and Data

Here the scope sweep rate has been increased by a factor of 100, going from 10 ns/division to 100 ps/division. The data crossover at the center of the figure is approximately 100 ps wide.

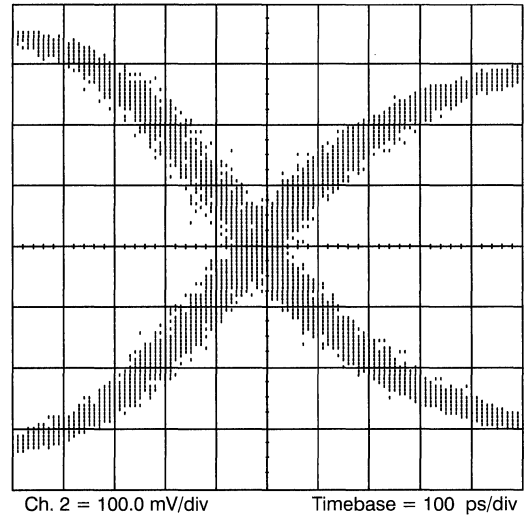


Figure 29. Baseline Jitter

This 100 ps should not be assumed to be the output jitter of the HOTLink Transmitter (it is substantially less than this). It does not take into account the trigger accuracy of the scope, any jitter present in the trigger waveform, or any power supply ripple that the scope may view as additional jitter. However, since all the following measurements are taken with the same set-up and under similar trigger accuracy conditions, this value can be used to provide relative comparisons of different types of media and coupling.

Test Set-Up

The test set-up is shown in *Figure 30*. Low-impedance (500Ω) probes were used for all the high-frequency measurements. These probes, when combined with the scope amplifier, provide a measurement bandwidth of approximately 900 MHz. The probe impedance was factored into the bias and termination networks (where possible) to maintain the desired impedances.

All probe connections were made using shielded probe-tip adapters to eliminate any measurement errors caused by probe ground-lead length.

All cable tests were performed using a single 30.4-meter segment (100 feet) of the specified cable. For those tests performed with a cable length of zero, the same test set-up as that shown in *Figure 30* was used, except that the termination resistor was placed directly on the output (secondary) of the coupling transformer.

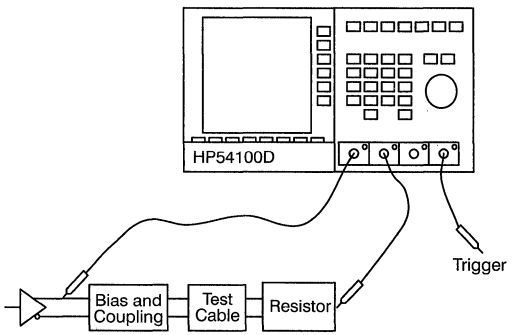


Figure 30. Test Set-Up

Test Configurations

The following test configurations were selected to determine how best to couple to coaxial media using transformers. Additional tests were added to either prove or disprove specific assumptions made in early ANSI Fibre Channel documents about how to couple using transformers. The selected configurations were:

- Thévenin bias, direct-coupled to transformer
- Thévenin bias, AC-coupled to transformer
- Transformer core saturation test
- Shunt bias, direct-coupled to transformer
- Shunt bias, high-frequency AC-coupled to transformer
- Single output, Thévenin bias, direct-coupled
- Single output, Thévenin bias, high-frequency AC bypass
- Single output, Thévenin bias, low-frequency AC bypass
- Dual transformers

These different configurations (where applicable) were tested with three different impedance coaxial cables:

- 50Ω—RG58 (Belden 8219)
- 75Ω—RG59 (Belden 9259)
- 93Ω—RG62 (Belden 9269)

These specific cables were chosen because they provide the three primary cable impedances in a similar category of cable; i.e., they are all made with similar diameters and dielectric materials. This allows a better comparison to be made of the affect of cable impedance on jitter and attenuation.

Thévenin Bias, Direct Coupled

The equivalent circuit for a Thévenin Bias differential driver, directly coupled to a transformer, is shown in *Figure 31*. At first glance this may appear to be the best way to couple a cable through a transformer. The bias voltage here is set by the pull-up/pull-down resistor ratio.

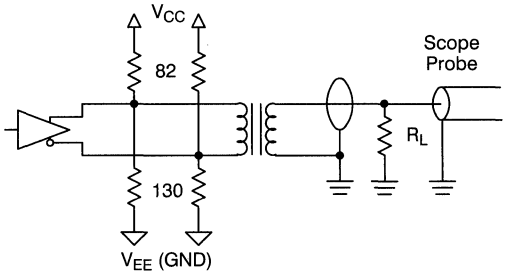


Figure 31. Thévenin Bias, Direct-Coupled

Figure 32 shows the output of one driver on the top trace, and the output of the transformer secondary (when connected to a 50Ω resistive load) on the bottom trace. The primary observation to be made here is that the transformer secondary amplitude is almost equal to that of a single ECL driver. Since two drivers are actually present (differential drive), half of the signal is being lost somewhere.

Figure 33 shows the results when the load on the transformer was changed from 50Ω to 75Ω. Here the driver amplitude remains the same, while the secondary amplitude increases by approximately 50%.

Figure 34 shows the results with a 93Ω resistive load. Now only a small improvement in output amplitude

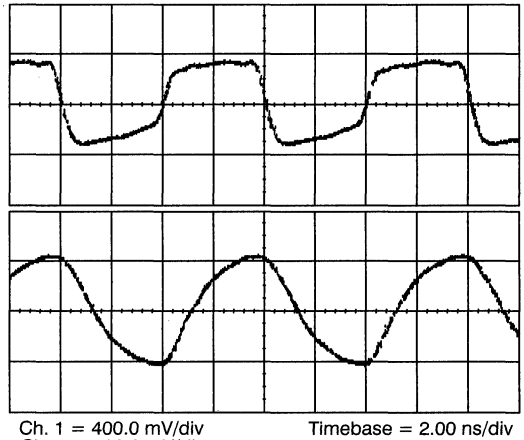


Figure 32. Thévenin Bias, Direct-Coupled, No Cable, 50Ω Load

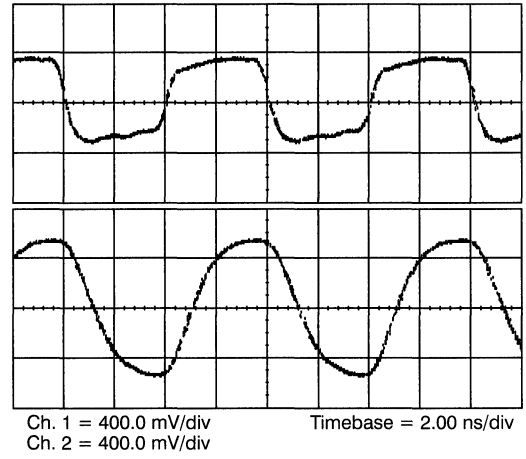


Figure 33. Thévenin Bias, Direct-Coupled, No Cable, 75Ω Load

is seen, while the driver output becomes much closer to a square wave.

The reason for these changes in output voltage with the different loads can be seen in *Figure 35*. Here the Thévenin bias network is converted into a resistor to specific bias voltage. Under DC conditions, the impedance of the transformer primary approaches zero, while under AC conditions the impedance of the primary reflects that present on the secondary.

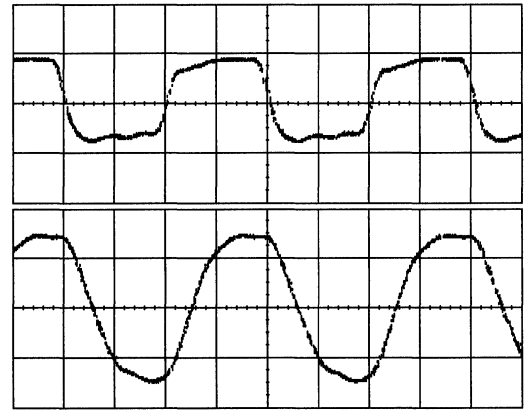


Figure 34. Thévenin Bias, Direct-Coupled, No Cable, 93Ω Load

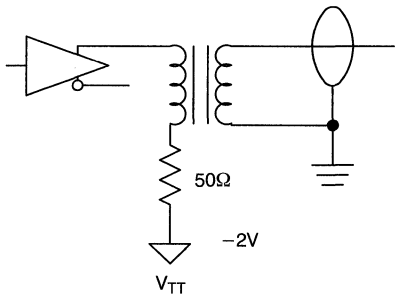
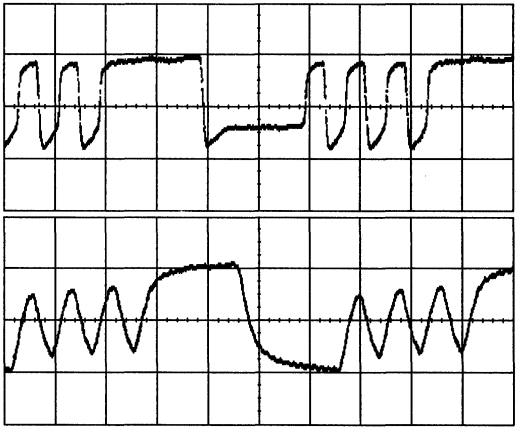


Figure 35. Thévenin Bias Equivalent Circuit

Placing a 50Ω load on the transformer secondary is equivalent to replacing the transformer primary with a 50Ω load. Because the Thévenin bias network is effectively in series with the primary, a voltage divider is created. Since both drivers are switching, half the amplitude of both of them is delivered to the load. With other load impedances, other divider ratios exist. The net effect of this type of biasing is that higher load impedances receive larger amounts of the total source signal amplitude.

Thévenin Bias with Cables

Other affects can be seen when a terminated cable is attached to the transformer secondary instead of just a resistive load. *Figure 36* again shows the driver



Ch. 1 = 400.0 mV/div
Ch. 2 = 400.0 mV/div
Timebase = 10.0 ns/div

Figure 36. Thévenin Bias, Direct-Coupled, with 50Ω Cable

signal on the top trace, and the signal present at the end of 30.4 meters of RG58 cable (50Ω) on the bottom trace. To see the effect of the run length limit of the 8B/10B code, a different pattern was selected that contains both long (5 zeros, 5 ones) and short (single-bit) pulses.

At the end of the cable (shown on the lower trace) the signal is quite different. Now the individual bit-transitions no longer remain centered vertically around the receiver threshold (center line of the lower waveform). This is due to a small DC offset built-up in the cable during the long-0 and long-1 pulses. During these long pulses, the transmission line has time to charge/discharge to near its maximum potential. During the shorter intervals, there is not sufficient time to fully charge or discharge the line. Under these conditions the transmission line is considered a long time-constant line.

Because the *dv/dt* rate for all transitions is effectively the same (regardless of the starting voltage), while the voltage change necessary to reach the receiver threshold is not, these long and short pulses are received shifted in time from nominal. This time shift is viewed at the receiver as a form of jitter called data-dependent jitter (DDJ).

As the length of the cable is increased, this difference in ending voltage between long and short transitions continues to increase. At some length of cable this difference becomes so great that the short transitions no longer cross the receiver threshold and the link becomes unusable. DDJ is one of the primary length-limiting factors of a copper-cable-based link.

Figure 37 shows the same signal as the bottom trace of *Figure 36*. The triggering and timebase have been changed to allow viewing of the individual bits in an overlay format called an "eye" pattern. The normal viewing of eye patterns has the eye opening (marked with the vertical arrow) in the center of the screen. This is used to see how large this opening is relative to a single bit time. The eye patterns shown in this (and following) figure is slightly time shifted, to allow central viewing of the signal crossing area.

Figure 37 shows that the maximum usable amplitude of the eye is around 350 mV (marked with the vertical

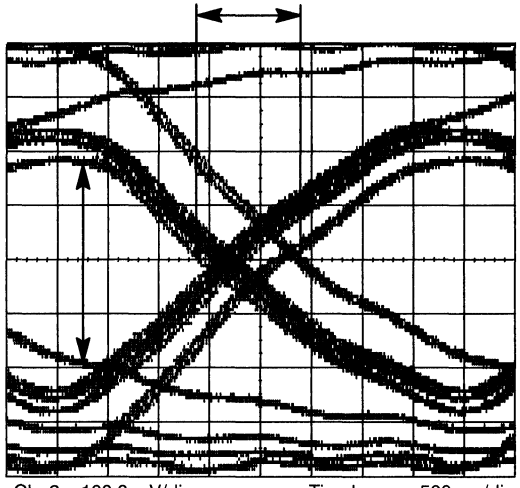


Figure 37. Eye Diagram, Thévenin Bias, Direct-Coupled, with 50Ω Cable

arrows). The jitter per bit (marked by the horizontal arrows) is around 1000 ps (25% of a single bit).

In *Figure 38,* this same configuration is tested using a 75Ω cable and termination. The signal amplitude at the end of the cable (bottom trace) has increased significantly from that of the 50Ω system. Taken as a percentage of the signal delivered to the destination, there is much less variation of peak signal amplitude from the short to long transitions.

Figure 39 shows the eye diagram for this 75Ω system. The usable amplitude here has increased to almost 600 mV; a 70% improvement over the 50Ω system. The amount of jitter present has also been substantially reduced, going to 700 ps. This is about 17% of a bit time.

Figures 40 and *41* show the source and destination signals for a 93Ω system. The signal at the end of the cable has increased again up to 700 mV, while the jitter has been reduced to 500 ps (12%).

By comparing these three systems in *Table 4*, certain relationships become apparent. First, that as the

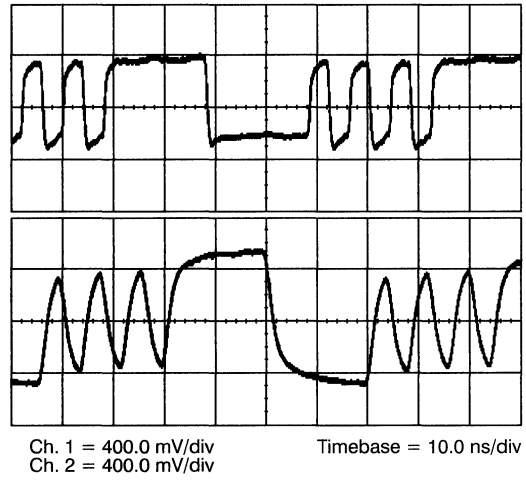


Figure 38. Thévenin Bias, Direct-Coupled, with 75Ω Cable

cable impedance is increased, the signal amplitude delivered to the load is also increased. This amplitude increase also provides a better signal-to-noise ratio (SNR) at the receiver.

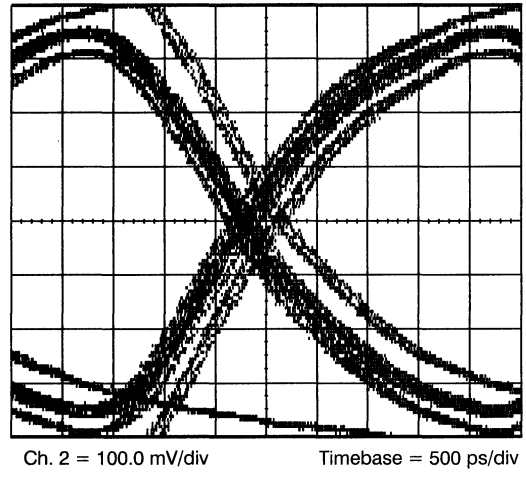


Figure 39. Eye Diagram, Thévenin Bias, Direct-Coupled, with 75Ω Cable

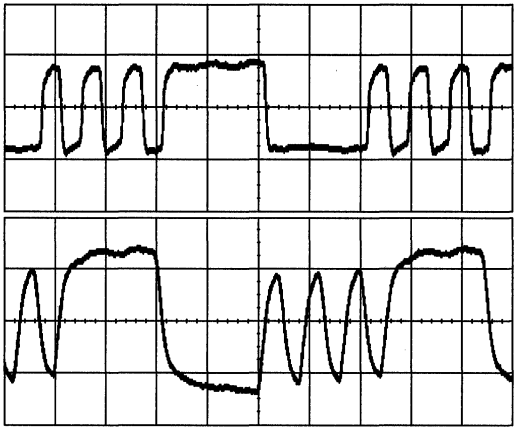


Figure 40. Thévenin Bias, Direct-Coupled, with 93Ω Cable

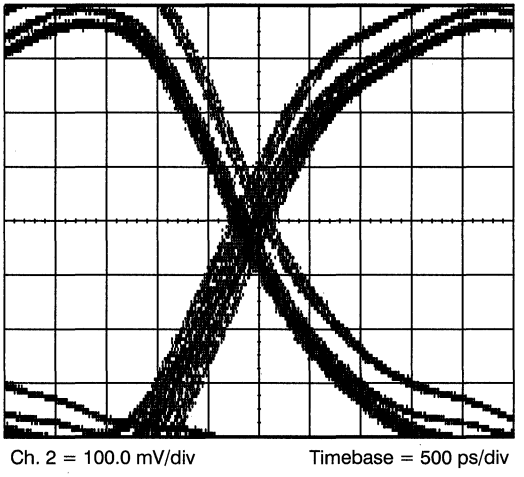


Figure 41. Eye Diagram, Thévenin Bias, Direct-Coupled, with 93Ω Cable

Table 4. Cable Impedance Comparison

Configuration	50Ω		75Ω		93Ω	
	Amplitude	Jitter	Amplitude	Jitter	Amplitude	Jitter
Thévenin Bias, Direct-Coupled	350 mV	25%	600 mV	17%	700 mV	12%

The second relationship is that as the impedance increases, the amount of jitter in the system is reduced. The ANSI Fibre Channel standard allows for links with up to 80% jitter at the receiver (Reference 8). While this standard only currently supports 75Ω coaxial cables (and 150Ω STP cables), these measurements show that 30.4-meter segments of 50Ω and 93Ω cable would also satisfy the maximum jitter specification.

Thévenin Bias, AC (Capacitive) Coupled

The equivalent circuit for a Thévenin bias differential driver, capacitively coupled to a transformer, is shown in *Figure 42*. Just as with the direct coupled system, the output bias voltage is set by the pull-up/pull-down resistor ratio. The capacitors now insure that there is no DC path through the transformer that might cause a core saturation that could limit both the bandwidth and energy transfer through the transformer.

The 1000-pF capacitors used here are RF-grade NP0-type parts. The passband of these parts (in this system) is such that they act like a high-pass filter. This limited low-end bandwidth can be seen on the long-0 and long-1 pulses in *Figure 43*. This figure shows the signal characteristics of all three impedances with a resistive load shown on the left, and 30.4-m cable and load on the right.

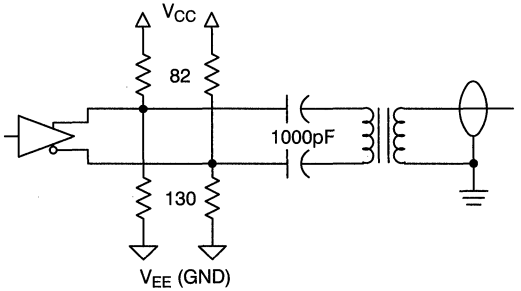


Figure 42. Thévenin Bias, AC-Coupled

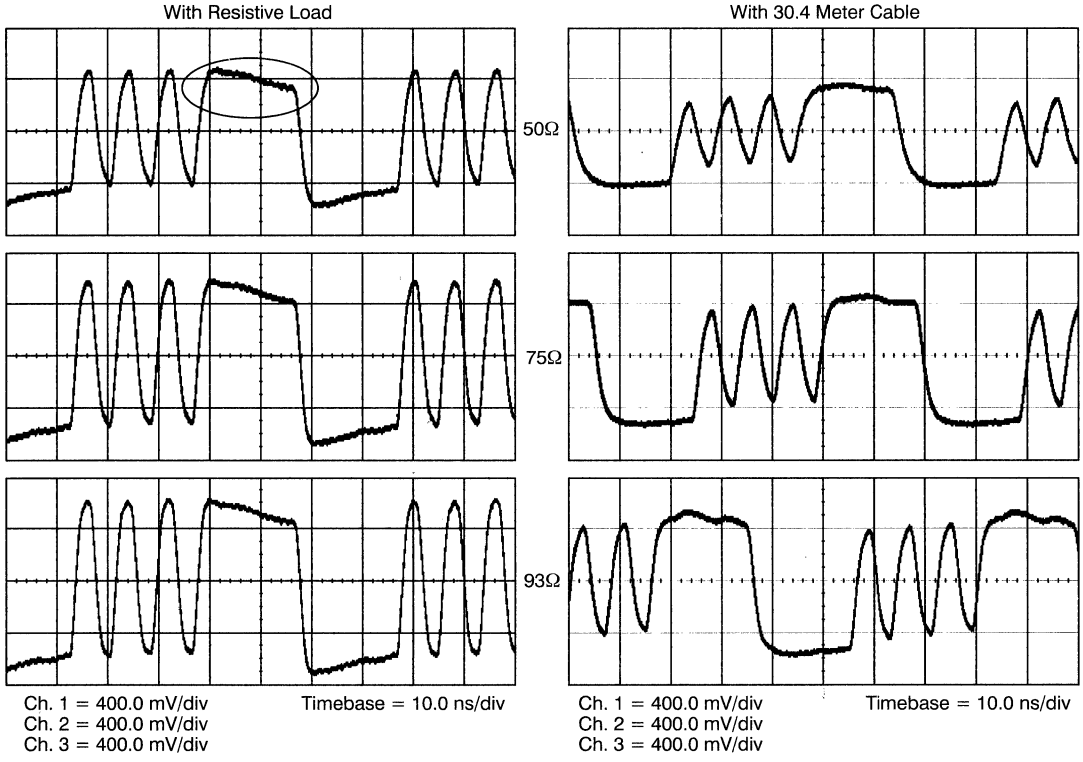


Figure 43. Thévenin Bias, AC-Coupled, with Resistive Load and Cable

On a direct-coupled connection (at the transformer secondary), these long-1 and long-0 pulses switch to their HIGH or LOW state and remain there. In this AC-coupled configuration, these same pulses switch to the same HIGH and LOW levels, but slowly lose amplitude over the duration of the pulse. This amplitude loss is called *droop*.

This droop in many cases can improve the signal characteristics at the load (receiver) end of the cable. Comparing the top right column trace in *Figure 43* with the bottom trace in *Figure 36* shows that the AC-coupled signal has a smaller peak amplitude for the long-duration pulses. This translates directly into a larger usable amplitude and smaller jitter percentage.

The capacitors in this link perform a rudimentary frequency-spectrum equalization. Because this equalization is performed prior to the signal being placed on the transmission line, it is called pre-compensation. A similar spectrum correction, when applied at the receiver end of the transmission line, is referred to as post-compensation or equalization.

These same signals are shown as eye patterns in *Figure 44*. *Table 5* compares the amplitude and jitter in these AC-coupled waveforms with the previous direct coupled configuration. The key observation made here is that the AC-coupling in all cases improves the amplitude and jitter. This improvement in all cases (with the specific coupling transformer and biasing evaluated here) is due to the limited bandwidth of the capacitor, not because there is no DC-path through the transformer. This was confirmed by actually forcing controlled amounts of DC through the transformer to determine where core saturation occurs.

Table 5. Driver Coupling Comparison

Configuration	50Ω		75Ω		93Ω	
	Amplitude	**Jitter**	**Amplitude**	**Jitter**	**Amplitude**	**Jitter**
Thévenin Bias, Direct-Coupled	350 mV	25%	600 mV	17%	700 mV	12%
Thévenin Bias, AC-Coupled	400 mV	20%	650 mV	15%	750 mV	11%

Transformer Core Saturation Testing

To validate that a small DC current flow (caused by a possible small mismatch in the ECL driver/load circuits) does not effect the signal coupled through the transformer, a small modification was made to the previous AC-coupled test set-up (see *Figure 45*). This change involved the addition of two resistors (labeled R in *Figure 45*), attached to the primary of the transformer, to force a DC current through the primary. All tests were performed with a 50Ω resistive load on the transformer secondary.

To better see the effect, the data pattern was changed to use maximum run-lengths of six bits. While this is beyond the limits of the 8B10B code, it serves to put the interface under greater stress.

The results of these tests are shown in *Figure 46*. The top trace shows the transformer secondary output with 13 mA of DC in the primary. The middle trace shows the same circuit with 30 mA of DC in the primary. The bottom trace shows 50 mA of DC in the primary. Notice that the secondary waveform starts to change around 30 mA, and is quite distorted at 50 mA. This means that the transformer core starts to saturate with around 30 mA of DC in the primary.

A normally biased and loaded ECL output can never have this much of a DC imbalance. This means that unless some type of pre-compensation is desired, there should be no need to AC-couple to the transformer primary.

Shunt Bias, Direct-Coupled

A shunt bias, where a single resistor is attached from each PECL output to V_{EE} (ground), is normally used only for digital logic applications. This is due to the slightly different rise and fall times generated as the outputs switch. When used to drive a wide-band transformer, as shown in *Figure 47*, this bias method has some distinct advantages.

First, it only requires a single resistor per driver, unlike the Thévenin bias which requires two resistors and a bypass capacitor. Second, and probably more important, this configuration allows much more of the ECL driver's signal swing to be seen on the transformer secondary.

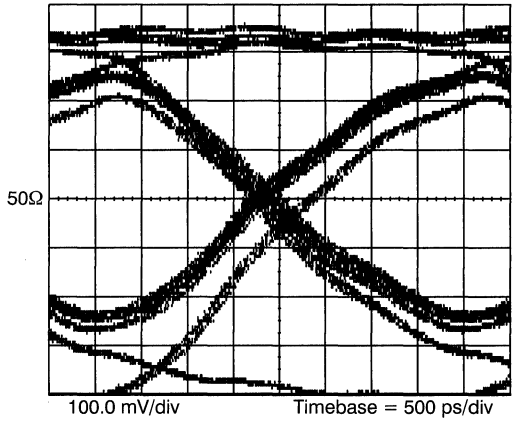
The signal transmission characteristics of this type of coupling are shown in *Figure 48*. This details the eye patterns for all three cable impedances. Unlike the previous eye diagrams, which could be displayed at a 100 mV/div scale, these signals are now shown at 200 mV/div.

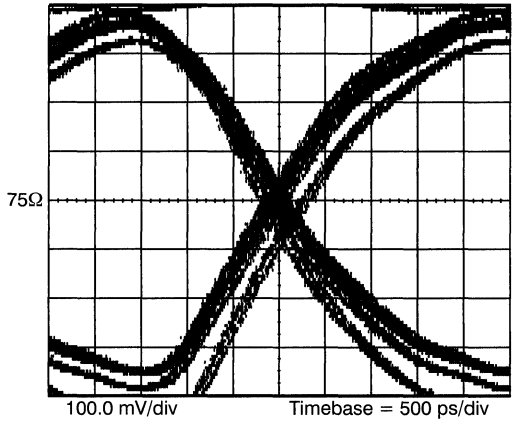
Because these signals are all direct coupled, the jitter measurements are back around where they were with the direct-coupled Thévenin bias setup. The received signal amplitude however has increased around 100 mV over that of a Thévenin bias. This shows that the system jitter is independent of the signal drive level.

Shunt Bias, AC-Coupled

By combining the improved amplitude of a shunt-bias coupling with the limited frequency response of a capacitively coupled system, it is possible to squeeze out a slightly better signal.

The circuit for this configuration is shown in *Figure 49*. Here the capacitors again serve to block some of the lower frequency spectral components, which are not as severely attenuated by the transmission line.





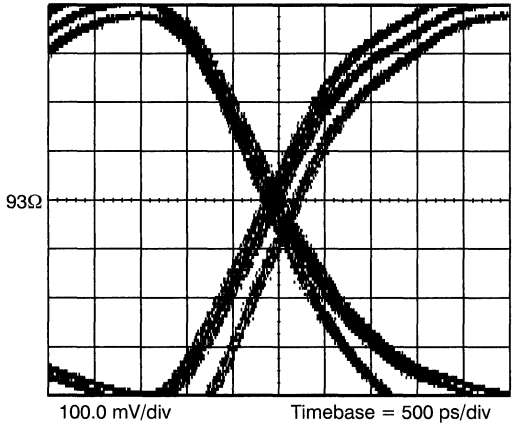


Figure 44. Eye Diagrams, Thévenin Bias, AC-Coupled, with Cable

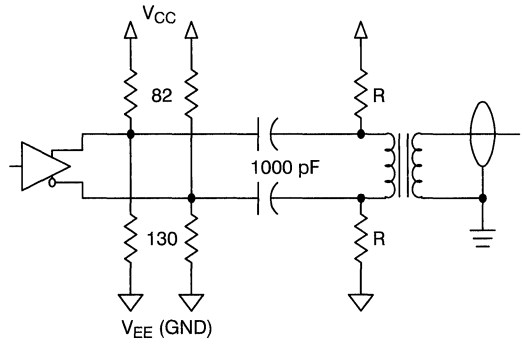


Figure 45. Transformer Core Saturation Test Fixture

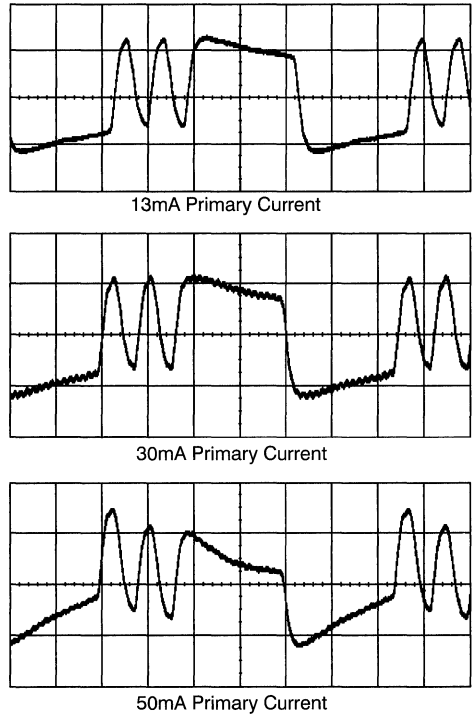


Figure 46. Transformer Core Saturation Test

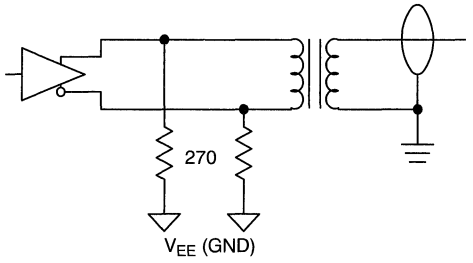


Figure 47. Shunt Bias, Direct-Coupled

The signal transmission characteristics of this type of coupling are shown in *Figure 50*. This figure details the eye patterns for all three cable impedances. These eye diagrams are again displayed at 200 mV/div.

The received signal amplitude in this shunt bias, AC-coupled configuration continues to operate as a function of cable impedance. As the cable impedance is increased, the received signal amplitude grows larger, and with less jitter.

The effect of the coupling capacitor on the circuit is more prominent on the lower impedance cables. On the 93Ω cable, the jitter improving effect (with this short length of cable) is basically non-existent. With longer cables it is expected that this will have a much larger effect.

A quantitative comparison of all four configurations is shown in *Table 6*.

Single Transformer Configurations

All of the previous coupling circuits were based on a differential driver working into a common load. These configurations allow the amplitude swing of both drivers to be presented to the load. While this is expected to be the primary coupling mode for copper interconnect, it is also possible to drive these same connections through a single driver.

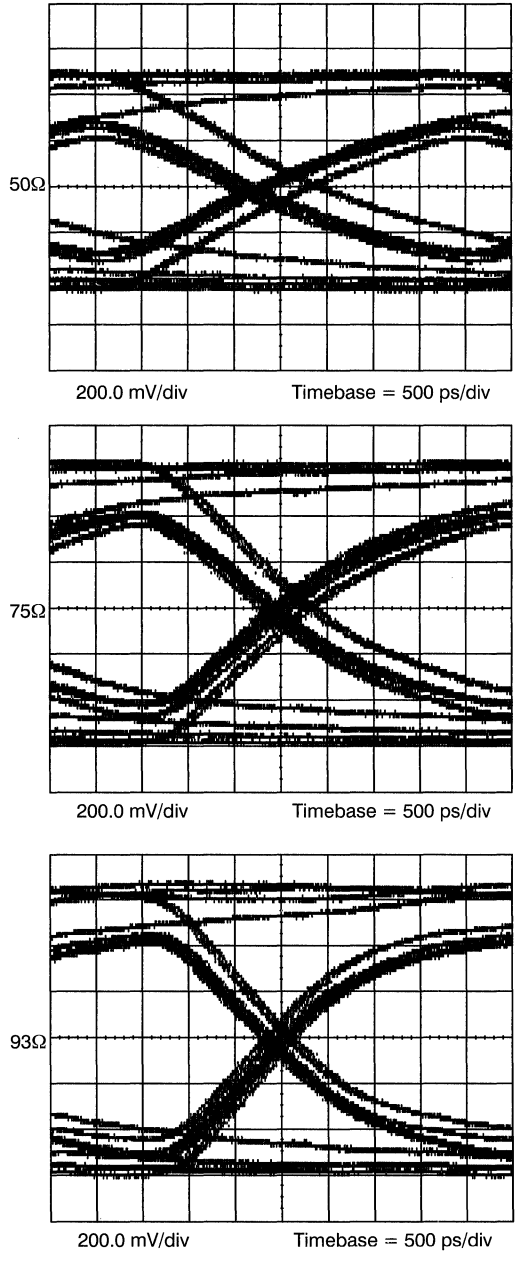


Figure 48. Eye Diagrams, Shunt-Bias, Direct-Coupled, with Cable

Table 6. Shunt vs. Thévenin Bias Comparison

Configuration	50Ω		75Ω		93Ω	
	Amplitude	Jitter	Amplitude	Jitter	Amplitude	Jitter
Thévenin Bias, Direct-Coupled	350 mV	25%	600 mV	17%	700 mV	12%
Thévenin Bias, AC-Coupled	400 mV	20%	650 mV	15%	750 mV	11%
Shunt Bias, Direct-Coupled	400 mV	25%	700 mV	16%	800 mV	11%
Shunt Bias, AC-Coupled	500 mV	17%	800 mV	15%	850 mV	12%

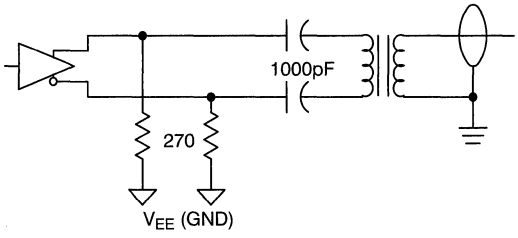


Figure 49. Shunt Bias, AC-Coupled

Single Driver, Direct-Coupled

A Thévenin-biased direct-coupled configuration is shown in *Figure 51*. When coupled in this mode it is possible to double the number of connections driven from a single source, at the expense of approximately 6 dB of amplitude on the cable.

This loss of amplitude will have minimal affect on how far a signal can be driven on a copper cable. Copper-based links for the most part are limited by jitter accumulation rather than attenuation. The amplitude loss may effect the bit-error-rate for the link due to the reduced noise margins.

Figure 52 shows the signal characteristics for this configuration when driving a 50Ω resistive load. Here the top trace shows the output of the driver while the bottom trace is at the transformer secondary. While the traces may look similar, the bottom trace is shown at a different vertical resolution. In effect only half of the driven signal is appearing at the load. This is again due to the voltage divider that exists between the transformer and the Thévenin bias network

Comparing the top trace in this figure with the same trace in *Figure 36* shows that the low side distortion is now gone. The pulses also are much more squared-off in this single driver configuration.

Single Driver, Direct-Coupled, AC Bypass

With the Thévenin bias network, both AC and DC signal components are dissipated in the network. By capacitively shunting the Thévenin network, it is possible to drop the DC signal component across the bias network, and drop the AC component across the transformer's primary. This configuration is shown in *Figure 53*.

The added capacitor will effectively double the signal delivered to the load. Because of the size of capacitor selected here, there will be some limiting of the low-frequency signal components. These affects are shown in *Figure 54*.

The capacitor again provides a small amount of precompensation to the circuit. This configuration tends to increase the source end jitter, while decreasing the jitter at the end of the cable.

Replacing the 1000-pF capacitor with a 0.027-µF part significantly changes the AC passband characteristics of the coupling network, as shown in *Figure 55*. Now the low-frequency signal components that were blocked by the small 1000-pF capacitor are allowed to couple through the transformer. This configuration will provide minimal jitter at the transformer secondary, but will have more at the end of the cable than the high-frequency bypass configuration.

Dual Transformers

In the previous differential coupling configurations where a single transformer was driven at both ends,

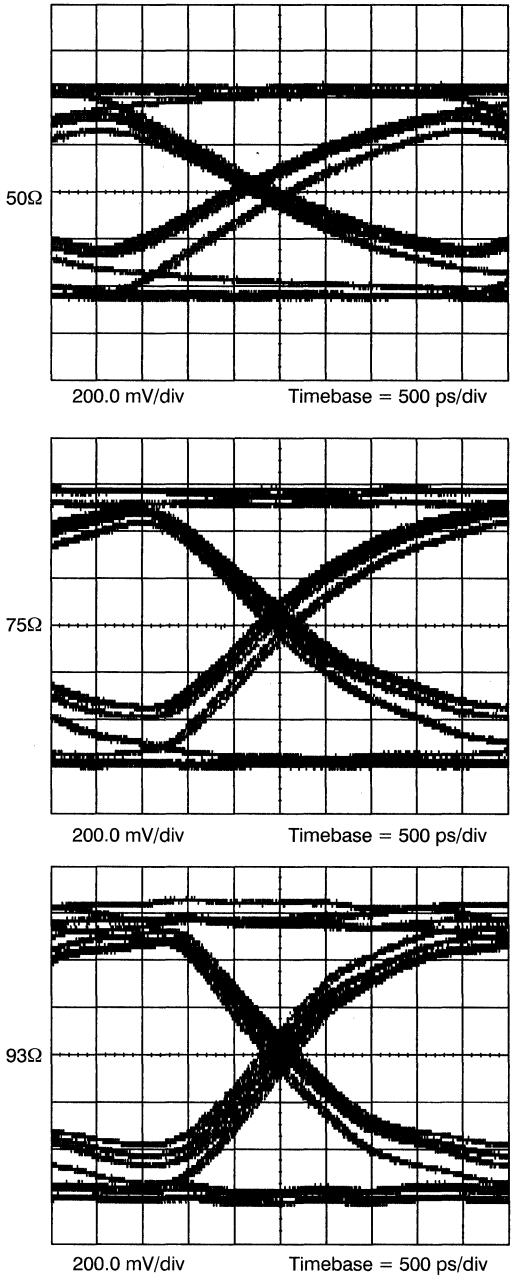


Figure 50. Eye Diagrams, Shunt Bias, AC-Coupled, with Cable

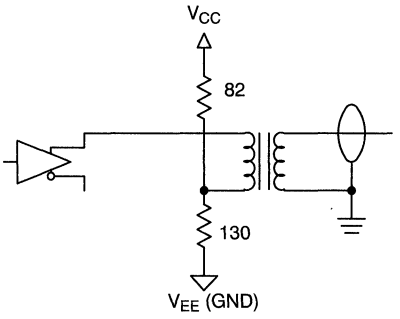


Figure 51. Single Driver, Thévenin Bias

the possibility existed of one driver having an effect on the other. To see if any such affect was present, tests were performed that used separate transformer primaries to drive a common load.

Based on the excellent waveform results achieved from a single driver/transformer configuration, the configuration in *Figure 53* (with the larger 0.27-µF capacitor) was duplicated on the complement output of the differential driver. With each of these circuits operated into separate 50Ω resistive loads, the waveforms remain the same as those shown in *Figure 55*.

When connecting the secondaries of these two transformers in series (as shown in *Figure 56*), re-

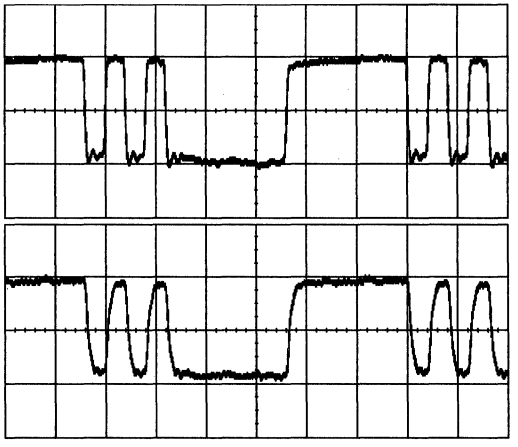


Figure 52. Single Driver, Direct-Coupled, 50Ω Resistive Load

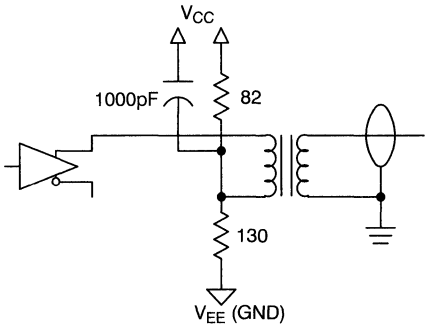
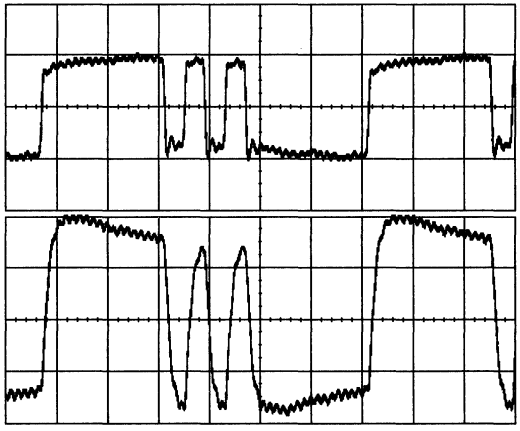


Figure 53. Single Driver, Direct-Coupled, High-Frequency Bypass

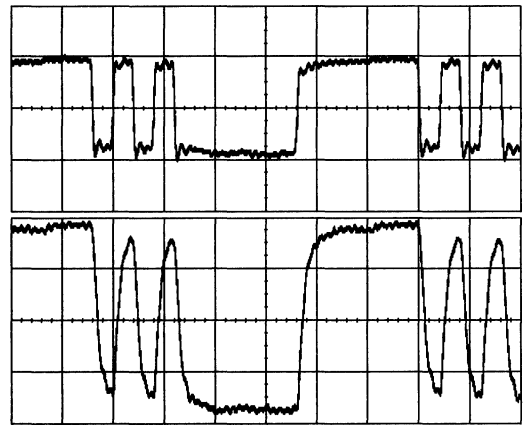
member that the polarity of the signals of the transformer attached to the complimentary driver are 180° out of phase with those of the true driver. This allows their signal amplitudes to add. *Figure 57* shows the net result of this circuit. Note the LOW-level distortion present.

The major changes that have occurred in the circuit are the amount of inductance present in the transformer(s) and the current run through them. With a single driver switching 800 mV into a 50Ω load, 16 mA of current are present. Doubling the output



Ch. 1 = 400.0 mV/div
Ch. 2 = 200.0 mV/div
Timebase = 10.0 ns/div

Figure 54. Single Driver, Direct-Coupled, High-Frequency Bypass, 50Ω Resistive Load



Ch. 1 = 400.0 mV/div
Ch. 2 = 200.0 mV/div
Timebase = 10.0 ns/div

Figure 55. Single Driver, Direct-Coupled, Low-Frequency Bypass, 50Ω Resistive Load

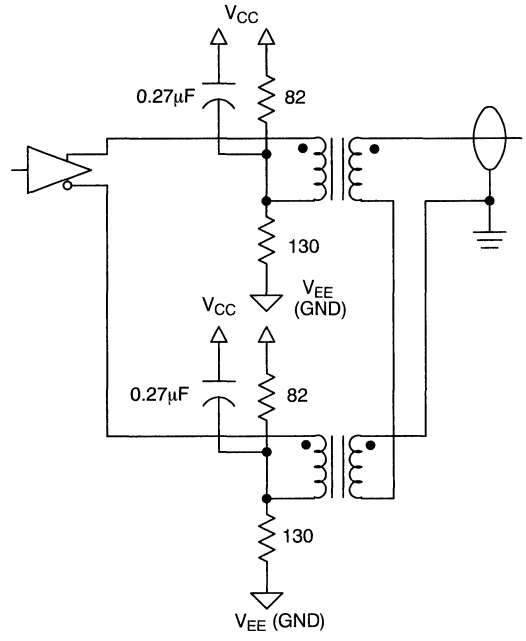
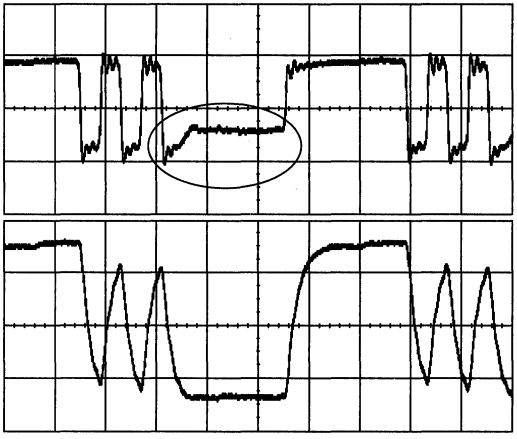
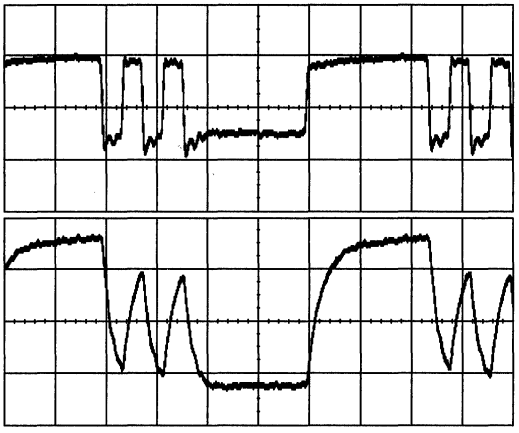


Figure 56. Dual Transformer, Series Secondaries



Ch. 1 = 400.0 mV/div
Ch. 2 = 400.0 mV/div
Timebase = 10.0 ns/div

Figure 57. Dual Transformer, Series Secondaries, 50Ω Resistive Load



Ch. 1 = 400.0 mV/div
Ch. 2 = 200.0 mV/div
Timebase = 10.0 ns/div

Figure 59. Single Driver, Direct Coupled, 25Ω Resistive Load

swing into the same 50Ω load by using both drivers, also doubles the current.

In these dual-driver configurations each driver must source twice as much current as a single driver configuration. The reason for this can be seen in *Figure 58*. With a 50Ω load on the secondary of a transformer, this same load is reflected on the primary. With dual transformers, half of the load is present on each transformer.

To confirm this, the single driver circuit in *Figure 53* (with the larger 0.27-µF capacitor) was tested with a 25Ω resistive load. The results of that test are shown in *Figure 59*. This shows that the single driver configuration also generates the zero-level offset when presented with a low-impedance load, but at half the amplitude of a dual-transformer configuration.

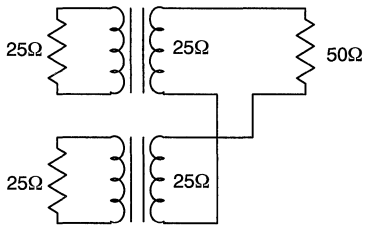


Figure 58. Dual-Transformer Equivalent Loading

This low side distortion is caused by the biasing network being sized for too large of a load impedance. An ECL driver sources current to set the HIGH or 1-level, while the bias network must sink sufficient current to set the LOW or 0-level.

The need to drive low-impedance loads places specific requirements on the current capability of the drivers. To differentially drive a 50Ω load (or transmission line) each driver must be capable of driving 25Ω single-ended loads. The line-bias networks must also be capable of sinking these large currents. This drive capability is beyond that of most ECL components, which are usually designed for only 50Ω loads. Only a few parts specifically identified as line drivers are made for operation with 25Ω loads.

The HOTlink transmitter PECL drivers are high-current line drivers and are designed specifically for driving 25Ω transmission lines. The use of standard ECL outputs designed for only 50Ω loads requires the addition of series current-limiting resistors in each primary leg of the transformer.

Long Cable Observations

When interfacing HOTLink to long cables

- Higher cable impedances exhibit lower losses and less DDJ induced jitter.
- DC-block capacitors are not necessary but may be used to provide some pre-compensation to lower the destination jitter.
- Lower transformer inductance values provide less distortion and better high-frequency bandwidths.

Conclusions

The HOTLink family of data communications parts are designed to work optimally with either fiber-optic or copper-based interconnect. When interfaced to copper media, they may be interfaced to short, medium, and long-distance connections using only low cost passive components.

References

1. Orr, William I., *Radio Handbook*, 23rd Edition, SAMS, 1992
2. Blood Jr., William R., *MECL System Design Handbook*, Fourth Edition, 1988
3. Trompeter, Ed, *Electronic Systems Wiring & Cable*, technical paper, Trompeter Electronics, Inc.
4. *The Radio Amateur's Handbook,* 50th Edition, ARRL, 1973
5. Hess, David & Goldie, John, *AN–916 A Practical Guide to Cable Selection*, National Semiconductor/Berk-Tek, 1994
6. Fowler, Bill, *Transmission Line Characteristics*, AN–108, National Semiconductor
7. *1990–91 Resistor/Capacitor Data Book*, Philips Components, 1990
8. *Fibre Channel Draft Standard*, DpANS X3.230-1994, American National Standards Institute, 1994
9. *Compliance Engineering Reference Guide*, Compliance Engineering, 1994

HOTLink™ Copper Interconnect— Maximum Length vs. Frequency

Introduction

The most common question asked about any serial interface is, "How long of a link can I have?" The answer comes down to a mixture of transmission line characteristics, and the jitter generation and tolerance of the serial data transmitter and receiver.

While the jitter characteristics of both the CY7B923 and CY7B933 HOTLink™ Transmitter and Receiver are very stable across frequency, temperature, and voltage, such is not the case for copper cables. The signal distortion introduced by these cables is very non-linear with respect to distance and frequency. In addition, there are large variations in these non-linear characteristics based on the specific type of cable selected.

Cable Testing

To determine just how these cable characteristics affect data transmission, a number of tests were performed to determine the maximum data-rate versus distance characteristics of a number of common cable types. These tests were performed using multiple CY9266–T and CY9266–C HOTLink Evaluation Boards, and nine different types of copper cable.

Equipment

The following equipment was used for the cable evaluations:

- HP8116A pulse/function generator
- Three CY9266–C HOTLink Evaluation Boards for testing coaxial cable
- Four CY9266–T HOTLink Evaluation Boards for testing twisted-pair cable
- Multiple segments of each cable type, capable of being combined to length multiples of 50 feet

The specific cable types evaluated are not meant to be inclusive of all possible cable types that may be used with HOTLink. Instead they were selected to represent a relatively wide range of commonly available cable types that are often used for communications or networking. The cables evaluated are listed in *Table 1*.

The electrical configuration used for the testing, between the HOTLink Transmitter and Receiver, is shown in *Figure 1*. This figure is somewhat simplified from the actual circuit on the CY9266 boards, but serves to illustrate the transmitter and receiver bias, coupling, and termination networks.

The only changes made to the CY9266 boards (to accommodate the different cable types), was to change the termination resistors to match the cable impedance of the specific cable under test.

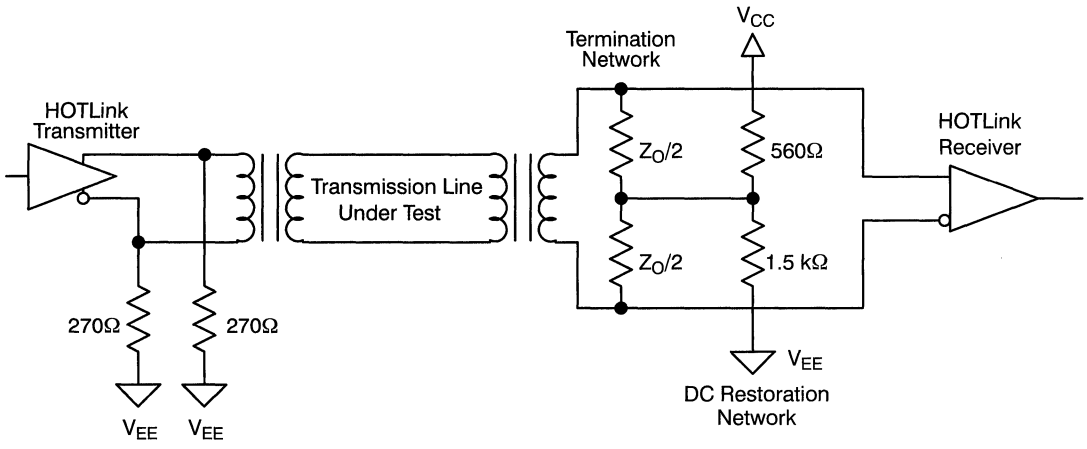


Figure 1. Cable Test Configuration

Table 1. Tested Cable Types

Cable Type	Mfgr.	Model	Type	Z_O
Coax	Belden	8219	RG58	50Ω
Coax	Belden	9259	RG59	75Ω
Coax	Belden	8255	RG62	93Ω
Coax	Belden	9066	RG6	75Ω
Coax	Belden	8218		75Ω
Coax	Belden	83264	RG179	75Ω
Shielded twisted pair	Belden	9688	IBM® Type-1	150Ω
Twisted pair	Comtran	PCC–FT4	UTP-3	100Ω
Twisted pair	Comtran	PCC–FT6	UTP-5	100Ω

Test Procedure

The testing consisted of using the built-in self-test (BIST) capability of the HOTLink Transmitter and Receiver to determine where the link was usable (error free) and where errors started to occur. An external frequency source was applied to both the transmitter and receiver and adjusted (both up and down) in frequency while monitoring the BIST error display for any link errors.

The criteria selected for an error-free link was that no errors be detected for a period of 20 minutes at a specific operating frequency and distance. This allows a large number of bits to be sent and received, and allows the HOTLink Transmitter and Receiver to stabilize at an operating temperature.

It is understood that this period of time does not guarantee an error-free link forever. Any link, no matter how good, will still have some error rate characteristic associated with it. However, observations of these copper based links (made in the process of these tests) has shown that if a link runs error free for this 20-minute period of time, it will remain so for a much longer period (i.e., multiple days).

Test Results

RG58—50Ω Coaxial Cable

The first system tested used the 50Ω RG58 coaxial cable. This cable is commonly used in the Ethernet physical variant known as 10BASE2 or ThinNet. The test results for this cable are plotted in *Figure 2*.

Of the three cards used in the testing, one used coupling transformers that had approximately twice the inductance of the other two. For this specific cable type (and for all other coaxial cables tested with this card) the maximum error free lengths at a specific operating frequency were always shorter than the

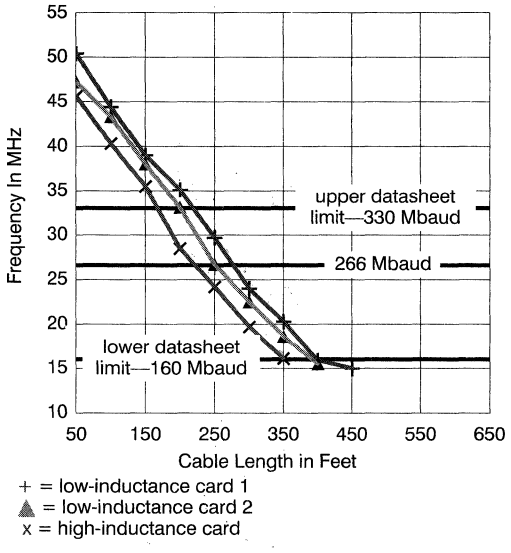


Figure 2. RG58 Test Results, Linear Frequency Scale

equivalent lengths on the cards with low inductance transformers.

Two reasons exist for this difference in operational length. First is based on the high-frequency bandwidth of the transformers. The high-inductance transformer (per the manufacturer's data) has a high end −3 dB bandwidth of around 250 MHz. Around this frequency point in the transformer, significant attenuation and phase shifts occur in the transmitted signal. Since it is these upper frequencies that provide a reasonable shape to the signal, their attenuation and distortion in the transformer causes less of these signal components to be available at the receiver.

The second effect is caused by the low-frequency bandwidth of the transformer. The higher the transformer inductance, the better its low-frequency response. Unfortunately it is the low-frequency content of the transmitted signal that induces most of the data-dependent jitter (DDJ) in the serial link.

In *Figure 2*, the frequency scale shows the clock rate delivered to the HOTLink Transmitter and Receiver. This clock rate is the byte rate for the transmitter and receiver. Because of the 8B10B encoding used to send serial information, the actual bit-rate on the serial interfaces is ten times this rate (i.e., 25 MHz=250 Mbits/second).

This figure shows that with an RG58-type cable (having the same attenuation characteristics of the cable tested here), that it is possible to reliably transmit information at all distances ≤200 feet when operating at the maximum HOTLink datasheet limit of 330 Mbits/second (when using low-inductance transformers). As the data-rate is reduced, the maximum operable length increases, such that at the minimum datasheet limit of 160 Mbits/second, the link may be operated at all lengths ≤400 feet.

Note: These distances are all based on uncompensated (non-equalized) links. By adding frequency-selective filter components to either the source or destination ends of the cable it is possible to greatly extend the error-free link lengths. All test data presented in this application note is only for uncompensated links.

RG59—75Ω Coaxial Cable

RG59 is a 75Ω coaxial cable manufactured in a similar size and construction to RG58. The main difference between them is the ratio of inner to outer diameters that determine the characteristic impedance of the cable. When tested to the same criteria as the RG58 cable (as shown in *Figure 3*), numerous differences in operation become apparent.

The most obvious difference is that the operable lengths have increased significantly: as much as 50% at 330 Mbits/second and 37% at 160 Mbits/second. In addition there is now a flat portion at the top end of the operating frequency range where changing the cable length has no effect on the maximum data-rate.

At this top-end frequency the interconnect system still modifies or distorts the transmitted signal. However, the amount of distortion is small enough that a different factor is limiting the maximum operable distance of the link. At this frequency, the phase-locked loops (PLLs) in the transmitter and receiver are up against their maximum operable limit. Because the received signal characteristics remain within the minimum acceptable limits of the

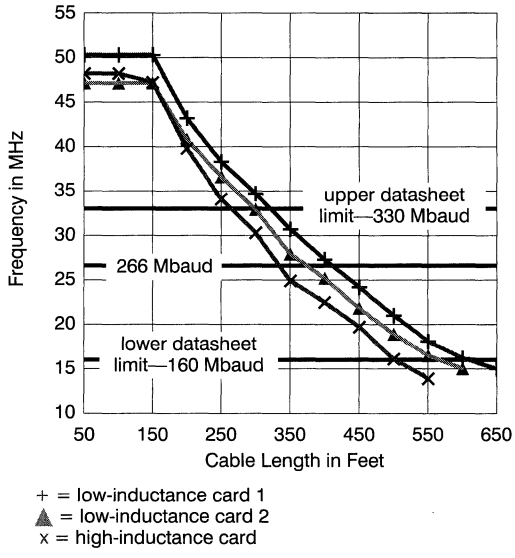


Figure 3. RG59 Test Results, Linear Frequency Scale

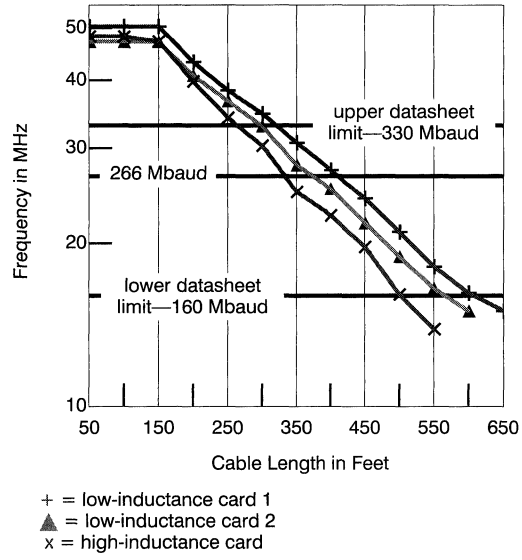


Figure 4. RG59 Test Results, Log Frequency Scale

receiver through the 150-foot distance, the line remains flat.

Beyond the 150-foot length the received signal is distorted enough such that the operating frequency must be reduced in order to bring the signal back to where the receiver can accurately capture it.

By taking the same data in *Figure 3* and plotting it on a logarithmic frequency scale in *Figure 4*, another characteristic becomes visible. Now the curves for data-rate versus distance appear as a straight line. This means that this is actually an exponential function.

Other Cable Types

Data-rate versus distance information was taken for all the cable types listed in *Table 1*. By plotting a composite chart of all these cable types, it is possible to see how the different cable characteristics affect the maximum operable length. This information is shown in *Figure 5*.

RG62—93Ω Coaxial Cable

RG62 is a 93Ω version of RG59 cable. It is made by removing some of the dielectric in the RG59 cable and replacing it with air, lowering the dielectric constant. Since the cable impedance is based on the dielectric constant of the spacer (in addition to the dimensions of the conductors), lowering the dielectric constant raises the impedance to 93Ω

Comparing the operable length characteristics of this cable with that of the RG58 and RG59 cables shows that the higher impedance RG62 again improves the maximum usable distance at all frequencies.

RG6—75Ω Coaxial Cable

RG6 is a 75Ω coaxial cable commonly used for CATV applications. While this cable does have the same impedance as the RG59 cable, its construction is quite different, as are its data transmission characteristics. This cable has larger inner and outer diameters for the conductors used in the cable. While the ratios of these diameters do maintain a 75Ω system, the increased dimensions create larger surface areas for the conductors and therefore lower losses.

When compared to the RG59 cable, RG6 allows operable distances of nearly twice as far. At the low end (160 Mbits/second) of the HOTLink operating range, this approaches 1000 feet.

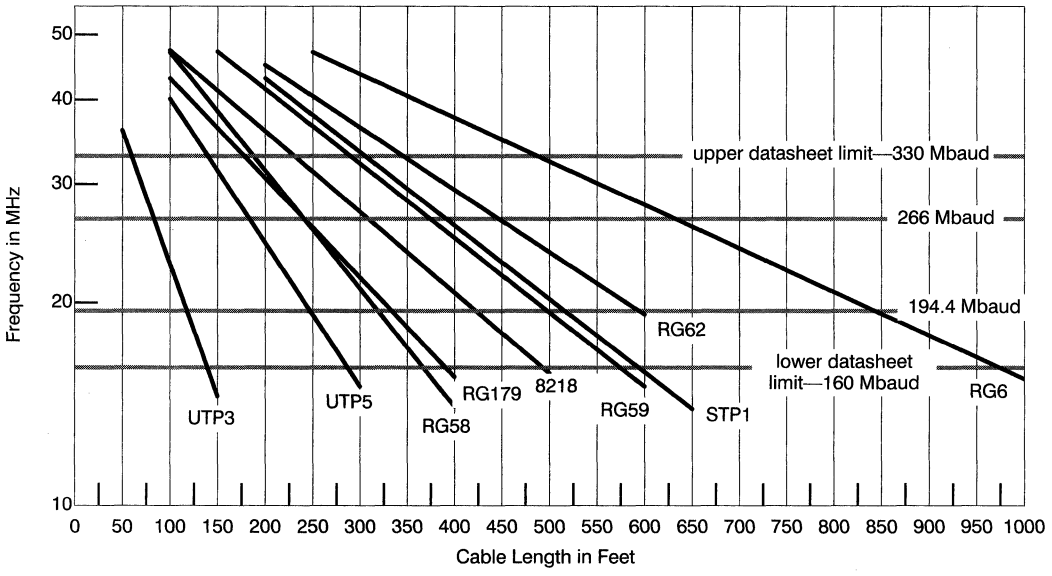


Figure 5. Maximum Data-Rate versus Distance Comparison

RG179 and Belden 8218—75Ω Coaxial Cable

The RG179 and Belden 8218 cables are also 75Ω types. These cables, however, are designed for different environments where signal loss is not the primary concern. The 8218 cable type is a miniature form of RG59. With the smaller diameters (and smaller surface area) its losses at all frequencies are greater than those of RG59.

RG179 is a cable designed both for tight spaces and harsh environments. Its Teflon® jacket allows it to be used where most cables cannot. If it was manufactured using the same materials as RG59 or 8218 cable, its losses would be much higher than they currently are. To limit the losses, the inner copper conductor is plated with silver to improve the skin-depth for high-frequency signals.

Twisted-Pair Cables

The other family of cables supported by HOTLink are known as twisted-pair cables. These cables were tested with the CY9266−T HOTLink boards.

IBM Type-1—150Ω Shielded Twisted-Pair Cable

The IBM Type-1 cable (STP1) consists of two individually shielded twisted pairs in a single cable. The cable itself was designed for token-ring network applications operating at 4 or 16 Mbits/second. These network speeds are much less than those supported by HOTLink. Due to the excellent signal generation and handling characteristics of the HOTLink components, this same cable is usable over even greater distances at more than ten times its designed data rate.

This cable has similar distance characteristics over frequency to the RG59 coaxial cable. Because two signal pairs are present in the same cable, a bidirectional link can be built using a single cable.

Note: Other coupling mechanisms exist that permit bidirectional signal transmission on a single set of conductors. The theory and implementation of these specialized structures is beyond the scope of this document.

Mechanically different forms of this cable exist with slightly modified signal characteristics. These variants (Type-2, Type-6, etc.) add extra non-data conductors or uses stranded-conductor construction to

improve flexibility. If the variant selected has similar attenuation characteristics to Type-1, it should operate with a similar data-rate versus distance curve.

UTP3 and UTP5—100Ω Unshielded Twisted Pair

UTP3 and UTP5 are unshielded twisted-pair cables, most commonly used for 10BASE-T Ethernet or telephone installations. UTP3 (also known as category 3) is rated for Ethernet use at 10 Mbits/second at distances up to 100 meters (329 feet), while UTP5 is rated at 100 Mbits/second at the same distance. In these unshielded cables (unlike STP1 or the coaxial cables), crosstalk becomes a significant link-limiting factor.

Crosstalk occurs because of the close proximity of the two signal pairs. With no shield to keep their respective signals separated, the cable itself becomes both a long coupling transformer and coupling capacitor. This crosstalk combines with the attenuation characteristic of the cable to distort the signals on the cable.

These unshielded cables will work fine for short- to medium-length interconnections when used with HOTLink. However, the lack of a cable shield may limit their use to environments where radiated emissions are not a concern; i.e., inside a shielded cabinet or other enclosure.

General Observations

- Lower inductance transformers allow greater operating distance due to wider bandwidth.
- Higher impedance cables have lower losses and allow greater operating distance.
- Larger diameter cables have less attenuation and allow greater operating distance.

Eye Pattern Testing

While measurement of errors in a link does yield a significant amount of information about link operation, it does not explain the actual failure mechanism; i.e., why a signal is received in error. To do this requires looking at the actual signal. The following eye patterns and oscilloscope diagrams are used to explain the signal failure mode. All measurements are made with error free links based on RG59 cable.

Figure 6 shows the wide-open eye at the source end of a link for both a normally driven and a source terminated (series resistance added to the driver,

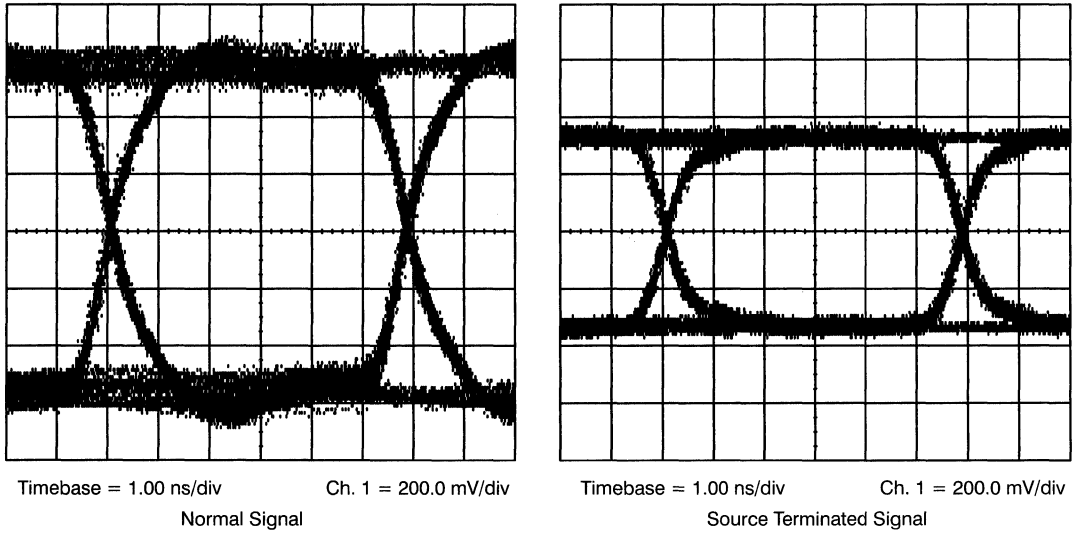


Figure 6. Error-Free, 173-Mbit/second Signal at the Driver End of a 550-Foot RG59 Cable

equal to the cable impedance) system. The eye has minimal distortion in both systems, but the added source resistance reduces the source signal amplitude by 6 dB for the source terminated link. These links both operate error free at 173 Mbits/second with 550 feet of cable attached.

The same two systems are shown in *Figure 7* at the receiver end of 550 feet of cable. Things look a bit different here. Now the eye is almost completely closed. The width of the opening in both configurations is approximately 500 ps. The only significant difference between the two links is that the source terminated signal has a smaller noise margin.

To view the effect on high data-rate signals, two new links were configured at 363 Mbits/second with 300 feet of cable. At this data rate the bit-cell time is approximately half that of the pervious configuration. The source-signal eye diagrams for these systems are shown in *Figure 8*. Again, at the source end of the cable the signals are clean. While the edges appear to have somewhat slower ramp rates, this is due to the change in sweep frequency for the oscilloscope from 1 ns/division to 500 ps/division.

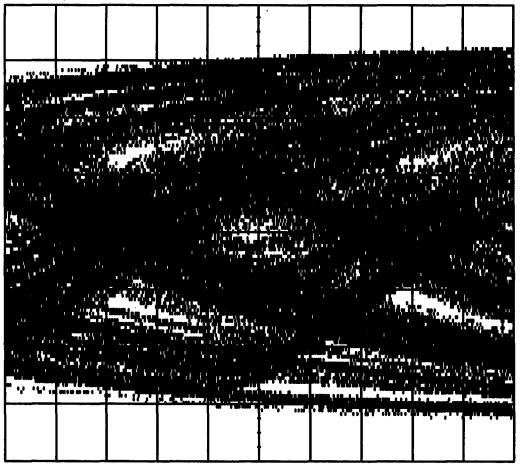
Figure 9 shows the signals at the receiving end of the 300-foot cable. These signals look similar to the 550-foot link. The overall amplitude is somewhat larger, due to the lower attenuation of the shorter cable, but the eye is still almost completely closed. At this faster data rate, the minimum eye opening is again approximately 500 ps.

The fact that the minimum eye opening of approximately 500 ps remains the same at both data-rates is not just a coincidence. This number is based on the jitter tolerance and static alignment characteristics of the HOTLink receiver PLL and data-capture circuits.

Linear Time View

The minimum-eye handling capability is a fixed characteristic of the HOTLink receiver. Changing the source-signal amplitude or data rate has no significant effect on this characteristic. But this still does not explain why the eye closes in the first place. To see this, it is necessary to look at how individual bits interact with each other.

To see bit interaction on an oscilloscope it is necessary to change from a random data pattern (like the BIST pattern that was used for the previous tests), to a fixed pattern. To show the worst-case bit interaction it is also necessary to use a data pattern that contains the maximum and minimum run-lengths of



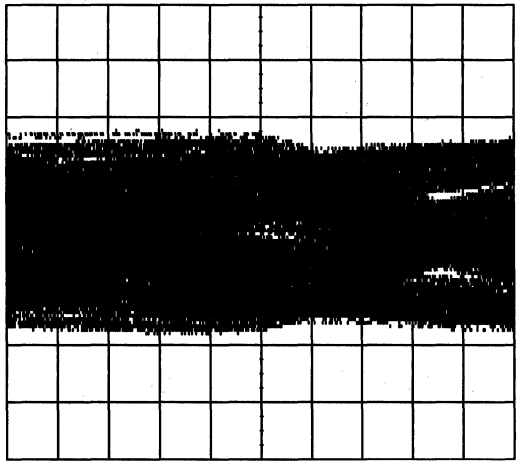
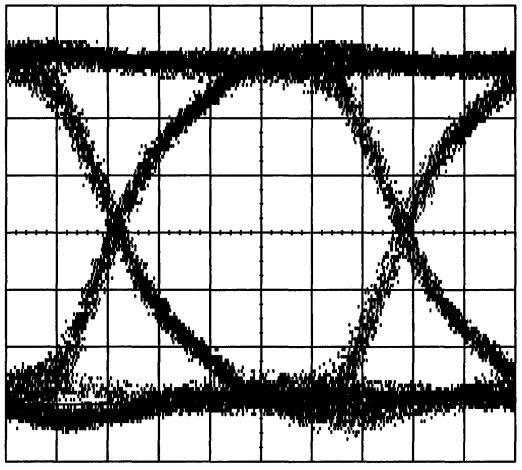
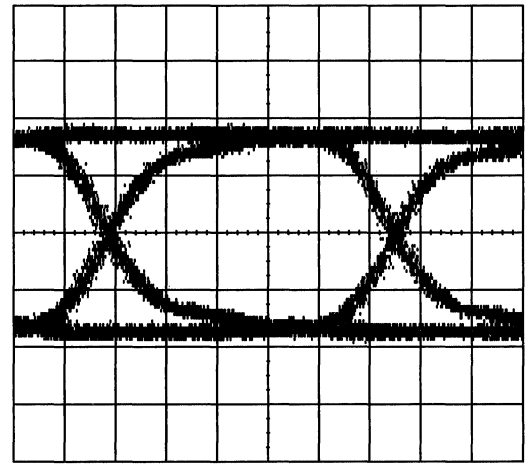


Figure 7. Error-Free, 173-Mbit/second Signal at the Receiver End of a 550-Foot RG59 Cable



Timebase = 500 ps/div Ch. 1 = 200.0 mV/div
Normal Signal



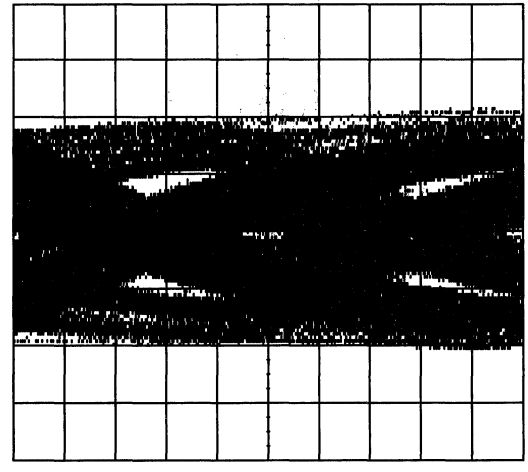
Timebase = 500 ps/div Ch. 1 = 200.0 mV/div
Source Terminated Signal

Figure 8. Error-Free, 363-Mbit/second Signal at the Driver End of a 300-Foot RG59 Cable

1s and 0s. Fortunately, a pattern meeting these characteristics is automatically generated by the HOTLink Transmitter when both $\overline{ENA}$ and $\overline{ENN}$ are disabled. The character sent under these conditions is known as a K28.5 code, which (following the 8B10B disparity rules) generates a repeating 20-bit pattern of 00111110101100000101.

This pattern, when viewed at the end of the cable under the same data-rate and cable lengths of the

Timebase = 500 ps/div Ch. 1 = 100.0 mV/div
Normal Signal



Timebase = 500 ps/div Ch. 1 = 100.0 mV/div
Source Terminated Signal

Figure 9. Error-Free, 363-Mbit/second Signal at the Receiver End of a 300-Foot RG59 Cable

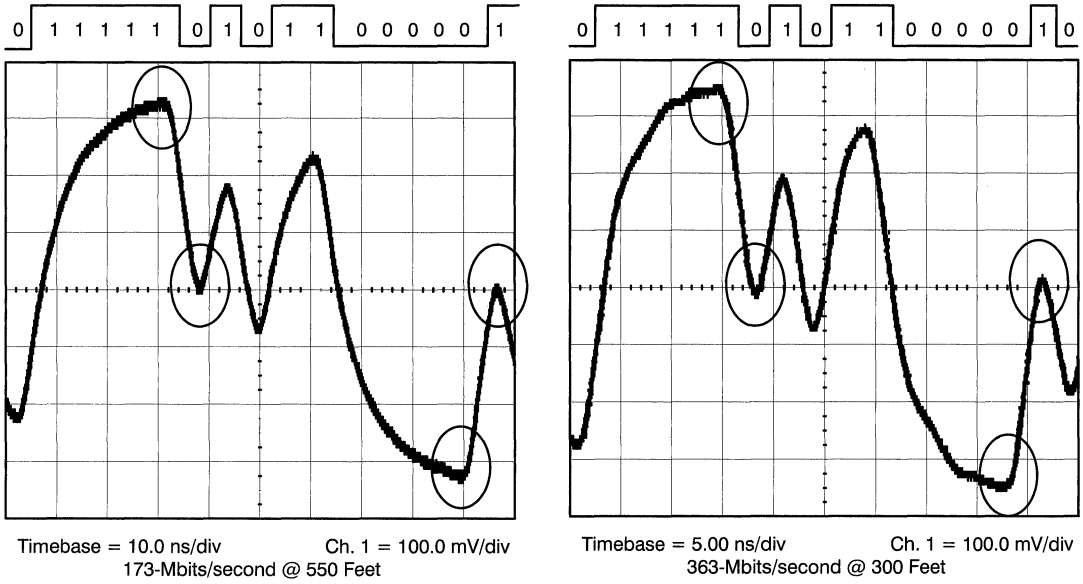


Figure 10. Error-Free, K28.5 Character at Maximum Data Rate

previous two tests, is shown in *Figure 10*. The highlighted areas in each configuration show the bits that interact to cause the eye to close. In both configurations, two of these bits (at this worst-case data-rate) barely cross the receiver threshold. The long 1s and 0s immediately preceding them cause the signal to move the farthest from the receiver threshold.

The K28.5 character will always generate a signal that looks approximately the same at the maximum length limit of an uncompensated link. This is due both to the physics of the transmission line, and to the exceptional jitter tolerance of the HOTLink Receiver. The addition of an equalizer would level out the transitions and keep them centered around the receiver threshold.

General Observations

- Signal amplitude is not the length-limiting factor for most links.
- The HOTLink Receiver's jitter sensitivity window is approximately 500 ps in size.
- Equalization will allow much longer links.
  - — HOTLink Receiver only requires 50 mV of signal.
  - — Equalization may allow link lengths of four times that of a non-equalized link.

Conclusion

The CY7B922 and CY7B933 HOTLink data communications components can be used in communications links with almost any configuration of copper media. In these links the frequency attenuation characteristics of the copper media are the primary length limiting factors for a link. The enhanced sensitivity of the HOTLink receiver allows usage of forms of signal equalization that allow operation over much greater distances than non-equalized links.

Using HOTLink™ with Long Copper Cables

Overview

The use of HOTLink™ data communications products to drive copper media is documented in a Cypress application note titled "Driving Copper Cables with HOTLink." Long transmission lines (those that cannot be treated as lossless) present additional design concerns. The special characteristics and concerns of operation with long copper cables are covered here in this application note. This application note is also expected to be used in conjunction with a companion document titled "HOTLink Design Considerations."

Primary Topics

The primary topics covered in this application note are

- Signal propagation
- Attenuation/Dispersion
- Equalization

Signal Propagation

Communication on short lengths of copper media allow the transmission line to be treated as lossless; i.e., a 1V square wave driven at one end of the cable comes out the other end with the same amplitude and waveshape. This is based on the simple relationship for transmission line impedance listed in *Equation 1.*

$$Z_o = \sqrt{\frac{L}{C}} \qquad \text{Eq. 1}$$

Real life transmission lines are not lossless. They contain numerous parasitic elements that cause a signal to distort as it propagates down the transmission line. When dealing with long cables, this equation must be modified to take into account the actual parasitics present in the transmission line. This places series-R and shunt-G components back in the calculation as shown in *Equation 2* (Reference 1).

$$Z_o = \sqrt{\frac{R + j\omega L}{G + j\omega C}} \qquad \text{Eq. 2}$$

Loss Factors

This equation gets us bit bit closer to reality, but it assumes that the L, R, C, and G elements for a transmission line remain constant over frequency. In reality these "constants" often vary with frequency and are modified by four secondary loss factors:

- Skin effect
- Proximity effect
- Radiation loss effect
- Dielectric loss effect

Skin Effect

Skin effect is a current flow phenomenon where the cross-sectional current distribution in a conductor is affected by frequency. The higher the signal frequency, the higher the concentration of current on the surface of the conductor.

Skin effect is usually modeled as a dividing line that specifies the depth from the conductor surface where all current at a specific frequency is concentrated. In reality there is always some current flow in all parts of the conductor. At the higher frequencies most of it is concentrated at the surface.

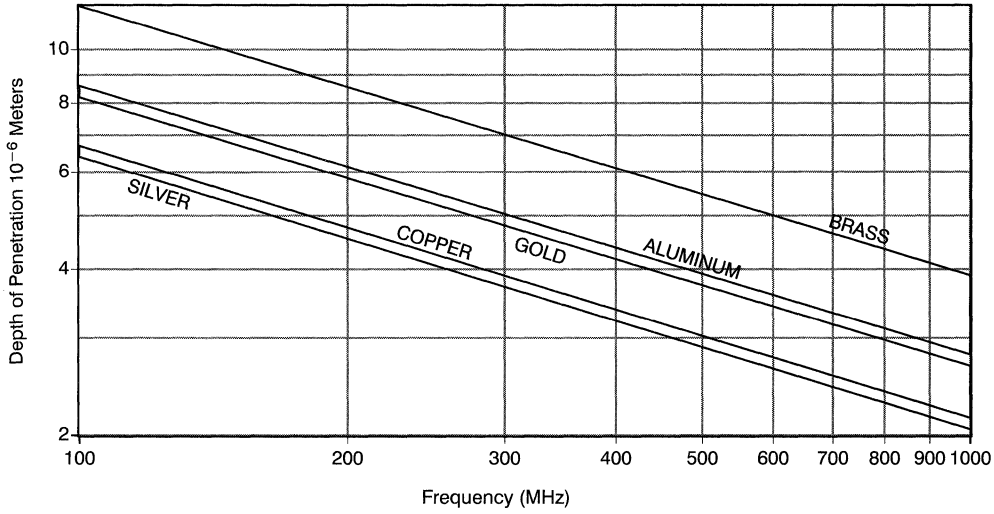


Figure 1. Effective Skin Depth

The effective skin depth is calculated using *Equation 3* (Reference 5).

$$d = \frac{1}{\sqrt{\pi f \mu \sigma}} \qquad \text{Eq. 3}$$

where:

μ = magnetic permeability of the conductor and

σ = conductivity of the conductor

Plotting effective skin depth over frequency (log/log scale) for a few common conductors (as shown in *Figure 1*) shows an interesting effect: all the lines are parallel. This is because the effective skin depth is directly proportional to the square root of frequency (Reference 2).

This change in the skin depth increases the conductors resistance as frequency is increased. This resistance change over frequency generates most of the attenuation losses in a cable (the L and C reactances are assumed to be lossless).

Figure 2 shows a frequency response plot of a few common cable types. The attenuation slope is approximately 0.5 for most of the cable types. This holds true for most standard sized cable constructions. For cables with composite plated conductors (like the RG179 cable) with various plating types (silver over copper over steel) the slope is modified by the changing current distribution in the different conductor types.

Proximity Effect

The proximity effect is caused by the current generated forces in adjacent conductors. Here the current distribution within a conductor is altered by the current present in a nearby conductor. This current redistribution works in conjunction with skin effect losses to further attenuate a signal. This loss factor does not effect coaxial cables but does effect twisted/parallel-pair cables, especially at higher frequencies. Generally the closer the conductors are and the higher the frequency, the greater the loss.

Radiation Loss Effect

Radiation loss is that signal lost due to electromagnetic radiation. This primarily effects unshielded-pair cables, or cables with poor shielding effectiveness. This loss type is often affected by those materials in close proximity to the transmission line.

For balanced transmission lines, it is also affected by the current balance within the two conductors in the transmission line. Any mismatch in amplitude or phase between the signals in the two conductors will

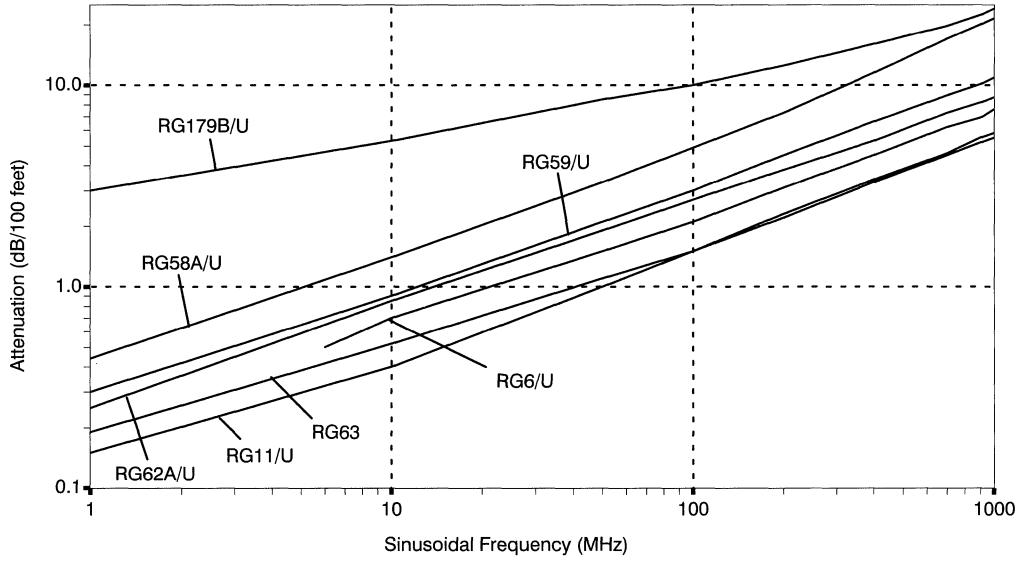


Figure 2. Coaxial Cable Attenuation Characteristics

radiate energy instead of propagating that energy down the transmission line.

Dielectric Loss Effect

Dielectric losses are those caused by the shunt conductance in the cable. This is represented by the G parameter in the impedance calculation in *Equation 2*. The loss mechanism here is current leakage through the dielectric. This loss is frequency sensitive and increases with frequency.

Reactance Factors

Just as the cable resistance and conductance vary with frequency, so do the inductance and capacitance. Both tend to decrease slightly with increasing frequency.

The change in inductance is due to the changes in skin effect, proximity effect, self inductance, and radiation loss. The change in capacitance is due to the dielectric constant of the dielectric spacer changing with frequency. The amount of capacitance change varies with the type of dielectric and the range of frequencies (Reference 1).

Signal Effects

These attenuation characteristics do more than just degrade the amplitude of a signal as it travels down a transmission line. They also affect the waveshape by distorting the rising and falling edges. The amount of the distortion is actually predictable, but it requires transformation of the source signal from the time domain to the frequency domain. This transformation is done using Fourier analysis.

Some of these effects may be illustrated using two simple square wave patterns. The first pattern is based on the highest frequency data pattern that can be sent, a continuous 0101 (D21.5 character) pattern. Using a 30-MHz byte-clock this pattern is equivalent to a 150-MHz square wave. The second pattern is based on the lowest frequency data pattern that can be sent, a continuous 0000011111 (K28.7) pattern (Reference 6). This pattern ends up being an exact match in period to the source clock (30 MHz) with a fixed 50% duty cycle.

Because the input waveforms are not true square waves, time constant curves based on a natural logarithm were used to synthesize the the rising and fal-

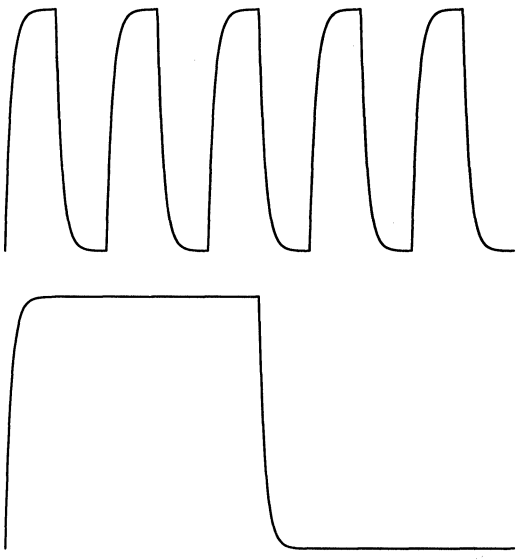


Figure 3. Synthesized D21.5 and K28.7 Waveforms

ling edges. These rising and falling edge equations are listed in *Equations 4* and 5 respectively.

$$T_R = 1 - e^{(-t/T)} \quad \text{Eq. 4}$$

$$T_F = e^{(-t/T)} \quad \text{Eq. 5}$$

In these equations, T represents the time constant for rise and fall time. For the waveforms generated for this example, a T of 400 ps was used. *Figure 3* illustrates the signals generated with these equations for both D21.5 and K28.7 characters (300-Mbit/second bit-rate).

Running a 4096 point FFT on these waveforms yields the spectral components in *Figure 4*. The vertical axis here is plotted on a log scale and shows the magnitude of the phasor at each spectral point.

Unlike a spectrum analyzer which only displays the magnitude of the spectral components, an FFT of a waveform yields both magnitude and phase in rectangular form as a complex number. To plot this information requires conversion to polar notation of magnitude and phase angle. This calculation of the magnitude portion is done using *Equation 6* (Reference 7).

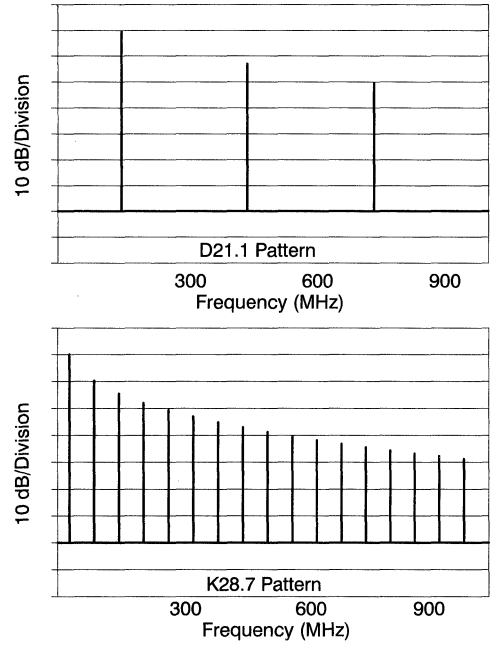


Figure 4. FFT Spectrum of Synthesized D21.5 and K28.7 Patterns

$$Magnitude = \sqrt{Re^2 + Im^2} \quad \text{Eq. 6}$$

An FFT is based on numeric analysis rather than a physical measurement and will calculate signal components with an amplitude of zero. Because Log(0) is equal to $-\infty$, a calculated FFT does not have a noise floor. To plot the results in a usable form requires the addition of an artificial noise floor to present the points of interest on a reasonable scale. To allow a better comparison with a real life environment, the noise floor in *Figure 4* is set at −80 dB.

Attenuation Effects

Now that the relative signal amplitude of each of the spectral components is known, a correction factor, based on the attenuation generated by a length of cable, can be applied to the spectral components. This attenuation is applied to the magnitude of the vector. A separate correction factor must be applied to the phase component.

Examination of a cable vendor's catalog will find a table for each cable listing attenuation at a few spe-

cific frequencies. The vendor's list of one such cable is found in *Table 1* (Reference 3). This information would be very helpful if the frequencies listed just happened to match up with the frequency components present in the signal being evaluated. Unfortunately this is rarely the case. Instead what must be done is to translate the table back into its transfer function, and use this function to calculate the attenuation at the specific frequencies of concern.

From *Figure 2* it is understood that that transfer function for a cable (in most cases) is approximated by a straight line, when plotted in log/log format. Geometry allows this line to be described in multiple ways, either by two points or as a slope and offset.

The manufacturer's attenuation data listed in *Table 1* is the same data that is plotted in *Figure 2*. Because this curve has few inflections, any of the points listed in the table may be used to approximate the transfer function. Since the data is plotted on a log/log scale, the calculations must be based on the log of both the frequency and the attenuation as shown in *Equation 7*. *Equation 8* calculates the slope for this cable type using data points at 10 MHz and 400 MHz (both at 100 meters).

Table 1. Attenuation for Belden 9659 Cable (RG59-type)

Frequency (MHz)	Nominal Attenuation	
	dB/100 Feet	dB/100 meters
1	0.3	1.0
10	0.9	3.0
50	2.1	6.9
100	3.0	9.8
200	4.5	14.8
400	6.6	21.7
700	8.9	29.2
900	10.1	33.1
1000	10.9	35.8

$$slope = \frac{Y_2 - Y_1}{X_2 - X_1} = \frac{\log(A_2) - \log(A_1)}{\log(F_2) - \log(F_1)} \quad \text{Eq. 7}$$

$$\frac{1.3365 - 0.4771}{8.6021 - 7} = \frac{0.8593}{1.6021} = 0.5364 \quad \text{Eq. 8}$$

The slope for most copper cables is around 0.5. (If only one attenuation data point is available, assuming 0.5 for a slope will get you close to the actual attenuation at other frequencies.)

With the slope available it is now possible to calculate the offset using *Equation 9*. The result as calculated at 400 MHz is shown in *Equation 10*.

$$offset = (log(F) \cdot slope) - log(A) \quad \text{Eq. 9}$$

$$(8.6021 \times 0.5364) - 1.3365 = 3.278 \quad \text{Eq. 10}$$

With the slope and offset now available, it is possible to calculate the attenuation per unit-distance at any frequency using *Equation 11*.

$$Attenuation(dB) = 10^{(log(Frequency) \times slope - offset)} \quad \text{Eq. 11}$$

Note: Because all the previous calculations were based on 100 meter distances, the numbers generated here give the attenuation for 100 meters of RG59 cable at any frequency. These numbers may be scaled linearly to get the attenuation at any other length of cable.

The waveforms in *Figures 3* and *4* have symmetrical rise and fall times and therefore only contain odd harmonics. For the 30-MHz signal this yields harmonics at 30 MHz, 90 MHz, 150 MHz, 180 MHz, etc. The calculated attenuation for these harmonics (through 1 GHz) are listed in *Table 2*.

By applying these attenuation amounts to the specific signal components it is possible to determine the signal's spectrum at other points on the cable. These calculations were performed assuming a 100 meter length of cable to generate the spectrums shown in *Figure 5*.

By using an IFT (inverse Fourier transform) on these new spectrums it is possible to reconstitute the time domain form of the signal. If the same phase components are used with the attenuated amplitudes, the waveforms in *Figure 6* are generated (Reference 7).

Table 2. Calculated Attenuation for Belden 9659 Cable (RG59-type)

Frequency (MHz)	Nominal Attenuation	
	dB/100 Feet	dB/100 meters
30	1.64	5.40
90	2.96	9.75
150	3.90	12.8
210	4.67	15.4
270	5.34	17.6
330	5.95	19.6
390	6.51	21.4
450	7.03	23.1
510	7.51	24.7
570	7.98	26.2
630	8.42	27.7
690	8.84	29.1
750	9.24	30.4
810	9.63	31.7
870	10.0	32.9
930	10.4	34.1
990	10.7	35.3
1050	11.1	36.4

With these data rate and cable combinations, only 25% of the peak-to-peak amplitude of the D21.5 (1010101010) pattern remains after 100 meters of cable, while the K28.7 (1111100000) pattern has nearly 60% of its signal available.

Figure 7 shows the actual measured signals at the source and after 100 meters of cable. While the measured amplitudes are a close match to the calculated amplitudes, the waveshape of the K28.7 signal at the end of the cable is significantly different. The cause of this distortion is a variation in propagation velocity verses frequency known as dispersion.

Dispersion

Dispersion is a propagation characteristic more commonly linked to optical fibers. This causes light

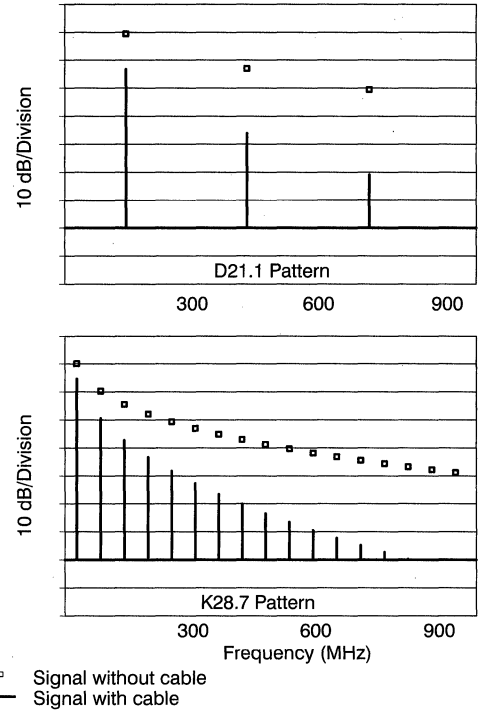


Figure 5. Spectrum of Synthesized D21.5 and K28.7 Patterns After 100m of RG59 Cable

at different wavelengths to propagate at different rates through the fiber.

This same phenomenon exists in copper cables where higher frequency signals propagate faster than slower frequency signals. This variation in propagation is caused by two different phenomena: a change in dielectric constant of the cable dielectric with frequency, and a change in the reactance of the cable with frequency.

Dielectric Dispersion

Recall from the "Driving Copper Cables with HOTLink" application note that for coaxial cables and stripline transmission lines

$$V_P = \frac{1}{\sqrt{\varepsilon_r}} \qquad \text{Eq. 12}$$

If the dielectric constant (ε_r) for a transmission line remains constant across all frequencies, the signal

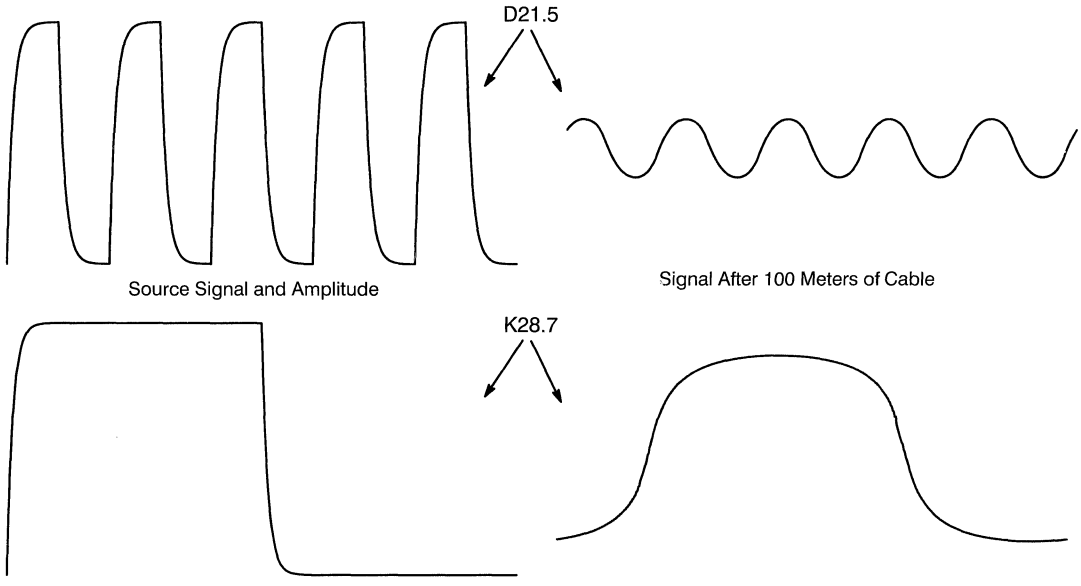


Figure 6. Synthesized D21.5 and K28.7 Waveforms with Simulated Cable Attenuation

spectral components will propagate down the transmission line at the same rate. Unfortunately, many dielectrics are not stable with frequency. Dielectrics such as bakelite, glass, rubber, and PVC (polyvinyl chloride) exhibit from several percent to 10s of percent change in dielectric constant over the 1-MHz to 1-GHz frequency range. Common circuit board materials also are not stable with frequency. *Figure 8*

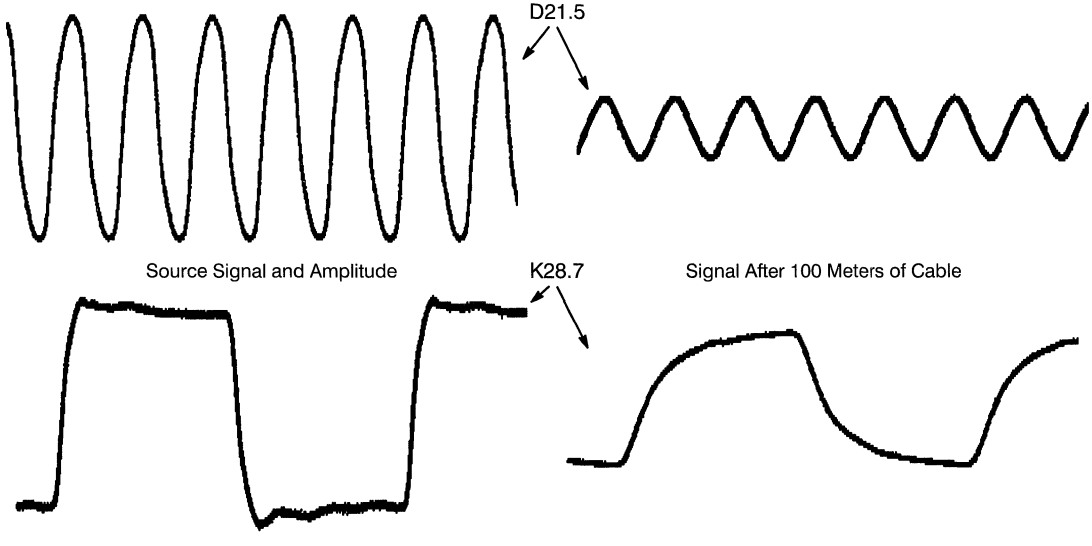


Figure 7. Measured D21.5 and K28.7 Waveforms with 100m of RG59 Cable

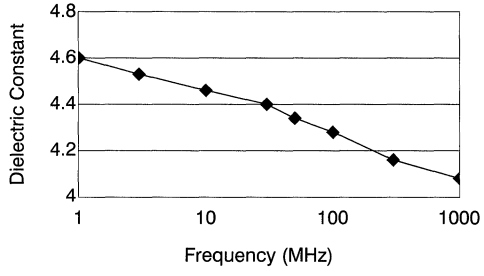


Figure 8. Dielectric Constant of G10/FR4 Circuit Board Laminate

shows how the dielectric constant (ε_r) changes in G10/FR4 circuit board laminate (Reference 8).

Applying these ε_r values to the spectral components in the K28.7 and D21.5 signals shown in *Figures 5* and *6* yields signal components traveling at the rates listed in *Table 3*. When sending an actual data stream, these and other signal components are present in the transmission line at the same time. The D21.5 fundamental (150 MHz, equal to the bit-rate) has a wavelength in the transmission line of only 0.974 meters. Because the K28.7 fundamental (15 MHz, equal to the byte-rate) is traveling 1.3% slower than the D21.5 fundamental, this signal component will lag the D21.5 fundamental by 90° of phase (equal to 50% of one bit time) after only 16.2 meters of transmission line.

Table 3. D21.5 Signal Propagation Rates

Frequency	ε_r	Vp (%)	Vp (m/second)
15 MHz	4.45	47.4%	1.422×10^8
45 MHz	4.35	47.9%	1.438×10^8
150 MHz	4.22	48.7%	1.460×10^8
450 MHz	4.12	49.3%	1.478×10^8

The 90° phase point was selected because it is equivalent to 100% jitter in the received bits. In reality other signals components will cross this 90° phase point in much less than the 16 meter length. This is because a normal data stream contains other signal components both higher and lower in frequency than the two selected here. Since these other signals components are traveling both slower and faster than the 15 MHz and 150 MHz signals used in this example, the 90° point will be reached with a much shorter transmission line. Due to the limited energy present in each of these signal components, they individually cannot close up the received signal eye.

Other Dispersion Factors

Good RF-grade cables are usually made with stable dielectrics; i.e., those that exhibit only minor changes in dielectric constant over frequency. These cables are usually constructed using dielectrics based on polyethylene, polypropylene, polyolefin, polystyrene, and various Teflon® derivatives. These dielectrics vary in dielectric costant by less than 0.5% from 100 Hz through 10 GHz. Calculations for dielectric based dispersion show little interaction even after hundreds of meters of cable, yet these cables still exhibit dispersion. The dispersion effect in these cables is caused by the variation in reactance that occurs in the cable with changing frequency. The dispersion caused by reactance (per unit length of cable) is much smaller than that caused by non-frequency stable dielectrics. This allows cables based on stable dielectrics to be used for much longer signal transmission.

In reality the calculation of velocity of propagation in *Equation 12* is a simplified form that only assumes first order effects. A proper calculation must take into account all four distributed properties (R, G, C, and L) in a transmission line. This is normally described as the complex propagation constant γ, and is calculated using *Equation 13* (Reference 4).

$$\gamma(\omega) = \alpha + j\beta = \sqrt{(R + j\omega L)(G + j\omega C)} \qquad \text{Eq. 13}$$

The complex propagation constant γ consists of a real portion α, representing the attenuation of the signal, and an imaginary portion $j\beta$, representing the angular velocity of the signal. Both of these are per unit length of the transmission line. This shows that the propagation rate is based on all four parameters, not just the dielectric constant of the line.

With RF-grade dielectrics the dielectric constant remains stable over frequency and thus does not effect the propagation rate. The conductance parameter does increase at a rate directly proportional to frequency. At frequencies over a few megahertz its effect, relative to that of $j\omega C$, is so small that it is usually discarded. With the long transmission lines

considered here this very small effect is still important. The resistance parameter also increases with frequency. This change in resistance is caused by the previously described skin effect, where the transmission line resistance is affected by the uneven current distribution.

The distributed inductance is also affected by frequency. The total inductance is a sum of the external inductance (that present between the two conductors of the transmission line) and the internal or self inductance of the conductors (assuming a dielectric free of magnetic properties).

Each of these pieces has a small effect on the total propagation rate of signals. Most of these effects are only observable with cables extending for tens or hundreds of meters.

Equalization

Equalization is the application of frequency selective gain or attenuation to compensate for distortion. Equalization is used in analog audio, analog video, and digital signal transmission systems to compensate for characteristics of the system and the distortion created by the operating environment.

For HOTLink-based communications, the primary cause of signal distortion is the non-linear characteristics of the interconnecting cable. This cable attenuates the high-frequency signal components much more than the low-frequency signal components, and introduces a frequency selective phase delay into the signal.

As the length of the interconnecting cable increases, so does the signal distortion. At some point the distortion becomes so great that the HOTLink receiver is no longer able to correctly recover the serial datastream. While there is still sufficient amplitude available in the signal, the data-dependent jitter (DDJ) exceeds the jitter tolerance of the HOTLink receiver (typically >90%). To allow reliable communications with these long cables it is necessary to "equalize" the the cable.

Equalization Circuits

Equalization can take many forms. For many low-frequency circuits, equalization often uses a combination of active and passive components to create frequency selective filters that provide specific amounts of gain or attenuation for a signal. These same filters may be made to automatically adapt to different cable, frequency, and distance combinations.

At higher operating frequencies (such as those used with HOTLink), the design and implementation of active filters becomes more difficult, and equalization is usually performed using only fixed passive components, followed by a non-frequency-selective amplifier. This provides the lowest cost form of equalization, but is not as flexible as an adaptive/active equalization circuit.

With a passive equalizer, the only functions that the circuit can provide are attenuation and phase change—they cannot provide gain (peak amplitude of some signals may increase, but this is due to alignment of the signal component phasors). To equalize a copper cable, the circuit must operate in a manner opposite that of the interconnecting cable. This effectively means a high-pass filter that delays the phase of high-frequency signal components.

Many such circuits are available, all with different topologies and characteristics. A simple equalizer circuit recommended for HOTLink use is explained in detail in the following example.

Equalizer Example

A pair of equalizers suitable for use with HOTLink are shown in *Figure 9*. The Bridged-H circuit is a balanced circuit that operates with balanced transmission lines. This balanced equalizer may also be used with unbalanced cables if placed on the balanced side of a balun coupling transformer.

The Bridged-T circuit is an unbalanced form of the Bridged-H equalizer. This circuit is designed for use with unbalanced transmission lines. When used with a HOTLink receiver that is transformer coupled, this circuit must be used in the unbalanced portion of the transmission line. It may be used with coaxial (or other unbalanced) cables by placing the

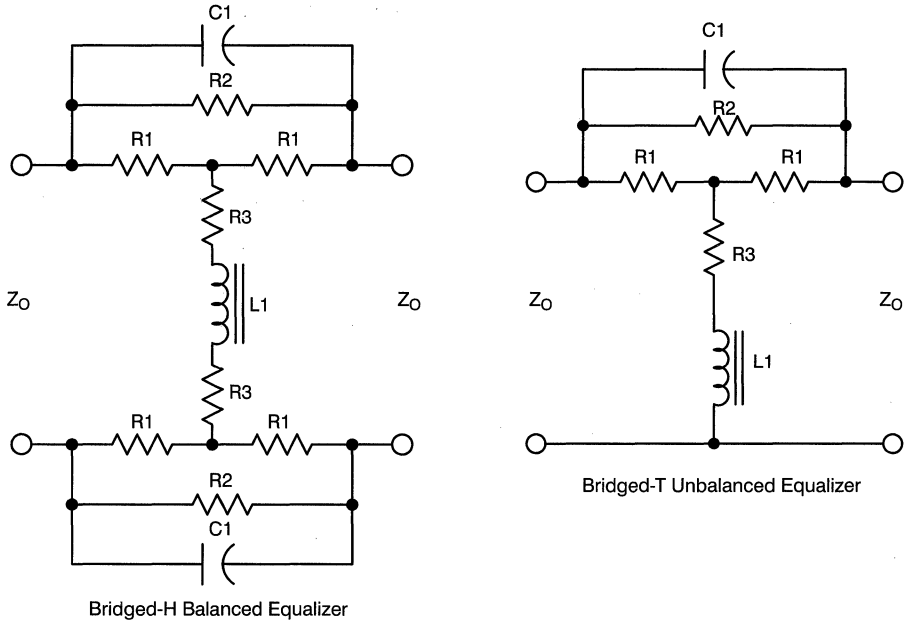


Figure 9. Constant Impedance Equalizer Circuits

circuit between either end of the transmission line and the coupling transformer.

Both of these circuits are AC-forms of a fixed-attenuator or "pad". A pad is often used for impedance matching or attenuating between a source and destination, with minimal parts count and minimum loss. The equalizers in *Figure 9* are converted to their pad equivalent by removing the capacitors and shorting out the inductor. Unlike some pads which can perform impedance transformation, these Bridged-H and Bridged-T circuits require the input and output impedances to be the same.

These equalizers, when properly implemented, appear across a wide frequency range as a DC resistance at the end of a cable. For frequencies at or near DC, the gain (insertion loss) is determined only by the resistors. As the frequencies approach the active region of the filter, the reactive nature of the capacitor starts to have an effect. The higher frequencies see less reactance and are passed through the capacitor with minimal attenuation. The inductor is selected to exactly match (but with increasing reactance) the frequency response characteristics of the capacitor(s).

The component values for these circuits are determined by the specific cable type selected, the frequency of operation, and the desired distance of operation. The design equations for both structures are detailed in *Table 4*. Because the balanced Bridged-H circuit is based on the unbalanced Bridged-T (and all values for it may be derived from the Bridged-T equations), only the Bridged-T circuit will be explained in detail.

Table 4. Equalizer Equations

Component	Bridged-T	Bridged-H
R1	Z_O	$Z_O/2$
R2	Z_O*X	$(Z_O*X)/2$
R3	Z_O/X	$Z_O/(2*X)$
C1	$L1/(Z_O^2)$	$(2*L1)/(Z_O^2)$
L1	$C1*Z_O^2$	$(C1*Z_O^2)/2$

Z_O = characteristic impedance of cable,
X = see *Equation 15.*

Equalizer Example

The R1 value is the easiest to determine. For the Bridged-T circuit it is equal to Z_O. For the RG59 cable documented previously (Z_O=75Ω), the R1 value would be 75Ω.

The relationship for R2 and R3 determines both the DC-gain (loss) of the equalizer and the correction attenuation slope. To keep a constant impedance, it is necessary for

$$Z_o = \sqrt{R2 \cdot R3} \qquad \text{Eq. 14}$$

The gain is determined by the ratio of each resistor to the filter impedance, and a gain constant X. The gain constant (X) determines how much insertion loss the filter should have at low (near DC) frequencies, and is determined using *Equation 15*.

$$X = \sqrt{10^{\left(\frac{dBAttenuation}{10}\right)}} - 1 \qquad \text{Eq. 15}$$

Attenuation Slope

This same gain constant also determines the slope of the attenuation curve in the active region of the filter. For equalization purposes the gain constant must be determined by the slope of the transmission line attenuation over the main frequency range of interest.

The transmission line presents an attenuation verses frequency slope that increases with cable length. *Figure 2* shows that the source (cable) attenuation function is linear when plotted in log/log space (attenuation verses frequency). To flatten the system frequency response the equalizer must then present an attenuation verses frequency slope that is equal in magnitude but opposite in slope to that of the cable.

Unfortunately a single pole filter (like that used here) can only generate a correction slope of at most −20 dB/decade. The source signal attenuation also increases at a logarithmic rate per decade rather than a linear rate per decade. This means that the correction applied to the signal can only be a coarse approximation rather than a perfect correction.

Using the RG59 cable documented earlier, and assuming a cable length of 100 meters and a data rate of 300 Mbaud, it is possible to calculate the approximate attenuation slope (in dB/decade) that the equalizer must attempt to correct. The goal is to have the low-frequency content of the received signal match the high-frequency content at a specific length of cable.

The data from *Table 2* identifies that the attenuation at 150 MHz (the bit-rate equivalent sinusoidal frequency of 300 Mbaud) is 12.8 dB for a 100 meter cable. At the 30 MHz frequency (the byte-rate equivalent sinusoidal frequency) the attenuation is 5.4 dB. These two points are then used to determine the necessary correction attenuation slope (in dB/decade) using *Equation 16*. Entering these values into *Equation 16* yields an attenuation slope of 10.61 dB/decade.

$$slope = \frac{A1 - A2}{log(F1) - log(F2)} \qquad \text{Eq. 16}$$

Equalization Slope

To equalize the cable it is necessary to present a correction having a matched slope but starting from the bit-rate fundamental frequency. This slope is controlled only by the R2/R3 resistors, with the frequency being determined by C1/L1. As the R2/R3 resistor ratio varies (as set by the gain constant X) the attenuation slope varies from between zero and 20 dB/decade. The necessary gain constant may be determined directly using *Equation 17*. Using the previously calculated source slope yields a gain constant of 2.224.

$$X = \sqrt{3.9 \times \left[\tan\left(slope \times \frac{\pi}{40}\right)\right]^{2.49}} \qquad \text{Eq. 17}$$

Note: This equation was derived from empirical data. Its function matches simulated response curves to within 0.15 dB for the entire 0 to 20 dB/decade range.

With the gain constant now available, the values of R2 and R3 may be determined. Using the equations from *Table 4* for R2 and R3, these calculate to R2=166.8Ω and R3=33.7Ω. Inserting this same gain constant into *Equation 18* sets a DC gain of −10.17 dB.

$$dBattenuation = 10 \times \log[(X + 1)^2] \qquad \text{Eq. 18}$$

Center Frequency

The L1 and C1 components are used both to select where the signal attenuation occurs, and to keep the equalizer impedance constant. To maintain the a constant impedance in the equalizer, the product of the shunt and bridge impedances must always equal the square of the characteristic impedance. In terms of L1 and C1 this can be reduced to the relationship in *Equation 19*.

$$Z_O = \sqrt{\frac{L1}{C1}} \qquad \text{Eq. 19}$$

Setting the roll-off point for the high-pass filter is not quite as intuitive. At first glance the equalizer appears as a single-pole filter yielding a fixed 6 dB/octave or 20 dB/decade attenuation below a cutoff frequency. This is the actual filter response when set for a DC gain of 0 (DC loss= ∞) by removing R2 and shorting R3. In this configuration the -3 dB cutoff frequency is determined using *Equation 20*.

$$f_C = \frac{1}{2\pi\sqrt{L1 \cdot C1}} \qquad \text{Eq. 20}$$

Adding R2 and R3 back into the circuit however changes the slope of the attenuation curve, moves the upper cutoff frequency, and adds a lower cutoff frequency point. *Figure 10* shows the gain and phase response for this equalizer implemented with an arbitrarily selected (but properly balanced) C1/L1 pair of 200 pF and 1125 nH. The attenuation slope is correct, but the location within the frequency spectrum is not. An examination of the phase response curve shows that it peaks at the midpoint of the active region of the filter.

The capacitor C1 is responsible for the location of the attenuation curve within the frequency spectrum. As the capacitance is decreased, the curve is shifted higher in frequency, but with an identical slope. The correct capacitor (and corresponding inductor) are selected when the line determined by the equalizer attenuation slope intersects the bit rate frequency (150 MHz for this example) at 0 dB.

Unfortunately, any simulation or measurement will show that the attenuation slope is not linear at the upper and lower ends of the active region of the filter. The only point on the gain curve whose slope actually matches the desired correction slope is at the midpoint of the curve, located at the same frequency

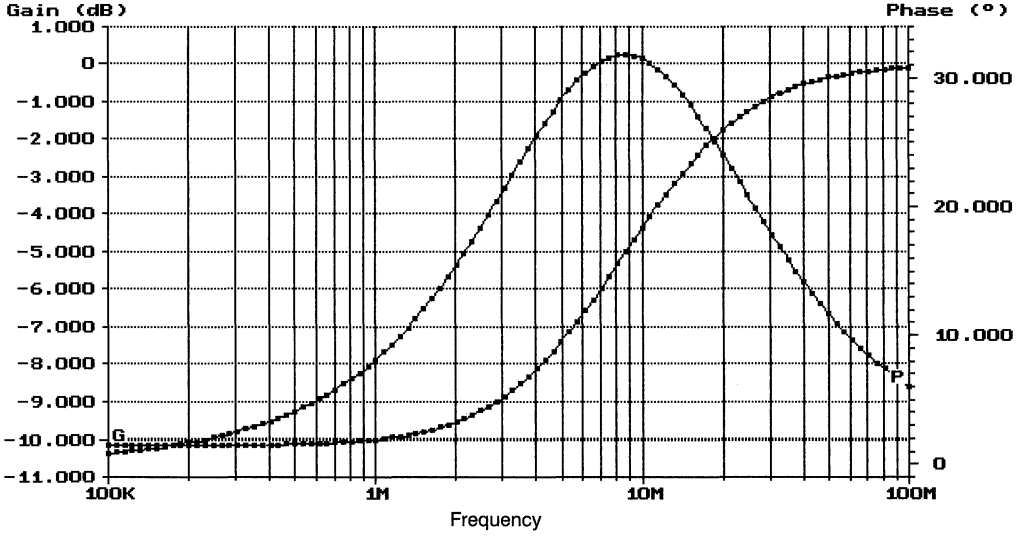


Figure 10. Gain/Phase Plot for Initial C1/L1 Values

as the peak in the phase response (8.5 MHz). The attenuation at this point is exactly half the DC attenuation (−5.08 dB).

The filter response of the present circuit is obviously too low for proper compensation of a 300 Mbaud data stream. What is necessary is to shift this midpoint to a different frequency. This new midpoint intercept frequency is calculated using *Equation 21*. Using this equation with the current bit-rate frequency (150 MHz), DC gain (−10.17 dB), and equalization slope (−10.61 dB/decade) yields a new center frequency of 49.8 MHz.

$$F\_new = 10^{\left(\log(F\_bit\_rate) - \frac{DC\_Gain/2}{slope}\right)} \quad \text{Eq. 21}$$

To determine the correct C1 and L1 values that will center the filter response through this point requires determining the magnitude of the reactance phasor at this point. The reactance at this center point in the filter response remains the same with any properly matched C1/L1 pair. In the gain/phase plot in *Figure 10*, the center frequency is at 8.5 MHz. The impedance phasor magnitude for the bridge (R2/C1) and shunt (R3/L1) paths are calculated using *Equations 22* and *23* respectively.

$$X_B = \frac{1}{\sqrt{\frac{1}{R2^2} + (2\pi f \cdot C1)^2}} = 81.6\Omega \quad \text{Eq. 22}$$

$$X_S = \sqrt{R3^2 + (2\pi f \cdot L1)^2} = 68.9\Omega \quad \text{Eq. 23}$$

These X_B and X_S values are the magnitudes of the complex impedances present in the R2/C1 and R3/L2 component pairs respectively. Solving for the specific C1 and L1 components at the desired 49.8 MHz midpoint frequency involves converting the impedance vectors into their real and imaginary components, and determining what size component will yield the proper reactance at the specified center frequency. The calculations for C1 and L1 are shown here in *Equations 24* and *25*.

$$L1 = \frac{\sqrt{X_s^2 - R3^2}}{2\pi f} = 192.2\ mH \quad \text{Eq. 24}$$

$$C1 = \frac{\sqrt{\frac{1}{X_B^2} - \frac{1}{R2^2}}}{2\pi f} = 34.2\ pF \quad \text{Eq. 25}$$

Placing these new C1 and L1 components into the Bridged-T equalizer yields the filter response shown in *Figure 11*. The slope of the curve (in dB/decade)

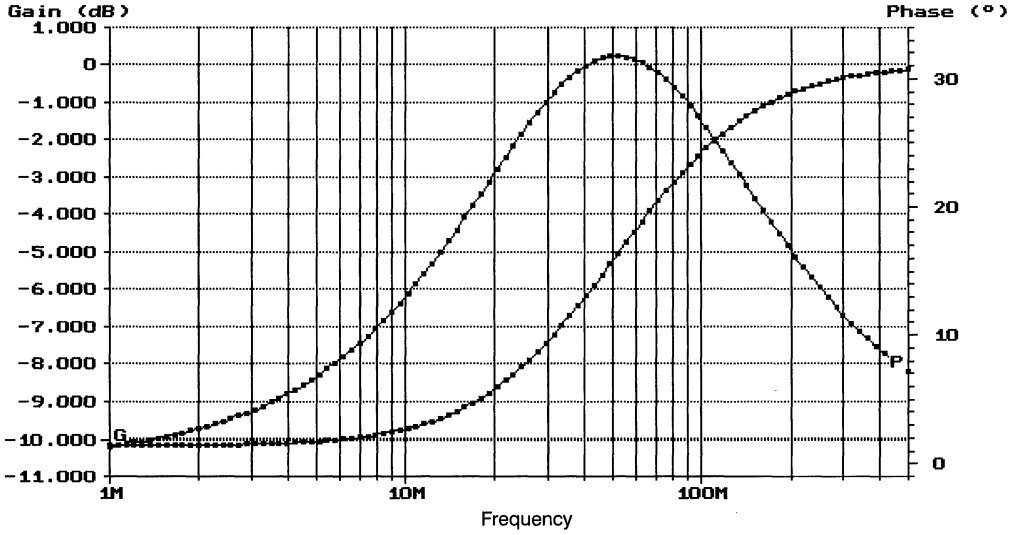


Figure 11. Gain Phase Plot for Final C1/L1 Values

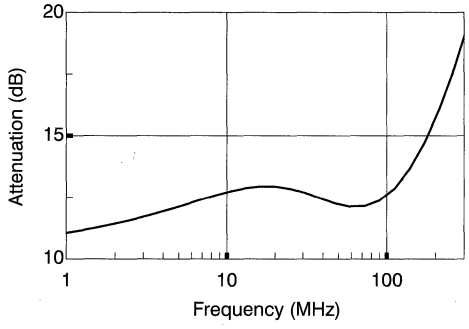


Figure 12. Combined Cable and Equalizer Attenuation

remains the same, but now the phase response peak occurs near 50 MHz.

Composite Response

Figure 12 shows how close this equalization matches the cable's frequency response. This curve is a sum of the cable and equalizer attenuations at each frequency point. Note that the link response (100 meters of cable and the equalizer) does not vary by more than 2 dB for over two decades of frequency spectrum. Once the signal spectral components are above the bit-rate frequency of the filter, the cable attenuation becomes dominant and the attenuation slope increases dramatically. Slight alterations of the equalizer slope and frequency intercept can modify this curve to meet specific frequency response and flatness requirements.

Implementation Constraints

While the numeric calculations allow a design to be implemented on paper, bring such a design into the real world is much different. Finding components with even 1% accuracy can be difficult if not impossible. Parasitic reactances present in any component also effect the response of the equalizer circuit. This means that even the best equalizer will wind up being a number of compromises.

Resistors

The selection of resistor values is probably the easiest to make. These components are available in wide ranges of values and tolerances. For most equalizer implementations, these parts should be 1% tolerance components.

Because of the wide frequency range that the equalizer must cover, care should also be exercised in the selection of the type of resistive element used. Carbon composition and carbon film resistors have significant capacitive parasitics and should not be used in sizes over 100Ω in equalizers of this type. A better choice here would be metal film resistors.

The physical size of the component also makes a difference. Generally the smaller the components physical size, the lower the inductive and capacitive parasitics present.

Inductors

The inductor is the most difficult component to select, primarily because they are manufactured in so few standard sizes. In the range from 10 nH through 2000 nH (the range most likely to be used with HOTLink) all manufacturers provide the same series of part values in each decade of size. These values are 10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68, and 82. All other standard sizes are found by multiplying these values by 10, 100, 1000, etc. Custom sizes are available from some manufactures, but generally at a significant cost difference.

Another problem that plagues most inductors is a low series resonant frequency. For the equalizer to operate correctly (within its designed range of operation), the inductor must continue to provide increasing amounts of reactance with increasing frequency. This means making sure that the series resonant frequency of the inductor is greater than the bit-rate frequency of the data stream. The best inductors for this are generally made from a multilayer ceramic construction.

The last concern is manufacturing tolerance. Unlike resistors where 1% tolerance parts are low in cost and widely available, the common tolerance for inductors is 10%. A few manufacturers also offer 5% and 2% tolerance parts.

Capacitors

The choice of capacitors is almost dictated by the available sizes of inductors, and the small quantity of capacitance required for most equalizers. This

will generally fall in the 10 to 200 pF range. The majority of all chip capacitors in this range are made with a temperature stable low-K dielectric known as either NP0 or C0G. Other high-K dielectrics should not be used, both for their instability over temperature and for the ferroelectric effect these high-K dielectrics exhibit.

While capacitors also have a series resonant frequency, it is not generally a concern when using the types and sizes of capacitors required for these equalizers. In almost all cases the series resonant frequency is well above the bit-rate frequency and therefore of only minor concern.

Board Layout

Just as incorrect component selection can greatly effect the frequency response of an equalizer, so can a poorly implemented layout. The circuit traces, pads, and vias all have an effect on the circuit operation. The following guidelines should be applied to minimize these effects.

- Use as short of traces as possible to minimize the trace inductance and capacitance.
- Keep all components in close proximity to each other.
- Minimize the number of vias. These structures can be routed on a single layer without vias.
- For the Bridged-H balanced equalizer, keep routing symmetrical to keep the parasitics balanced.

Conclusion

Communications on electrically long transmission lines are possible with many types of media. How far a signal may be reliably transmitted is a function of many driver, cable, filter, and receiver characteristics. Application of equalization filters can allow communication over distances well beyond that of non-equalized systems. These equalizers may be implemented with a minimal number of low cost passive components.

References

1. True, Kenneth M., *Long Transmission Lines and Data Signal Quality*, AN808, National Semiconductor
2. Orr, William I., *Radio Handbook*, 23rd Edition, SAMS, 1992
3. Belden Master Catalog, Cooper Industries, Inc., 1992
4. True, Keneth M., *Data Transmission Lines and Their Characteristics*, AN–806, National Semiconductor
5. True, Keneth M., *Long Transmission Lines and Data Signal Quality*, AN–806, National Semiconductor
6. *Fibre Channel Standard*, ANS X3.230-1994, American National Standards Institute, 1994
7. Ramierez, Robert W., *The FFT, Fundamentals and Concepts*, Tektronix, Inc. 1985
8. AdCore Product Anouncement, GIL Copper Clad Laminates, Alpha Corporation, 1995

HOTLink™ CY7B933 $\overline{\text{RDY}}$ Pin Description

This application note describes the behavior of the $\overline{\text{RDY}}$ (Ready) pin in several modes of operation: Encoded, Bypass, and BIST (Built-In Self-Test). The $\overline{\text{RDY}}$ pin indicates the status of the HOTLink™ Receiver control logic and output pins. Its function and timing are dependent on the state of the MODE, $\overline{\text{BISTEN}}$ (Built-In Self-Test Enable), and RF (Reframe) pins. The following sections describe $\overline{\text{RDY}}$ behavior in detail.

Normal $\overline{\text{RDY}}$ Timing

The HOTLink CY7B933 datasheet specifies signal transitions for the receiver in bit-times relative to the rising edge of CKR. A bit-time refers to the period of the internal receiver bit-rate clock. The period of the recovered byte-rate clock, CKR, is ten times the bit period (bit period $t_B = t_{CKR} \div 10$). In the following discussions on timing, the rising edge of CKR is referenced as bit-time zero. The next rising edge of CKR occurs ten bit-times later (unless CKR stretches due to reframing). Transitions on other signal pins are defined in bit-times relative to bit-time zero. These timing conventions are adhered to throughout this application note.

The normal timing of the $\overline{\text{RDY}}$ pin refers to its behavior in Encoded or Bypass mode with $\overline{\text{BISTEN}}$ HIGH (Built-In Self-Test disabled). In either of these modes, $\overline{\text{RDY}}$ rests HIGH in its inactive state. During its active state, $\overline{\text{RDY}}$ transitions LOW on bit-time five and then transitions HIGH on bit-time one of the next clock cycle. *Figure 1* illustrates $\overline{\text{RDY}}$ timing in relation to CKR and DATA. For the exact timing margins[1] of these signals, refer to the HOTLink datasheet. In BIST mode, $\overline{\text{RDY}}$ assumes much different behavior and timing. These differences are explained later in the sections on BIST.

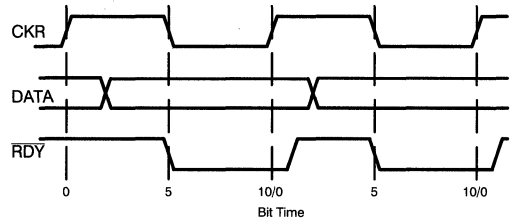


Figure 1. Normal $\overline{\text{RDY}}$ Timing

$\overline{\text{RDY}}$ in Encoded Mode

This section describes the operation of $\overline{\text{RDY}}$ in Encoded mode (MODE = LOW). In Encoded mode, the raw ten-bit serial data is decoded in the 8B/10B decoder and then presented at the parallel output pins.

Normal Operation

The normal operation of the $\overline{\text{RDY}}$ pin in Encoded mode (MODE = LOW, RF = LOW, $\overline{\text{BISTEN}}$ = HIGH) is to signal when new data is available at the parallel output pins (Q_{0-7}, SC/$\overline{\text{D}}$, RVS). $\overline{\text{RDY}}$ pulses LOW with a 60% LOW/40% HIGH duty cycle only when new data is present at the output. The timing of $\overline{\text{RDY}}$ is optimized for a seamless interface to industry standard FIFOs (First-In First-Out memories). $\overline{\text{RDY}}$ does not pulse LOW in a field of SYNC (K28.5) characters; however, $\overline{\text{RDY}}$ does pulse LOW for the last K28.5 in the field or for any single K28.5. This behavior helps prevent a FIFO from filling with meaningless strings of SYNC characters. *Figure 2* illustrates normal $\overline{\text{RDY}}$ behavior in Encoded mode.

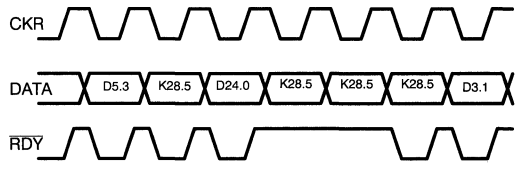


Figure 2. Normal $\overline{RDY}$ Operation in Encoded Mode

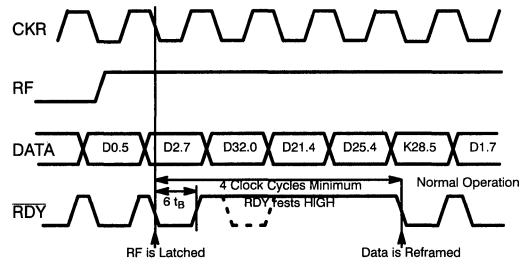


Figure 3. $\overline{RDY}$ During Framing in Encoded Mode

Entering Framing

When the RF pin is asserted HIGH, the receiver byte framer is enabled and the $\overline{RDY}$ pin leaves normal Encoded mode operation. The receiver latches the RF signal on the falling edge of CKR. When RF is latched HIGH, $\overline{RDY}$ is forced HIGH one bit time after the next rising edge of CKR (approximately $6t_B$ later). The exception to this is when there is a K28.5 in the framer when RF is asserted HIGH. In this case, an additional $\overline{RDY}$ pulse will occur after RF is latched HIGH. $\overline{RDY}$ will then pulse LOW when the data byte boundary is framed to an incoming SYNC character (K28.5). The latency of the receiver data pipeline and control logic insure that $\overline{RDY}$ will not pulse LOW any earlier than the fourth clock cycle after RF is latched HIGH. External framing logic should be designed to examine the $\overline{RDY}$ pin only after the 4 clock cycle delay.

After the data has been framed, $\overline{RDY}$ will assume its normal Encoded mode behavior (pulsing LOW for every character except strings of K28.5s). If RF remains HIGH, the framer still continues to frame the data to any K28.5 pattern found in the data stream. If RF is asserted HIGH for more than 2048 REFCLK cycles, the framer converts to a double-byte framer requiring two K28.5s within five bytes for framing. The function and timing of $\overline{RDY}$, however, remain unchanged. The timing of $\overline{RDY}$ while entering framing is outlined in *Figure 3*.

Leaving Framing

When RF is deasserted, the framer is disabled and the $\overline{RDY}$ pin assumes its normal Encoded mode operation. If the data was framed during the assertion of RF, $\overline{RDY}$ will have already assumed its normal operation. If the framer is disabled without having framed the data, one clock cycle will pass before $\overline{RDY}$ assumes normal operation. *Figure 4* shows the framer being disabled before the data is framed. $\overline{RDY}$ resumes normal operation one cycle after RF is latched LOW.

$\overline{RDY}$ in Bypass Mode

This sections describes the operation of $\overline{RDY}$ in Bypass mode (MODE = HIGH). In Bypass mode, the raw ten bit serial data bypasses the 8B/10B decoder and is presented at the parallel output pins.

Normal Operation

The normal operation of the $\overline{RDY}$ pin in Bypass mode (MODE=HIGH, RF=LOW, $\overline{BISTEN}$=HIGH) is to signal when a data pattern matching K28.5 character is present on the receiver's parallel output pins (Q_{a-j}). $\overline{RDY}$ will remain HIGH during all other data patterns. *Figure 5* shows an example of $\overline{RDY}$ in Bypass mode.

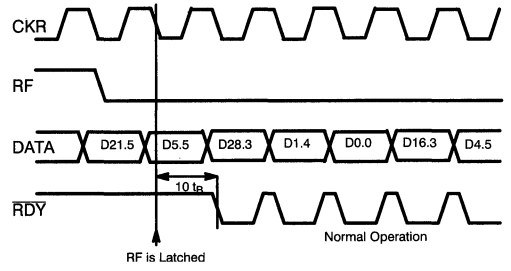


Figure 4. $\overline{RDY}$ While Leaving Framing

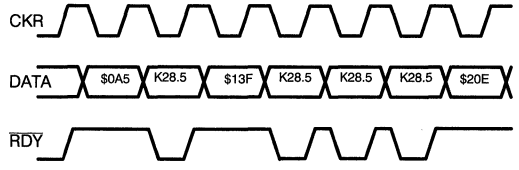


Figure 5. Normal $\overline{RDY}$ Operation in Bypass Mode

Entering Framing

The behavior of $\overline{RDY}$ while entering framing from Bypass mode is very similar to entering from Encoded mode. When RF is latched HIGH, $\overline{RDY}$ leaves normal Bypass mode operation and is forced HIGH one bit time after the next rising edge of CKR. When the framer is enabled, a LOW pulse on $\overline{RDY}$ indicates that the serial data has been framed to an incoming SYNC character (K28.5). The latency of the data pipeline and control logic insure that $\overline{RDY}$ does not pulse LOW any earlier than the fourth clock cycle after RF is latched HIGH. External framing logic should be designed to examine the $\overline{RDY}$ pin only after the 4 clock cycle delay. After the data has been framed, $\overline{RDY}$ assumes its normal Bypass mode behavior (pulsing LOW only on K28.5 characters). While RF is HIGH, the framer continues to frame the data to any K28.5 pattern in the data stream. The timing of $\overline{RDY}$ while entering framing from Bypass mode is outlined in *Figure 6*.

Leaving Framing

When RF is deasserted (LOW), the framer is disabled and the $\overline{RDY}$ pin assumes normal Bypass mode behavior. If the data was framed during the assertion of RF, $\overline{RDY}$ will have already assumed its normal operation. If Reframe is exited without having framed the data, one clock cycle passes before $\overline{RDY}$ assumes normal operation. *Figure 7* shows RF deasserted before the serial data has been framed.

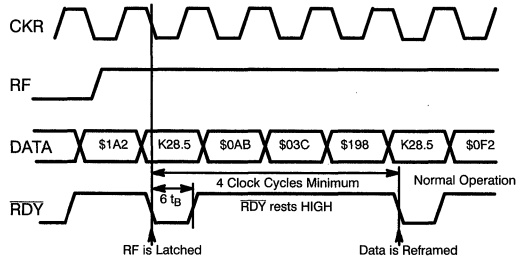


Figure 6. $\overline{RDY}$ During Framing in Bypass Mode

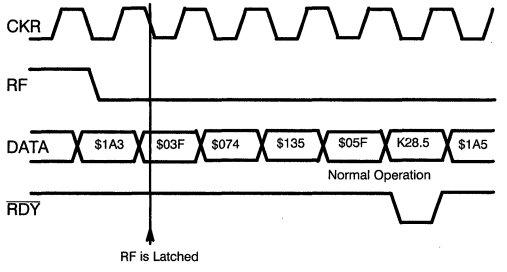


Figure 7. $\overline{RDY}$ While Leaving Framing

$\overline{RDY}$ and CKR Stretching

During framing (RF = HIGH), $\overline{RDY}$ and CKR may stretch as the byte boundary is synchronized to an incoming K28.5 character. If a K28.5 pattern is found in the serial data stream that is not aligned with the current byte boundary, the framer will realign the phase of CKR so that the receiver shift register properly deserializes the K28.5 character (and the following data). The HIGH or LOW phase of CKR and $\overline{RDY}$ will be stretched so that these signals maintain proper byte synchronization with the data. *Figure 8* shows $\overline{RDY}$ and CKR being stretched during framing due to a K28.5 character in the data stream. In this example, RF is held HIGH so that the framer remains enabled after has $\overline{RDY}$ assumed its normal operation according to the MODE pin (Encoded mode). The period of $\overline{RDY}$ and CKR

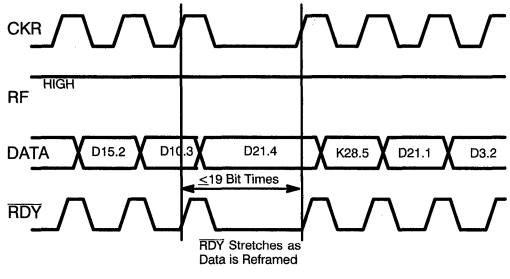


Figure 8. $\overline{RDY}$ and CKR Stretching (Encoded Mode)

may stretch up to a length of 19 bit-times depending on the position of the K28.5 character relative to the old byte boundary. Note that the K28.5 character comes out of the receiver one cycle after the CKR and $\overline{\text{RDY}}$ stretch due to the receiver pipeline.

$\overline{\text{RDY}}$ in BIST Mode

The Built-In Self-Test (BIST) feature provides a simple but exhaustive method for testing the integrity of the physical link. BIST Mode is entered by asserting the $\overline{\text{BISTEN}}$ pin LOW in either Encoded or Bypass mode. $\overline{\text{RDY}}$ has two normal modes of operation while in BIST. $\overline{\text{RDY}}$ initially rests HIGH when BIST is entered, signaling that the BIST logic has not started checking the received data. When a valid start of BIST sequence is received, the $\overline{\text{RDY}}$ pin will rest LOW, indicating that BIST checking is in progress. The timing of these transitions is discussed below. For more information on BIST, consult the "HOTLink Built-In Self-Test" application note.

Entering BIST Mode

BIST mode is entered by asserting $\overline{\text{BISTEN}}$ LOW. $\overline{\text{BISTEN}}$ is latched into the receiver on the falling edge of CKR. When $\overline{\text{BISTEN}}$ is latched LOW, $\overline{\text{RDY}}$ leaves its current mode of operation (Encoded or Bypass) and is asserted LOW for one full CKR cycle. On bit-time one of the next clock cycle, $\overline{\text{RDY}}$ is forced HIGH. The BIST logic will check the incoming data stream for the start of BIST sequence (D1.0 followed by D0.0). $\overline{\text{RDY}}$ rests HIGH while the BIST logic waits for this sequence. *Figure 9* shows the behavior of $\overline{\text{RDY}}$ when $\overline{\text{BISTEN}}$ is asserted LOW.

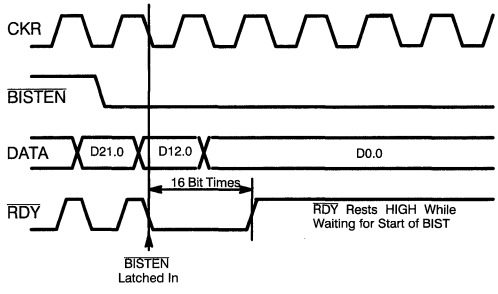


Figure 9. $\overline{\text{RDY}}$ while Entering BIST

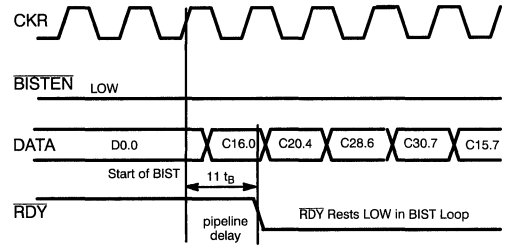


Figure 10. $\overline{\text{RDY}}$ at Start of BIST

Start of BIST

When the start of BIST pattern is found, $\overline{\text{RDY}}$ will transition LOW one bit time after CKR rises. Due to the pipeline nature of the receiver, there is a one cycle delay from when start of BIST is detected and when $\overline{\text{RDY}}$ is asserted LOW. $\overline{\text{RDY}}$ will remain LOW for the duration of BIST except to pulse HIGH for one clock cycle each time a BIST Loop starts (once every 511 bytes). *Figure 10* shows the $\overline{\text{RDY}}$ pin during the start of BIST sequence.

BIST Loop

Figure 11 shows $\overline{\text{RDY}}$ behavior once BIST checking has begun. $\overline{\text{RDY}}$ rests LOW and pulses HIGH at the start of each new BIST loop. During this pulse, $\overline{\text{RDY}}$ rises on bit time one and then falls one cycle later on bit time one. This pulse is useful for counting the number of BIST loops completed.

Leaving BIST

BIST is disabled by setting $\overline{\text{BISTEN}}$ HIGH. $\overline{\text{RDY}}$ will assume the behavior dictated by the MODE pin

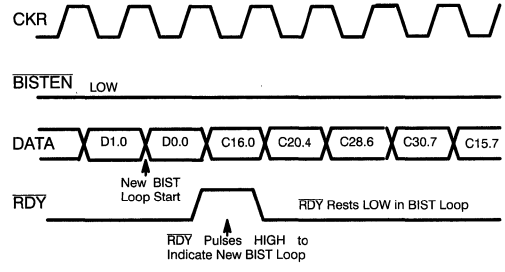


Figure 11. $\overline{\text{RDY}}$ in BIST Loop

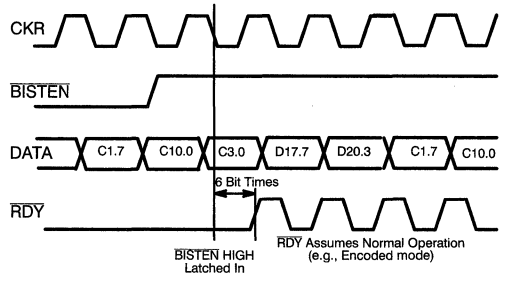


Figure 12. $\overline{RDY}$ While Leaving BIST

(Encoded or Bypass) one clock cycle after $\overline{BISTEN}$ is latched HIGH. *Figure 12* shows the $\overline{RDY}$ pin while leaving BIST Mode.

Framing While in BIST

Framing may be performed while in BIST Mode. The BIST pattern includes one alias K28.5 and several instances of byte aligned SYNC characters. If the framer is enabled (RF = HIGH), the data byte boundaries are aligned to any incoming K28.5 characters found in the serial data. $\overline{RDY}$ ceases its normal BIST behavior and rests HIGH while the framer waits for a K28.5 character. The timing for the $\overline{RDY}$ pin to be forced HIGH is the same as the timing discussed in the preceding sections on entering framing (i.e., $6t_B$ after RF is latched HIGH). When a K28.5 character is found, $\overline{RDY}$ will pulse LOW for one clock cycle. During this cycle, $\overline{RDY}$ falls on bit time five and then rises on bit time one of the next clock cycle. $\overline{RDY}$ then resumes its normal BIST behavior after one more clock cycle (see *Figure 13* and *Figure 14*).

Figure 13 shows RF asserted HIGH (framer enabled) while BIST is in the middle of checking the data. $\overline{RDY}$ initially rests LOW and then transitions HIGH when the framer is enabled. When a K28.5 character is found, $\overline{RDY}$ pulses LOW and then rests HIGH again. One cycle later, $\overline{RDY}$ transitions LOW as it resumes its normal BIST behavior (resting LOW during BIST).

Figure 14 shows $\overline{RDY}$ behavior while BIST is waiting for the start of BIST sequence. Initially, $\overline{RDY}$ rests HIGH while waiting for the start of BIST sequence. When RF is asserted HIGH, the framer checks the

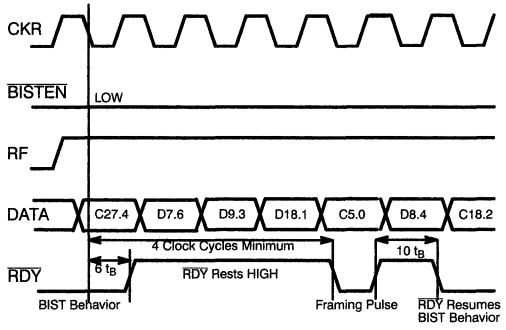


Figure 13. $\overline{RDY}$ While Framing in BIST

serial data for a K28.5 character. $\overline{RDY}$ pulses LOW when a K28.5 is encountered and then returns HIGH. $\overline{RDY}$ then returns to its normal mode of operation (resting HIGH until start of BIST is received).

If RF is deasserted before a K28.5 is found by the framer, $\overline{RDY}$ will resume its normal BIST behavior on the next clock cycle.

Enabling the framer while in BIST Mode may cause the BIST data to become temporarily misaligned. If the enabled framer encounters the alias K28.5 character in the BIST data stream, the BIST data will be aligned to the incorrect byte boundary. This will result in a large number of errors reported on the RVS (Receive Violation Symbol) pin until the data is framed again to one of the properly aligned K28.5s. If RF is asserted HIGH for less than 2048 clock cycles, the BIST data will be misaligned each time the alias K28.5 is found (once per BIST loop).

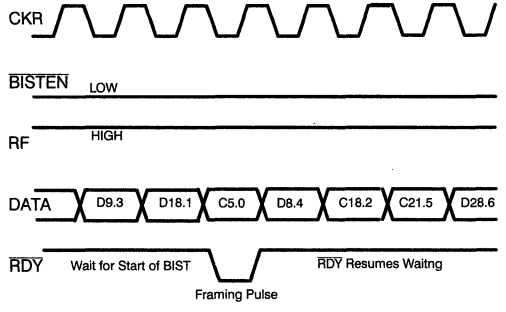


Figure 14. $\overline{RDY}$ While Framing in BIST

If RF is asserted for more than 2048 clock cycles (>4 BIST Loops), the double-byte framer will be enabled, and the framer will no longer frame the data to the alias K28.5 character.

Conclusion

The Receiver $\overline{\text{RDY}}$ pin indicates the status of the control logic and data pins in various modes of operation. The behavior and timing of the $\overline{\text{RDY}}$ pin have been optimized for easy integration with interface control logic and FIFO memories. The detailed information contained in this application note should serve as an aid when integrating the $\overline{\text{RDY}}$ pin into the interface logic.

Notes

1. Datasheet timing parameters that are defined in terms of bit times (t_B) include additional timing margin to account for internal buffer and routing delays and output load (e.g., $t_A = 2t_B$ +4/−2 ns).

CY7C42X/46X FIFO Interface to the CY7B923 (HOTLink™)

Transmitter Interface Description

This application note considers the interface between a Cypress CY7B923 (HOTLink™) Transmitter and generic FIFOs. Minimal interface logic is required to achieve a high-performance interface. A block diagram of the HOTLink Transmitter and generic FIFO interface is shown in *Figure 1*.

The FIFO operates as an asynchronous data rate buffer between the HOTLink Transmitter and the data source. The data is continually read from the FIFO into the transmitter when the Transmit signal is asserted. Reading continues until the FIFO is empty.

Critical Timing Analysis

The following equations describe the critical timing relationships. They have been solved for the minimum bit time t_B. The clock period time is $10t_B$. A timing diagram is provided in *Figure 2*. The critical timing equations are shown at the bottom of the diagram.

Read Pulse Width

$$t_{PR(min)} \leq 6t_B - 3\text{ ns} - 2\text{ns} \qquad \text{Eq. 1}$$
$$t_B \geq (t_{PR(min)} + 5\text{ ns}) / 6$$

The read pulse width for the FIFO is t_{PR}.

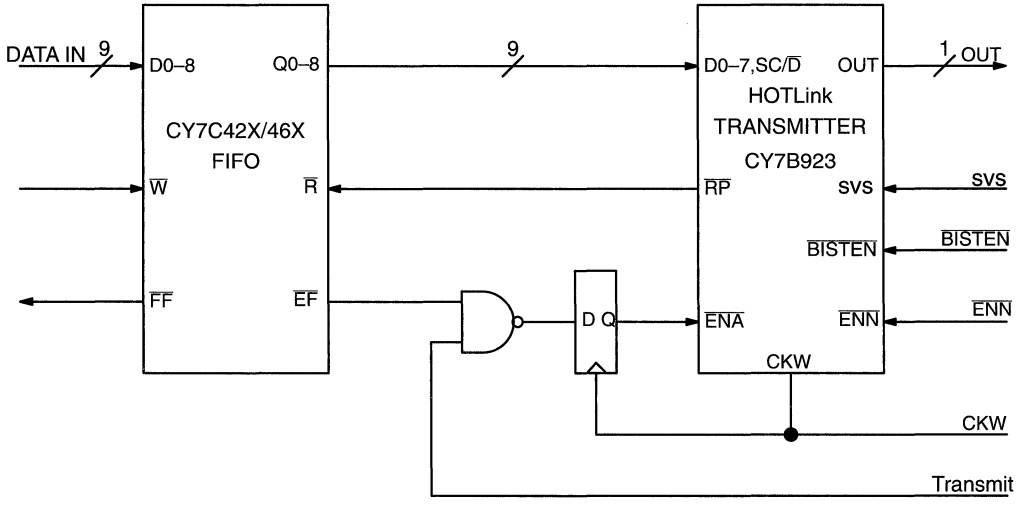
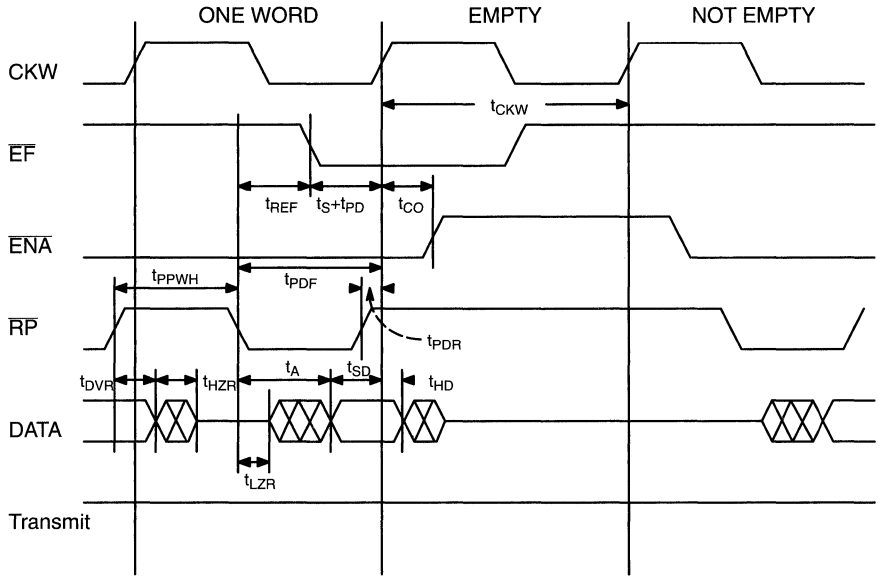


Figure 1. Transmitter Interface Diagram



Critical Timing Analysis

1. Read pulse width:
   $t_{PR(min.)} \leq t_{PDF(min.)} - t_{PDR(max.)}$
2. Read recovery time:
   $t_{RR(min.)} \leq t_{PPWH(max.)}$
3. Data set-up time:
   $t_{A(max.)} + t_{SD(min.)} \leq t_{PDF(min.)}$
4. Empty flag to register set-up time:
   $t_{REF(max.)} + t_{PD(max.)} + t_{S(min.)} \leq t_{PDF(min.)}$
5. Transmit enable to HOTLink set-up time:
   $t_{CO(max.)} \leq 10\, t_B - t_{SENP(min.)}$
6. Data hold time:
   $t_{PDR(max.)} + t_{HD(max.)} \leq t_{DVR(min.)}$

Figure 2. Interface Timing Diagram

Read Recovery Time

$$t_{RR(min.)} \leq 4t_B - 3\text{ ns} \qquad \text{Eq. 2}$$
$$t_B \geq (t_{RR(min.)} + 3\text{ ns}) / 4$$

The read recovery time for the FIFO is t_{RR}.

Data Set-Up Time

$$t_{A(max.)} + 5\text{ ns} \leq 6t_B - 3\text{ ns} \qquad \text{Eq. 3}$$
$$t_B \geq (t_{A(max.)} + 8\text{ ns}) / 6$$

The data access time for the FIFO is t_A and it is the basis of FIFO speed ratings.

Empty Flag to Register Set-Up Time

$$t_{REF(max.)}+t_{PD(max.)}+t_{S(min.)} \leq 6t_B-3\ n \qquad \text{Eq. 4}$$
$$t_B \geq (t_{REF(max.)}+t_{PD(max.)}+t_{S(min.)}+3\ ns)/6$$

The Empty flag delay from the FIFO is t_{REF}. The register set-up time for the external register is t_S.

Transmit Enable to HOTLink Set-Up Time

$$t_{CO(max.)} + \leq 4t_B - 8\ ns \qquad \text{Eq. 5}$$
$$t_B \geq (t_{CO(max.)}+8\ ns)/4$$

The register clock to output delay is t_{CO}. The propagation delay of the external control logic is t_{PD}.

Data Hold Time

$$t_{DVR(min.)} \geq 2\ ns \qquad \text{Eq. 6}$$

The valid data hold time from a FIFO read is t_{DVR}. HOTLink has a zero data hold time.

Table 1. Critical FIFO Timing Parameters

Parameter	FIFO Speed Rating		
	−10	−15	−20
$t_{PR(min)}$	10 ns	15 ns	20 ns
$t_{RR(min)}$	10 ns	10 ns	10 ns
$t_{A(max)}$	10 ns	15 ns	20 ns
$t_{REF(max)}$	10 ns	15 ns	20 ns
$t_{DVR(min)}$	3 ns	3 ns	3 ns

Table 1 shows the critical timing parameters for various speed grades of generic FIFOs. The FIFO timing parameters are taken from a hypothetical CY7C42X−10, a CY7C46X−15, and a CY7C46X−20.

Table 2 shows the maximum frequency of CKW associated with each of the timing equations for the different speed grades of generic FIFOs. The maximum interface operating frequency is shown in italics. A PAL20−5 (t_{PD} = 5 ns, t_{CO} = 5 ns, t_S = 2.5 ns) is used for the flag register and enable control logic.

Equation 6 is independent of the clock frequency and is satisfied by all of the considered FIFOs.

Equation 4 is the critical timing relationship for all of the FIFO speed grades. Timing margins can be increased by using faster control logic (PAL20−4).

Table 2. Maximum Transmitter Interface Frequency with Asynchronous FIFOs

Eqn. #	−10	−15	−20	Units
1	40.0	30.0	24.0	MHz
2	30.7	30.7	30.7	MHz
3	33.3	26.1	21.4	MHz
4	29.2	23.5	19.7	MHz
5	30.8	30.8	30.8	MHz
bit rate	292	235	197	Mbits/s

Summary

With available CY7C46X−15 FIFOs, the HOTLink-FIFO interface can operate at a frequency of 23.5 MHz with minimal interface logic. This corresponds to a serial bit rate of 235 Mbits/s.

When −10 FIFOs become available, the maximum interface frequency will increase to 29.2 MHz (292 Mbits/s).

Interfacing the CY7B923 and CY7B933 (HOTLink™) to Clocked FIFOs

Introduction

This application note describes the interfacing issues between the Cypress CY7B923/CY7B933 (HOTLink™) transmitter/receiver and Cypress clocked FIFOs. The HOTLink-FIFO interface is capable of performing parallel bus transactions at rates of up to 33 Mbytes/s and serial transfers at rates of up to 330 Mbits/s. The FIFO serves as an asynchronous storage buffer between the data bus and the serial link.

Transmitter Interface

This section describes the design considerations of a high-speed serial transmitter with FIFO (First-In First-Out) data buffers. The interface design supports basic data transmission control and serial link testing. The transmitter design is intended to interface to a higher-level system controller responsible for handling bus transactions and the serial link protocol. The interface is a primitive building block that is easily modified to meet system requirements.

Data Path and Controller

The transmitter interface consists of a single CY7C441/3−14 clocked FIFO interfacing directly to the HOTLink Transmitter. A transmitter controller supplies the control signals to both the FIFO and the HOTLink Transmitter. The architecture of the controller is left unspecified, but it can be implemented in a PLD or FPGA. State diagrams and generic timing diagrams are provided. A block diagram of the transmitter interface is shown in *Figure 1.*

Built-In-Self-Test

The transmitter is capable of checking the functionality of the transmitter serial connection by exercising the Built-In-Self-Test (BIST) mode of HOTLink. To initiate BIST, the $\overline{\text{BISTEN}}$ pin is held LOW, resulting in the transmission of the repeating character 1010101010. The HOTLink $\overline{\text{ENA}}$ (Enable Parallel Data) pin is then pulled LOW to enable transmission of the BIST test pattern. The HOTLink Transmitter will assert the $\overline{R}_P$ (Read Pulse) pin HIGH at the beginning of BIST and will pulse it LOW once per BIST loop. During BIST, HOTLink ignores data at its parallel port and the FIFO must not perform any reads.

Resetting the FIFO

The higher-level controller should reset or clear the FIFO at power-up, before a new block of data is transmitted, or if an error occurs. Resetting the FIFO is accomplished by asserting the $\overline{\text{MR}}$ (Master Reset) pin on the FIFO LOW. Neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{\text{MR}}$. To insure that this condition is met, the interface controller must be in the IDLE state (*Figure 2*) during the entire Master Reset cycle. Proper FIFO reset also requires that $\overline{\text{MR}}$ be glitch free. The higher-level controller is responsible for coordinating the read and write ports and insuring that the reset conditions are met.

Controller State Description

For applications requiring high-speed asynchronous data buffering, the FIFO read and write ports

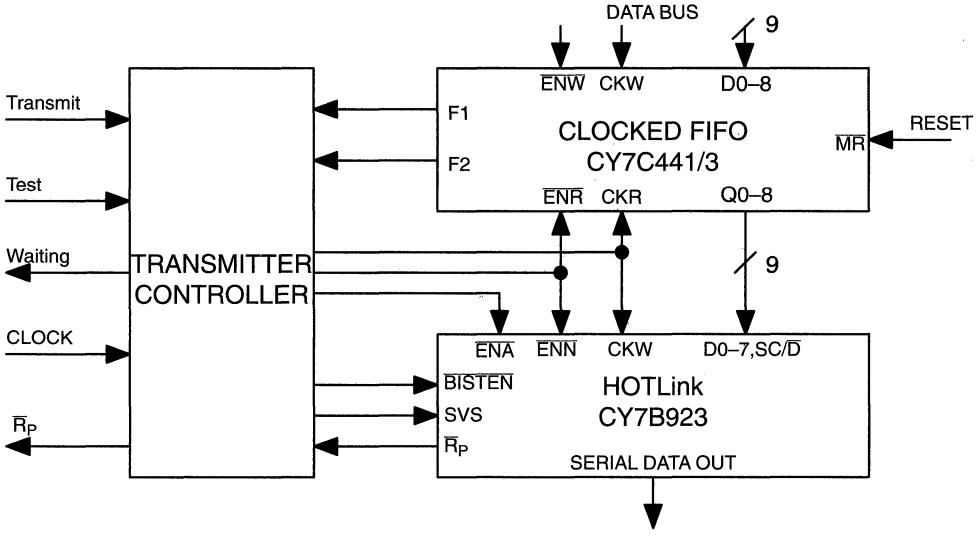


Figure 1. Transmitter Interface Block Diagram

should be controlled by separate control circuitry synchronized to the FIFO ports. The FIFO write port interfaces directly to a 9-bit data bus. Data is written into the FIFO by asserting $\overline{ENW}$ to enable the write clock (CKW). Data may be written at any time as long as the FIFO is not full (as indicated by the FIFO full flag) and a FIFO reset cycle is not in progress.

The FIFO read port interfaces to the HOTLink transmitter parallel port. Control of this interface is the focus of this section. The transmitter interface state machine controls FIFO-HOTLink data transactions and initiates the HOTLink Built-In-Self-Test. The interface state machine is under the control of a higher-level controller responsible for both the serial protocol and the data bus/FIFO transactions.

The interface controller is a simple state machine as shown in *Figure 2*. While the state machine waits in the IDLE state, HOTLink will transmit Sync fill

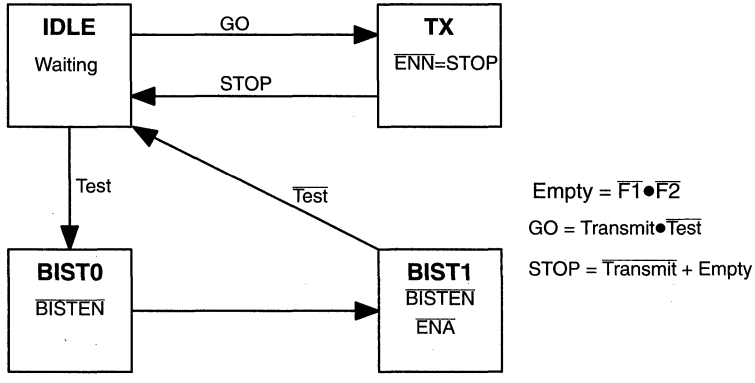


Figure 2. Transmitter Controller State Diagram

characters (K28.5). When the Transmit signal is asserted by the higher-level controller, the transmitter state machine transitions to the TX state. The TX state reads 9-bit words out of the FIFO into the HOTLink Transmitter until a Stop condition is detected (the FIFO is empty or the Transmit signal is deasserted). Reading data from the FIFO is accomplished by asserting $\overline{ENR}$ LOW. The same signal is connected to $\overline{ENN}$ (Enable Next Parallel Data) pin of HOTLink. Assertion of $\overline{ENN}$ causes data on the next rising edge of the clock to be latched into the HOTLink Transmitter. The functionality of the $\overline{ENN}$ pin is specifically designed to operate with the pipelined architecture of clocked FIFOs. After a Stop condition is detected, the state machine returns to the IDLE state and asserts the Waiting signal.

The state diagram includes test states for exercising the Built-In Self-Test (BIST) capabilities of HOTLink. The Built-In Self-Test loop is entered when the higher-level controller asserts the Test signal while the transmitter state machine is in the IDLE state. The BIST0 state asserts $\overline{BISTEN}$ to initiate to the transmission of the repeating character 1010101010. The BIST1 state then asserts $\overline{ENA}$ to start the BIST pattern generation. The higher-level controller could monitor $\overline{R}_P$ to count the number of BIST patters sent. Built-In Self-Test will conclude when the higher-level controller deasserts Test after the desired number of BIST patterns have been sent. Control then returns back to the IDLE state.

Critical Timing Analysis

Timing diagrams are provided for the transmitter interface. The analysis assumes that the state machine state state bits are accessible sooner than any data or input control signal.

FIFO-HOTLink Transmitter Data timing is governed by the FIFO access time (t_A= 10 ns) and the data set-up time for HOTLink (t_{SD} = 5 ns).

$$t_A + t_{SD} \leq t_{CKW} \qquad \text{Eq. 1}$$

With clock periods greater than 30 ns, the data has no trouble meeting these timing constraints.

The critical timing path of the FIFO-HOTLink Transmitter interface is due to the delay associated with decoding the flags and generating the enable for the clocked FIFO ($\overline{ENR}$) and HOTLink ($\overline{ENN}$). Note that these are the same signals, but $\overline{ENR}$ requires a longer set-up time than $\overline{ENN}$. The delay due to the state machine decoding the flags and generating the enable is represented as t_{PD}. The FIFO flag delay, t_{FD}, is 10 ns. The read enable set-up time for the FIFO, t_{SEN}, is 7 ns.

$$t_{PD} \leq t_{CKW} - t_{SEN} - t_{FD} \qquad \text{Eq. 2}$$

A 30-ns clock period leaves the controller 13 ns to generate the $\overline{ENN}$ signal. A timing diagram is provided in *Figure 3*.

Receiver Interface

The receiver interface uses a single CY7C451/3–14 clocked FIFO to buffer the parallel data presented by the HOTLink Receiver. The CY7C45X FIFO features programmable flags and three-state output drivers for bus applications. The HOTLink receiver interface is capable of receiving serial data at rates of up to 330 Mbits/second and then writing 9-bit words in the FIFO. Words in the FIFO can be read to the data bus at rates of up to 70 MBytes/s. A higher-level controller is responsible for coordinating the receiver interface and bus transactions according to the serial link protocol. *Figure 4* shows a block diagram of the receiver HOTLink-FIFO interface.

Reframe

The HOTLink serial receiver must synchronize itself with the proper word alignment of the incoming data. Assertion of the HOTLink RF (Reframe) input forces HOTLink to synchronize its internal bit counter with the boundary of a received K28.5 character. HOTLink will respond by asserting $\overline{RDY}$ LOW when the first K28.5 is received. The receiver state machine controller should be designed to synchronize HOTLink at the beginning of data reception or after excessive errors have been received.

Data Path and Controller

The receiver state machine responds to control signals from a higher-level controller. The higher-level controller initiates data reception by asserting the

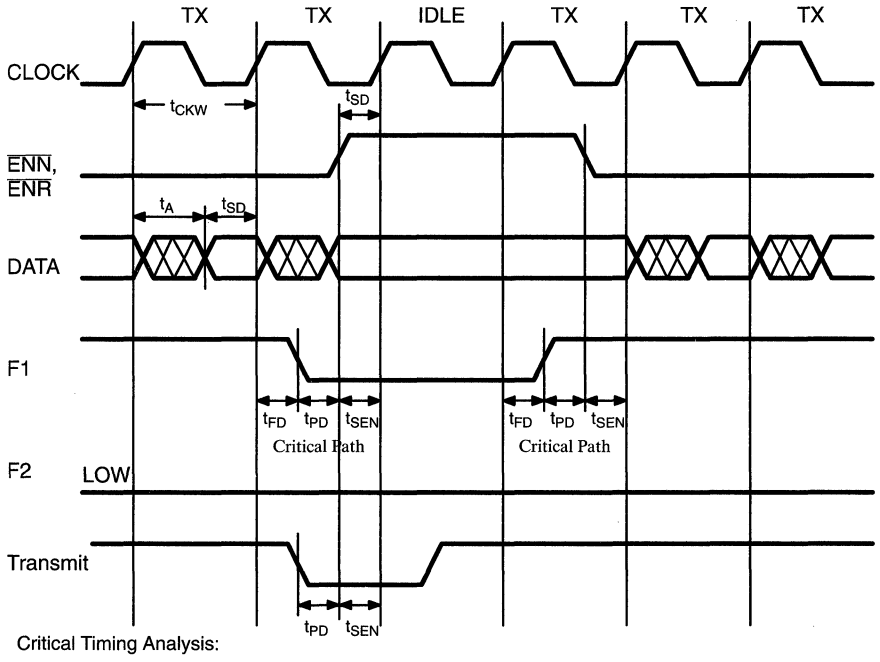


Figure 3. Transmitter Timing Diagram

Receive signal to the receiver state machine. Nine-bit words from the HOTLink parallel port are stored into the 7C45X FIFO each time $\overline{RDY}$ is asserted LOW. $\overline{RDY}$ will pulse LOW when new data is available at the HOTLink parallel port and will be HIGH when a pad sequence is received (multiple K28.5 SYNC codes). $\overline{RDY}$ is used to prevent the FIFO from filling with SYNC characters. Data storage will stop immediately when Receive is deasserted.

If the FIFO becomes full, it will ignore attempted writes. Full and Empty flags are decoded so that the higher-level controller can detect when the FIFO contains data or is completely full.

The 7C45X features programmable Almost Full and Almost Empty flags. The distance that these flags become active from the Empty and Full FIFO boundary is programmed during the FIFO Master Reset cycle. The distance can be set such that a flag is asserted when a fixed length packet of data has been received. The higher-level controller responds to the flag by reading the data packet out of the FIFO. The Almost Full flag is useful for preventing data from being lost. This flag can be programmed to compensate for the response latency of the higher-level controller so that data can be read from the FIFO before it becomes full. The decoding of the programmable flag signals is left out of the controller design for clarity.

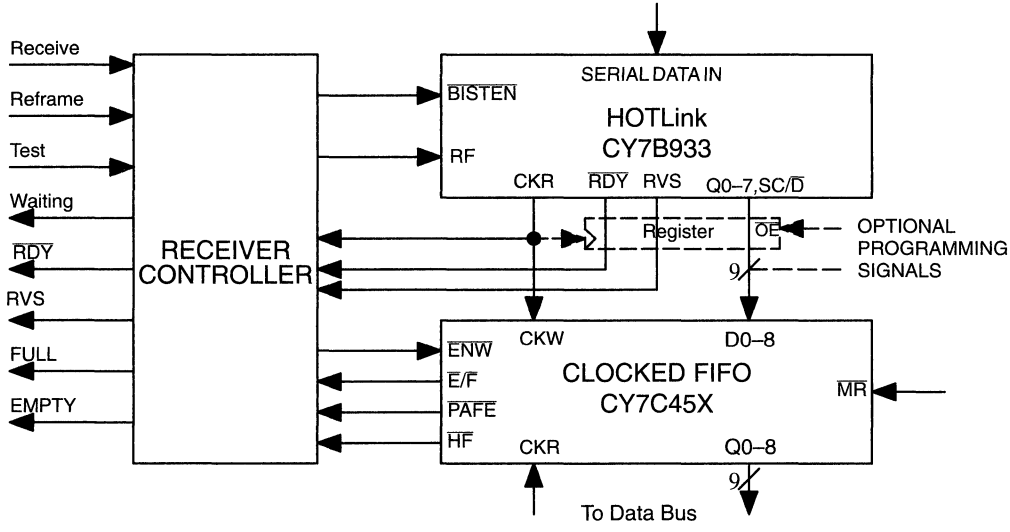


Figure 4. Receiver Interface Block Diagram

Optional Pipeline Register

The optional pipeline register increases interface speed by capturing the $\overline{RDY}$ pulse and easing the control signal timing margins. $\overline{RDY}$ is a delayed 60% LOW duty cycle signal shaped for asynchronous FIFOs. Without the pipeline register, the LOW phase of $\overline{RDY}$ leaves less than $\frac{1}{2}t_{CKR} - 10$ ns to generate the FIFO write enable and meet the set-up time. A clock period of 40 ns (250 Mbit/second) leaves a manageable 10 ns for the receiver state machine to generate the FIFO write enable, but as the clock period decreases to 30 ns (330 Mbit/second), the enable generation time shrinks to only 5 ns. This timing difficulty is overcome by pipelining the interface. The data and status signals must be pipelined to insure the proper word is written into the FIFO. The timing implications are considered in the section on critical timing analysis.

A data pipeline register with three-state output drivers can also be used to isolate the HOTLink Receiver parallel port from the FIFO write port while programming the CY7C45X FIFO flags. A 9-bit program word from an external source can be written into the FIFO during a Master Reset cycle. The program word sets the Almost Empty and Almost Full flags and sets the FIFO parity option.

Resetting and Programming the FIFO

The higher-level controller should perform a FIFO Master Reset cycle after power-up, before new data is received, if an error occurs, or in order to program the FIFO flags. A Master Reset cycle is accomplished by asserting the $\overline{MR}$ pin on the FIFO LOW. Proper resetting or programming requires that $\overline{MR}$ be glitch free. In addition, neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{MR}$ unless the FIFO is being programmed. If the FIFO is not being programmed, the receiver state machine should remain in the WAIT state during the Master Reset cycle.

In order to program the FIFO, the higher-level controller should put the data pipeline register in the high impedance state. The program word is then supplied to the FIFO by an external source (data bus, controller, etc.). This word is written into the FIFO internal program register during the Master Reset cycle on the rising edge of the clock that is enabled by $\overline{ENW}$ asserted LOW.

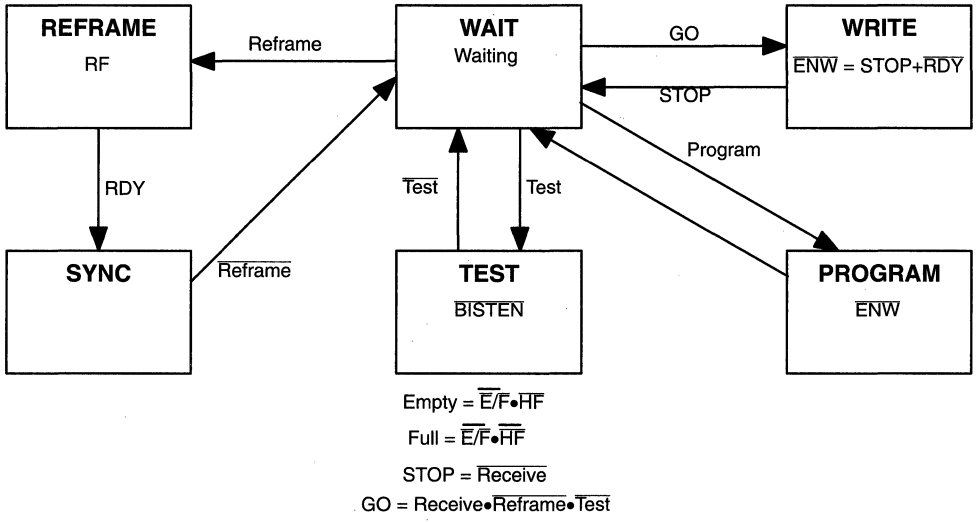


Figure 5. Receiver Controller State Diagram

Built-In Self-Test

The Built-In Self-Test mode is exercised by asserting the BISTEN pin on the HOTLink Receiver. Upon entering BIST, the HOTLink Receiver will wait for the BIST initialization code and then assert RDY LOW when the code has been received. RDY will pulse HIGH once per received BIST loop. RVS will pulse HIGH if a byte pattern mismatch occurs. RDY and RVS can be monitored by the higher-level controller to characterize the integrity of the link.

Controller State Description

A state diagram for a receiver state machine is shown in *Figure 5*. Five simple signals control the interface. The Receive signal instructs the state machine to store words into the FIFO when RDY pulses LOW. Deassertion of Receive ends data reception abruptly. The Reframe signal tells the state machine to synchronize the HOTLink Receiver to the serial data. The Test signal forces the HOTLink Receiver to enter BIST mode and the Program signal causes the state machine to write a word into the FIFO internal program register. The Waiting output signal is asserted when the state machine is in the WAIT state.

Full and Empty signals are decoded for the convenience of the higher-level controller to assist in reading data out of the FIFO. The programmable flags may also be decoded if they have been programmed. It is important that the flags be monitored because a full FIFO will ignore attempted writes. The higher-level controller is responsible for insuring that the FIFO does not become full.

The REFRAME state is entered by the assertion of Reframe from the WAIT state. The REFRAME state is used to synchronize the receiver to the incoming serial data stream. When the state machine asserts RF, the HOTLink Receiver synchronizes its internal bit counter with received K28.5 characters. RDY will pulse LOW when the first synchronized K28.5 character is available. The state machine will return to the WAIT state when the serial data has been resynchronized and Reframe is deasserted.

Data reception is initiated by asserting the Receive signal while the state machine is in the WAIT state. The controller will immediately transition to the WRITE state and store data when RDY is asserted

LOW. The WRITE state continually writes valid characters into the FIFO until Receive is deasserted. Control then returns to the WAIT state and Waiting is asserted.

The BIST state is included for handling the Built-In Self-Test. During BIST, writing to the FIFO is disabled. Assertion of RVS will signal a character reception error. $\overline{RDY}$ will pulse once per BIST loop and should be used to count the number of BIST loops received. The higher-level controller could monitor these signals in order to characterize the link.

The PROGRAM state writes the program word into the FIFO internal program register. This state is entered from the WAIT state at the command of the higher-level controller. Programming should only be performed during a Master Reset cycle ($\overline{MR}$ LOW). In order to meet the FIFO programming timing requirements, it is recommended that at least one clock cycle occur on each side of the program cycle while $\overline{MR}$ is LOW. The higher-level controller is responsible for meeting the specific programming timing requirements discussed in the Resetting and Programming the FIFO section of the CY7C45X datasheet.

Critical Timing Analysis

Timing analysis for both the pipelined and unpipelined interface are presented in this section. A Timing diagram is provided for the receiver interface that does not include the optional register. Critical timing relationships are provided at the bottom of *Figure 6*. This diagram highlights the critical timing of the $\overline{RDY}$ pulse. The interface timing with pipeline registers is straight forward and the results are presented below.

The timing analysis assumes that the state machine state bits are stable and valid before any critical signal is available to the state machine and that state bit set-up time is not an issue. This assumption allows the state machine timing to be modeled by its combinatorial t_{PD}.

Unregistered Timing

The delayed $\overline{RDY}$ pulse tightens the timing margins on the receiver controller. The state machine combinatorial delay for generating output control signals from valid inputs is modeled as t_{PD}. The FIFO enable set-up time is t_{SEN}=7 ns. Assuming t_{CKR} is 30 ns, the constraint on t_{PD} is

Write enable generation time from $\overline{RDY}$ LOW:

$$t_{PD} \leq 1/2\ t_{CKR} - t_{SEN} - 3\ ns = 5\ ns \qquad \text{Eq. 3}$$

A 40-ns clock period eases the timing constraint to a more reasonable 10 ns.

The parallel data have no problem meeting the timing constraints imposed by a 30-ns clock period. The HOTLink Receiver access time, t_A, is 9 ns and the FIFO data set-up time, t_{SD}, is 7 ns:

Critical data timing:

$$t_A + t_{SD} \leq t_{CKR} \qquad \text{Eq. 4}$$

This assumes no trace delays or clock skew.

Registered Timing

With the optional pipeline register inserted, the timing constraint on the controller is eased. A register access time, t_{AR}, of 10 ns and set-up time, t_{SU}, of 5 ns are assumed. Using a 30-ns clock, the HOTLink Receiver access time is $t_A = t_{CKR}/5 + 3$ ns = 9 ns.

The constraint on the combinatorial delay through the controller is

Write enable generation time from $\overline{RDY}$ LOW:

$$t_{PD} \leq t_{CKR} - t_{AR} - t_{SEN} = 13\ ns \qquad \text{Eq. 5}$$

The HOTLink data and $\overline{RDY}$ pulse timing constraints to the pipeline register are

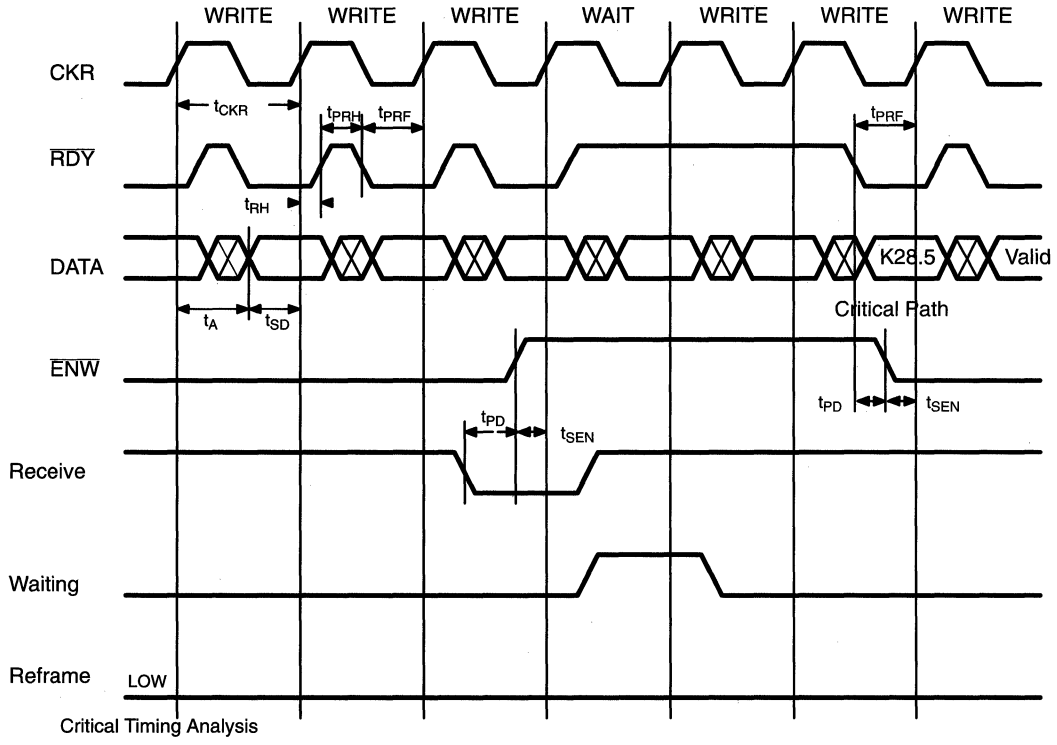
Data set-up time:

$$t_A + t_{SU} \leq t_{CKR} \qquad \text{Eq. 6}$$

$\overline{RDY}$ set-up time:

$$t_{SU} \leq 1/2 t_{CKR} - 3\ ns \qquad \text{Eq. 7}$$

These constraints are easily met.



Critical Timing Analysis

1. Data set-up time:
$$t_A + t_{SD} \leq t_{CKR}$$

2. Write enable set-up time from $\overline{RDY}$ going LOW:
$$t_{PD} + t_{SEN} \leq t_{PRF}$$

Figure 6. Receiver Timing Diagram

Conclusion

The HOTLink transmitter/receiver interfaces to clocked FIFOs can operate at speeds up to 330 Mbits/s with no external logic. Simple state machine controllers can be used to enable the transmission and reception of serial data and enable the HOTLink Built-In-Self-Test capability.

Interfacing the CY7B923 and CY7B933 (HOTLink™) to a Wide Data Clocked FIFO

This application note considers general interfacing issues between the Cypress CY7B923/CY7B933 (HOTLink™) Transmitter/Receiver and Cypress clocked FIFOs. The focus is on applications with a 36-bit data bus requiring high data transfer rates. A parallel FIFO solution is recommended for applications requiring large data bandwidth. Four FIFOs can achieve parallel data transfers on and off a 36-bit bus at rates of up to 280 Mbytes/s. The HOTLink serial link can transfer data at a serial rate of 330 Mbits/s. The FIFOs act as asynchronous storage buffers between the data bus and the serial link.

Transmitter Interface

This section describes the design considerations of a high-speed transmitter interface with FIFO (First In First Out) data buffers. The design implements basic data transmission and serial link testing capabilities. The transmitter is intended to interface to a higher-level controller responsible for coordinating bus transactions and handling the various protocol layers. The design considerations are easily extended to handle specific design requirements.

The transmitter interface consists of four Cypress CY7C441/3−14 clocked FIFOs buffering data between a 36-bit data bus and a Cypress HOTLink Transmitter. A 4:1 multiplexer (9 bits wide) funnels the wide FIFO data into the HOTLink parallel port. A local state machine controller coordinates the flow of data between the FIFOs and HOTLink. The FIFO−data bus interface and local controller architecture are left unspecified for generality. A block diagram of the FIFO-HOTLink interface is shown in *Figure 1*.

Data Multiplexers

The 4:1 multiplexers are part of the critical data path timing. These multiplexers can be implemented in several ways. Standard high-speed 153 dual 4:1 multiplexers can be used. Five of these devices are needed to accommodate 9-bit data. 74ACT153s with a maximum t_{SZ} of 11.5 ns and t_{DZ} of 9.5 ns are sufficient.

The 4:1 multiplexers can also be implemented with three Cypress 16L8−10s. Each 16L8 can accommodate three 4:1 multiplexers. This solution provides a smaller footprint and improves the critical timing margins. Critical timing margins are discussed in the Critical Timing Analysis section of this application note.

Built-In Self-Test

The transmitter interface is capable of checking the functionality of the serial link by exercising the Built-In Self-Test (BIST) mode of HOTLink. To initiate BIST, the $\overline{\text{BISTEN}}$ pin is held LOW, resulting in the transmission of the sequence ...1 0 1 0... . The $\overline{\text{ENN}}$ (Enable Next Parallel Data) pin is then pulled LOW to enable transmission of the BIST test pattern. HOTLink will assert the $\overline{\text{RP}}$ (Read Pulse) pin LOW at the beginning of BIST and will pulse it HIGH once per BIST loop. $\overline{\text{RP}}$ can be used to count the number of BIST loops sent. During BIST, HOTLink ignores data at its parallel port and the FIFOs do not perform any reads.

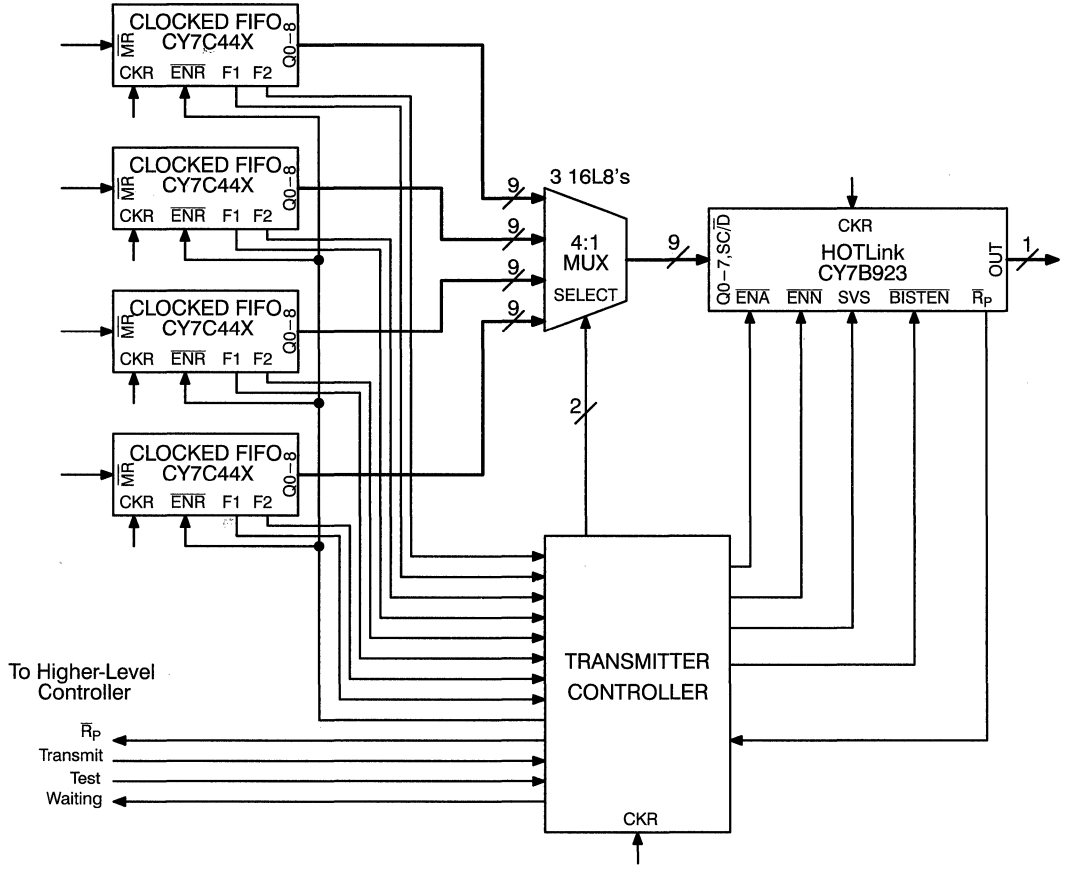


Figure 1. Transmitter Interface Block Diagram

Resetting the FIFOs

The higher-level controller should reset the FIFOs at power-up, before a new block of data is transmitted, or if an error is detected. Resetting or clearing the FIFOs is accomplished by pulsing the $\overline{MR}$ (Master Reset) pin on the FIFOs LOW. Neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{MR}$. $\overline{MR}$ must be glitch free. During the FIFO Master Reset cycle, the local transmitter controller should be in the WAIT state (see *Figure 2*). The higher-level controller is responsible for insuring that these conditions are met while performing the Master Reset cycle.

Transmitter Controller State Description

The local transmitter controller is responsible for reading data from the parallel FIFOs via the mux select lines and initiating the HOTLink BIST feature. The controller can be synthesized into a PLD or FPGA. Timing requirements of the controller are considered in the next section.

The local controller waits in the WAIT state while data is loaded into the FIFO. Meanwhile, HOT-

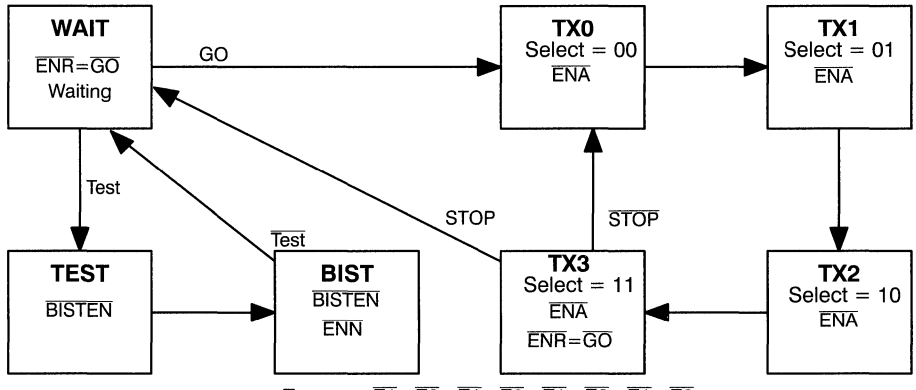


Figure 2. Transmitter Controller State Diagram

Link will transmit Idle special characters (K28.5). When the higher-level controller asserts the Transmit signal, the local transmitter controller issues a read ($\overline{ENR}$ LOW) to all the FIFOs and transitions to the TX0 state.

The transmit states (TX0−3) select data from the FIFOs in an ordered sequence. The TX0 state selects the byte out of FIFO0 for transmission and then transitions to the TX1 state. The TX1 state selects a byte out of FIFO1 and then transitions to the TX2 state. The TX3 state is responsible for checking the flags to determine if all of the FIFOs are empty, and then asserts $\overline{ENR}$ if they are not. (The controller can be designed to report an error if not all FIFOs are empty at the same time.) The transmit loop continues until all the FIFOs are empty or until Transmit is deasserted. Control then returns to the WAIT state. The Waiting signal should be monitored to determine when data transmission has ceased.

The state diagram of the local transmitter controller includes states for exercising the Built-In Self-Test capabilities of HOTLink. The local state machine enters the BIST state from the WAIT state when the higher-level controller asserts the Test signal. BIST is exited when Test is deasserted. The higher-level controller monitors $\overline{RP}$ for BIST loop counting. $\overline{RP}$ will pulse LOW one time per BIST loop. *Figure 2* illustrates the controller state diagram.

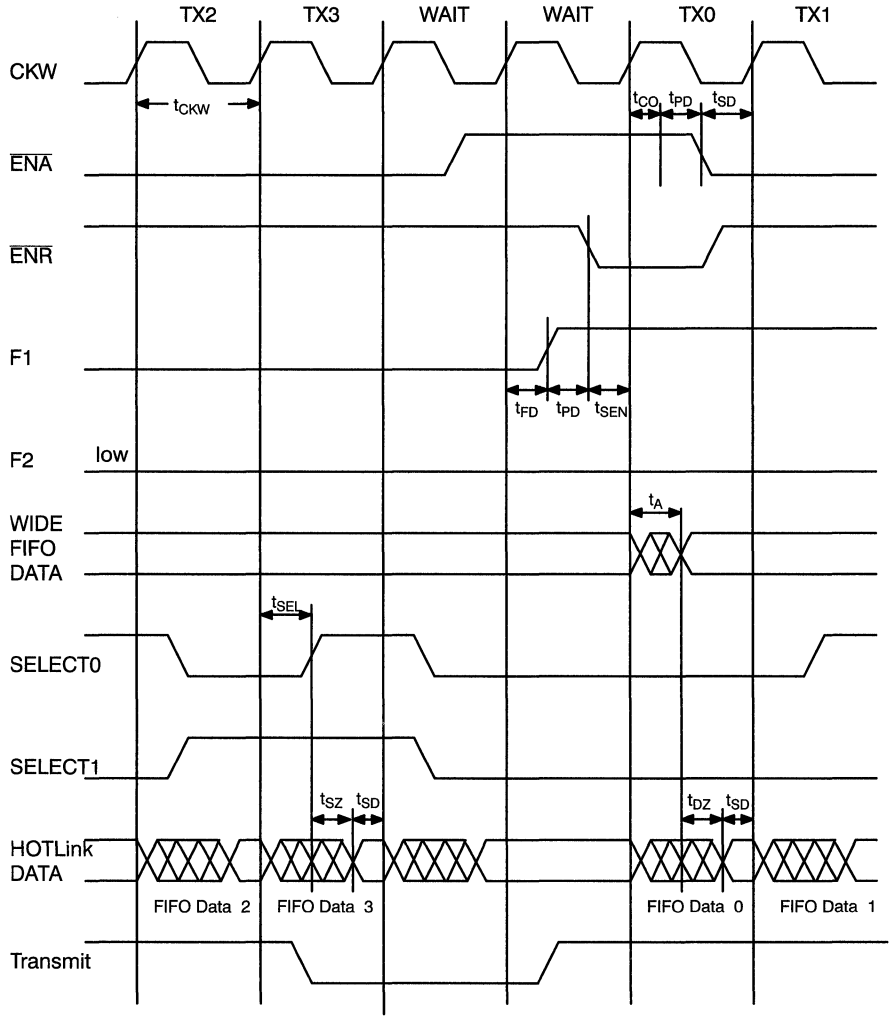
Critical Timing Analysis

The timing analysis in *Figure 3* highlights three critical data timing paths. The first critical path arises in the WAIT or TX3 states from the delay associated with decoding the flags and generating the read enable for the clocked FIFOs. The FIFO delay for generating the flags, t_{FD}, is 10 ns. The delay due to the controller decoding the flags and generating the enable is represented as t_{PD}. The read enable set-up time for FIFOs, t_{SEN}, is 7 ns ($t_{SEN} > t_{SD}$). The constraint imposed upon the controller is

$$t_{PD} \leq t_{CKW} - t_{FD} - t_{SEN}$$

With a-30 ns clock period, the signal propagation delay through the controller must be $t_{PD} \leq 13$ ns excluding trace delays and clock skew. This timing analysis assumes that the state register outputs are fed back to the controller before the flags signals are valid ($t_{CO} < t_{FD}$).

The second critical timing case assumes that data is available at the mux before the data selector signals



Critical Timing Analysis

1. Read enable set-up time:
$$t_{FD} + t_{PD} + t_{SEN} \leq t_{CKW}$$
2. HOTLink data set-up time from MUX data select:
$$t_{SEL} + t_{SZ} + t_{SD} \leq t_{CKW}$$
3. HOTLink data set-up time from FIFO data access:
$$t_{A} + t_{DZ} + t_{SD} \leq t_{CKW}$$

Figure 3. Transmitter Timing Diagram

($t_{SEL} > t_A$, where the delay from a clock edge to the arrival of the data selectors at the muxes is t_{SEL}). The delay from the selector pins to valid output data is t_{SZ}. The data set-up time to HOTLink, t_{SD}, is 5 ns. The critical timing associated with this path is

$$t_{SEL} + t_{SZ} + t_{SD} \leq t_{CKW}$$

The time to generate the data selectors from the controller is minimized by using the low-order bits of the state machine as the selectors and assigning TX0–3 to these states. This decreases the hardware required for the controller and reduces the selector signal-generation time to the clock-to-output time (t_{CO}) of the state registers. Assuming a 30-ns clock and t_{CO}=10 ns, the mux delay must be $t_{SZ} \leq 15$ ns.

The delay through the mux from valid input data to valid output is t_{DZ}. Assuming that the data selectors arrive before the data ($t_A > t_{SEL}$), the critical timing of this path is given by

$$t_A + t_{DZ} + t_{SD} \leq t_{CKW}$$

The data access time of the FIFOs, t_A, is 10 ns. With a 30-ns clock period, the constraint imposed upon the mux is $t_{DZ} \leq 15$ ns, assuming no trace delays or clock skew.

Receiver Interface

In this section a solution is presented for interfacing a HOTLink receiver to a 36-bit data bus. Control of the interface is simple and is easily adapted to system requirements. The four parallel CY7C451/3–14 FIFOs provide a high-speed interface to the data bus, allowing parallel transfer at rates up to 280 Mbytes/s. The serial link can receive data at serial rates up to 330 Mbits/s. The receiver interface is designed to provide proper word alignment in the FIFOs after synchronization to the data stream has been achieved. *Figure 4* shows a block diagram of the HOTLink-FIFO receiver interface.

Reframe

The receiver interface must synchronize itself to the incoming data and then store the data in the FIFOs with proper word alignment. The HOTLink RF (Reframe) input is used to synchronize the receiver to the transmitted data. Assertion of RF forces HOTLink to synchronize its internal bit counter with the boundary of a K28.5 character. HOTLink will respond by asserting $\overline{RDY}$ LOW when the first K28.5 is received. Reframing may be performed before data storage in order to synchronize HOTLink to the incoming serial data stream.

Idle Decoder

The Idle Decoder decodes the three types of idle characters: K28.5 (C5.0), –K28.5 (C1.7), +K28.5 (C2.7). These idle characters are used to signal the boundary of data words to be read into the FIFOs. A logic equation for the Idle Decoder is contained in *Figure 5*. A–H refer to HOTLink output pins Q0–Q7. When the Receivel signal is asserted by the higher-level controller to the local controller, reception of any of these idle characters will trigger received data to be continually stored in the FIFOs starting with FIFO0 (*Figure 5*). The combinatorial delay through the decoder is modeled as t_{ID}.

Data Path and Controller

The HOTLink receiver parallel port interfaces directly to the FIFOs' write ports. A pipeline register may be inserted to improve timing margins or allow the FIFOs to be programmed. A local receiver controller coordinates the data flow and enables the HOTLink receiver BIST feature. The local receiver controller interfaces to a higher-level controller that coordinates all of the protocol layers of the link and the data bus transactions.

The higher-level controller instructs the local controller when to start data reception. A K28.5 character delimits the start of a data transmission. When this character is detected by either HOTLink or the Idle Decoder, the local controller writes the incoming data into the 45X FIFOs. The writing process continues until the higher-level controller signals the local receiver controller to stop.

The FIFO flags are decoded to signal when the FIFOs are empty or are full. A full FIFO will ignore attempted writes. The 45X FIFO features programmable Almost Full and Almost Empty flags that can assist in signaling when the FIFO is becoming too full. Programmable flag signals are left out of the design for clarity.

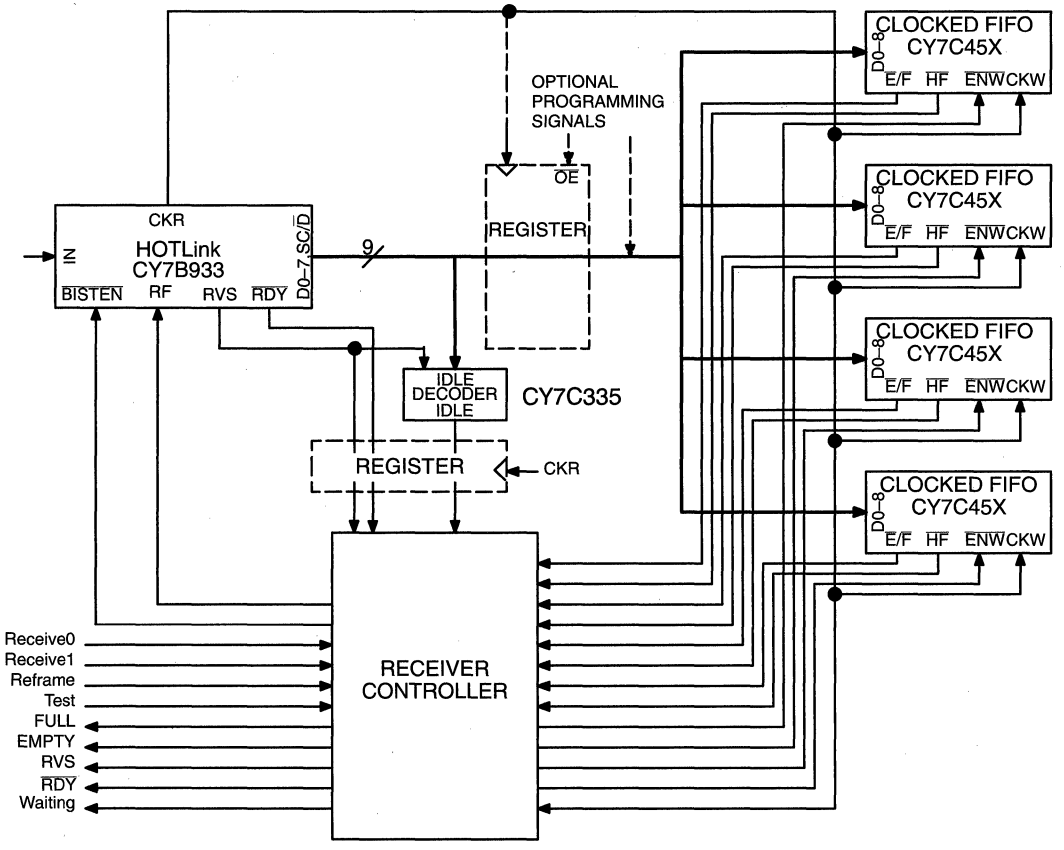


Figure 4. Receiver Interface Block Diagram

The Cypress 45X family of clocked FIFOs feature three-state data output drivers for direct interfacing to a data bus. The higher-level controller is responsible for reading words from the FIFOs' read port to the data bus.

The architecture of the local receiver controller is unspecified, but can be implemented with a PLD or FPGA. State machine descriptions and a timing analysis of the data path and local receiver controller are provided in the next sections.

Optional Pipeline Registers

The optional pipeline registers increase the interface speed by capturing the $\overline{RDY}$ pulse and easing timing constraints on the controller. $\overline{RDY}$ is a 60% LOW duty cycle signal shaped for interfacing to generic asynchronous FIFOs. The LOW phase of $\overline{RDY}$ leaves less than $\frac{1}{2}t_{CKR} - 10$ ns to generate the FIFO write enable and meet the FIFOs' set up time. A 40-ns clock period (250 Mbit/s) allows 10 ns for the local controller to generate a FIFO enable. This time shrinks to 5 ns when a clock period of 30 ns (330

Mbit/s) is used. The optional pipeline register captures the delayed $\overline{RDY}$ pulse and allows it to be processed earlier during the next clock cycle. The data and control signals must also be delayed by one clock cycle to ensure proper data alignment. A single CY7C335 PLD can be used to accommodate the data pipeline registers, the Idle Decoder, and the control signal delay registers. The timing implications of the registers are considered in the section on critical timing analysis.

The pipeline registers also isolate the HOTLink parallel port from the FIFO write ports while programming the FIFOs. A data pipeline register with three-state output drivers should be used so that data from an external source can be used to program the FIFOs. Additional states and control signals must be added to the controller. Programming is performed during the FIFO master reset cycle.

Built-In Self-Test

The Built-In Self-Test mode is exercised by asserting $\overline{BISTEN}$. Upon entering BIST, HOTLink will await the BIST initialization code and then assert $\overline{RDY}$ LOW when the code has been received. $\overline{RDY}$ will pulse HIGH once per received BIST loop. RVS will pulse HIGH if a byte pattern mismatch occurs. $\overline{RDY}$ and RVS can be monitored by the high-level controller to characterize the error rate.

Resetting and Programming the FIFOs

The higher-level controller should reset the FIFOs after power-up, before a new block of data is received, if an error occurs or in order to program the FIFOs. Resetting or programming the FIFOs is accomplished by pulsing the $\overline{MR}$ pin on the FIFOs LOW. Neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{MR}$ unless the FIFOs are being programmed. FIFO programming information is contained in the CY7C451/3 data sheet. $\overline{MR}$ must be glitch free. The receiver controller should only be in the WAIT or PROGRAM states during a master reset. The higher-level controller is responsible for insuring that these conditions are met.

Controller State Description

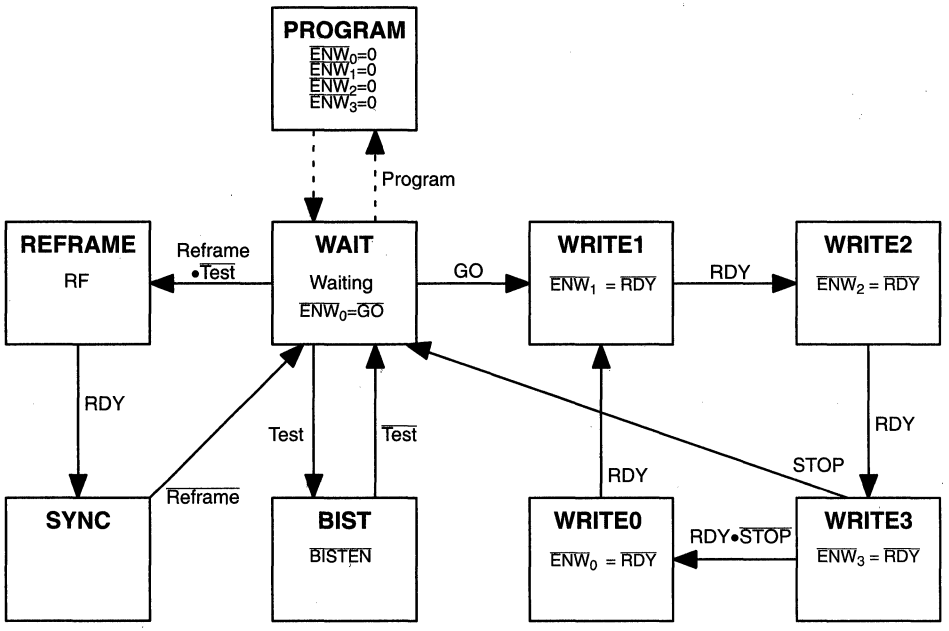
A state diagram for the receiver interface controller is shown in *Figure 5*. Five simple signals control the interface. The Receive0 and Receive1 signals are used to initiate and stop the reception of data. Reframe is used to synchronize the receiver to the serial data stream. Test causes HOTLink to perform BIST. Waiting is an output signal that indicates that the receiver is in the WAIT state.

Full and Empty signals are decoded for use by the higher-level controller to assist in managing data out of the FIFOs. The programmable flags may also be decoded but are not shown. A full FIFO ignores attempted writes resulting in lost data. Monitoring the state of the FIFOs is the responsibility of the higher-level controller. Resetting the FIFOs by pulsing $\overline{MR}$ LOW is also the responsibility of the higher-level controller.

The REFRAME state is used to synchronize the receiver to the incoming serial data stream. The REFRAME state asserts RF to the HOTLink receiver, signaling it to synchronize its internal bit counter with the first-received K28.5 character. $\overline{RDY}$ will pulse LOW when a synchronized K28.5 character is available. The controller will transition back to the WAIT state when synchronization is achieved and the Reframe signal is deasserted.

Receive0 and Receive1 initiate the storing of data in the FIFOs from the WAIT state. The assertion of Receive0 causes the controller to look for the assertion of $\overline{RDY}$ in order to begin data storage. The assertion of Receive1 causes the controller to look for the assertion of IDLE in order to begin data storage. The received K28.5 is written into FIFO0 and then the write loop is entered. The choice of which receive mode to use depends on the serial link protocol.

The write loop continually writes valid characters into the FIFOs. $\overline{ENW0}$–3 are cycled in order as the data is received. The fullness of the FIFOs is ignored by the controller. The higher-level controller monitors the Full flag signal and takes corrective action if the FIFOs become too full. The deassertion of both receive signals will end the writing process and return control back to the WAIT state on the



IDLE = $\overline{H}\bullet\overline{G}\bullet F\bullet\overline{E}\bullet\overline{D}\bullet C\bullet\overline{B}\bullet A\bullet SC/\overline{D}$ + $H\bullet G\bullet F\bullet\overline{E}\bullet\overline{D}\bullet\overline{C}\bullet\overline{B}\bullet A\bullet SC/\overline{D}$ + $H\bullet G\bullet F\bullet\overline{E}\bullet\overline{D}\bullet\overline{C}\bullet B\bullet\overline{A}\bullet SC/\overline{D}$

GO = (Receive0•RDY + Receive1•IDLE)•$\overline{\text{Reframe}}\bullet\overline{\text{Test}}$

STOP = $\overline{\text{Receive0} + \text{Receive1}}$

Empty = $\overline{E/F_0}\bullet\overline{HF_0}\bullet\overline{E/F_1}\bullet\overline{HF_1}\bullet\overline{E/F_2}\bullet\overline{HF_2}\bullet\overline{E/F_3}\bullet\overline{HF_3}$

Full = $\overline{E/F_0}\bullet\overline{HF_0}\bullet\overline{E/F_1}\bullet\overline{HF_1}\bullet\overline{E/F_2}\bullet\overline{HF_2}\bullet\overline{E/F_3}\bullet\overline{HF_3}$

Figure 5. Receiver Controller State Diagram

next word boundary. The higher-level controller should monitor the Waiting signal to determine when receiver controller has returned to the WAIT state.

The BIST state is included for handling the Built-In Self-Test. During BIST, writing to the FIFOs is disabled. HOTLink signals are passed on to the higher-level controller for error analysis. RVS will signal character reception errors. $\overline{RDY}$ will pulse HIGH once per BIST loop and should be used to count the number of completed BIST loops.

A single PROGRAM state that writes an external program word to all of the FIFOs in parallel can be added to the state machine. This state is entered and exited during a FIFO master reset cycle. The higher-level controller should assert $\overline{MR}$ LOW, put the data pipeline register in the high-impedance state, and then drive the external program word to the FIFO write ports. The higher-level controller then puts the local controller in the PROGRAM state. The program word is written into the FIFOs' internal program registers when the local controller exits the PROGRAM state.

Critical Timing Analysis

A critical timing analysis of both the pipelined and unpipelined receiver interfaces is presented in this section. A timing diagram with critical timing equa-

tions is provided in *Figure 6* for the receiver interface that does not include the optional pipeline registers. Timing for the pipelined case is very similar. The analysis assumes that the state register bits are valid before any critical signals are available to the controller.

The critical timing path constrains the propagation delays associated with the local receiver controller and Idle Decoder. The combinatorial timing delay through the controller is modeled as t_{PD}. The combinatorial delay through the Idle Decoder is modeled as t_{ID}.

Unpipelined Timing

The timing for the unpipelined configuration is as follows. Assuming t_{CKR}=30 ns and t_{SEN}=7 ns, the propagation delays are

Write enable generation time from $\overline{RDY}$ LOW:

$$t_{PD} \leq 1/2\ t_{CKR} - t_{SEN} - 3ns = 5\ ns$$

IDLE generation time from data:

$$t_{ID} \leq 4/5\ t_{CKR} - t_{SEN} - t_{PD} - 3\ ns = 9\ ns$$

These constraints require (approximately) $t_{PD} \leq 5$ ns and $t_{ID} \leq 9$ ns. With a 40 ns clock cycle, these timing constraints are relaxed to $t_{PD} \leq 10$ ns and $t_{ID} \leq 12$ ns.

Pipelined Timing

With the optional pipeline registers inserted the timing margins of the control logic are eased. Assuming the register access time is t_{AR}=10 ns and the register set up time is t_{SU}=5 ns

Write enable generation time from clock:

$$t_{PD} \leq t_{CKR} - t_{SEN} - t_{AR} = 13\ ns$$

IDLE generation time from data:

$$t_{ID} \leq 4/5\ t_{CKR} - t_{SU} - 3\ ns = 14ns$$

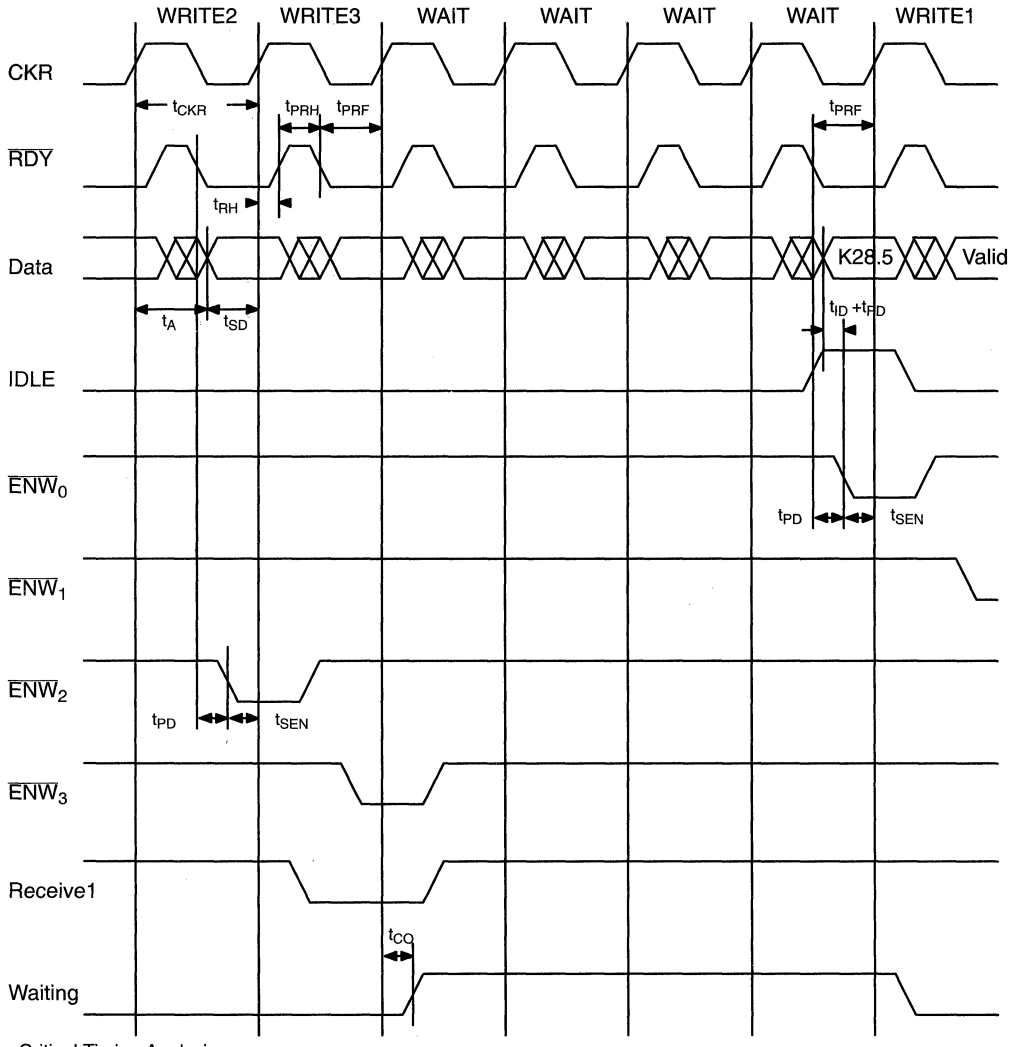
$\overline{RDY}$ capture timing:

$$t_{SU} \leq 1/2 t_{CKR} - 3ns = 12\ ns$$

The pipeline registers ease the receiver control logic timing margins to (approximately) 13 ns. The entire pipeline circuitry, including the Idle Decoder, can be synthesized into a single CY7C335–83 PLD while meeting these timing constraints.

Conclusion

The HOTLink Transmitter/Receiver interfaces to wide data FIFOs can operate at speeds of up to 330 Mbits/s with minimal interface logic. State machine controllers ensure proper word alignment during data transfers over the HOTLink serial link and provide Built-In Self-Test capability. Critical timing equations are provided. The interface designs are easily modified to meet specific demands.



Critical Timing Analysis:

1. Data set-up time
   $t_A + t_{SD} \leq t_{CKR}$
2. Write enable set-up time from $\overline{RDY}$ LOW
   $t_{PD} + t_{SEN} \leq t_{PRF}$
3. Write enable set-up time from idle HIGH:
   $t_A + t_{ID} + t_{PD} + t_{SEN} \leq t_{CKR}$

Figure 6. Receiver Timing Diagram

HOTLink is a trademark of Cypress Semiconductor Corporation.

Frequently Asked Questions about HOTLink™ Evaluation Boards

The following questions are frequently asked by customers who are using HOTLink™ Evaluation Boards. These cursory answers will serve as an introduction for each topic. Separate application notes cover these topics in more complete detail.

1. **How can I convert a CY9266−C (75Ω) Evaluation Board to use 50Ω cables? How can I convert a CY9266−C (75Ω) board to use 93Ω coax? How can I convert a CY9266−T (150Ω) STP (shielded twisted-pair) board to use 100Ω STP cables?**

   Conversions of the CY9266−C and CY9266−T boards to use transmission lines other than those shipped in the standard configurations is as simple as changing the transmission line termination resistors (R40 and R41) on the back side of the board. Carefully remove the ones currently on the board (presently 37.4Ω on a −C) and replace them with resistors with a value equal to half the transmission line characteristic impedance (i.e., 25Ω for a 50Ω cable). See *Table 1* for the values used for some common cable impedances. Extreme care must be used to avoid delamination of the board and damage to the traces by excessive heat during desoldering and resoldering.

   The change from higher to lower impedance transmission lines (e.g., 75Ω to 50Ω coax or 150Ω to 100Ω STP) may also require that the user change the transformer at T1. Changes from lower to higher impedance transmission lines usually do not require transformer changes. Alternatively, it may be desirable to add resistors at R54 and R55. (If these resistors are added, cut the built-in wire-traces that currently short the previously unused solder pads.) The higher currents involved in driving lower impedance transmission lines require either a higher inductance transformer or series current limiting resistors.

   As the impedance of the external cable changes, the drive level must vary to compensate. Part of the drive circuit, R61 & R62, needs to change to in order to vary the drive current available. See *Table 1* for the values required for various cable impedances. Changes in drive current will change the spectral characteristics of the souce signal and therefore the usable distance with a specific media type.

Table 1. Cable Impedance vs. R Values

Cable Impedance	R40 & R41	R61 & R62
150Ω	75Ω	392Ω
100Ω	50Ω	261Ω
93Ω	46.4Ω	243Ω
75Ω	37.4Ω	196Ω
50Ω	24.9Ω	130Ω

2. How can I convert a CY9266–C (75Ω) Evaluation Board to use 150Ω STP cables (like CY9266–T)? How can I convert a CY9266–T (150Ω) STP board to use 75Ω cables (like CY9266–C)?

Conversion of the CY9266–C and CY9266–T boards to use transmission lines other than those shipped in the standard configurations is as simple as changing the transmission line connectors and the transmission line termination resistors (see the answer to question 1).

For the CY9266–C: Carefully desolder and remove the BNC and TNC connectors installed at at J1 and J2. Replace them with the connector of choice using the mounting and solder terminal holes provided. WARNING: the CY9266–C board grounds the shield of the coax, and therefore one side of the transformer secondaries. Cut the traces leading to J1 and J2 on the solder side of the board (Under P1) to convert to balanced operation.

For the CY9266–T: Carefully desolder and remove the Sub-D installed at at P1. Replace it with the connector of choice using the mounting and solder terminal holes provided. The three traces running on the solder side from P1 to J1 and J2 were cut to unground the cable and allow balanced operation. Reconnect these wires for unbalanced cable connections.

Changing connectors often also involves changing the impedance of the cable used. See question 1 above about changing the resistor values for different values of cable impedance.

3. What types of Optical Modules are compatible with the CY9266–FX Evaluation Board?

We have tested and are shipping the CY9266–F Evaluation Board with Siemens, HP, and AT&T Optical Modules.

Table 2. Vendors for Optical Modules

Vendor	Part Number	Markings
CTS (formerly AT&T)	1408N	1408N ODL XCVR
HP	HFBR-5302	HFBR–5302
Siemens	V23806-A7-C2	Optical Data Link FC266 Transceiver
HP (formerly BT&D)	DLT1040-ST-2 DLR1040-ST-2	Separate TX & RX modules uses ST Fiber cabling
AMP/Lytel	269063–1	AMP SC Duplex Transceiver 270 Mb/s 269063-1

These modules may be purchased from the following vendors. Although this is not a complete list of Optical Module vendors, it will serve as a starting point for finding a module that may suit your needs:

AMP/Lytel Division
61 Chubb Way
P.O. Box 1300
Somerville, NJ 08876
(908) 685-2000

CTS Corp
1201 Cumberland Ave
West Lafayette, IN 47906–1388
(317) 463-2565

Hewlett-Packard
Components Division
370 West Trimble Road
San Jose, CA 95131
(800) 535-7449 or (408) 435-6342

Siemens Fiber Optic Components
20F Commerce Way
Totowa, NJ 07512
(201) 890-1606

Sumitomo Electric
Fiber Optics Corporation
777 Old Sawmill River Road
Tarrytown, NY 10591-6725
(914) 347-3770

4. Is this board compatible with (i.e., how do I use it with ...) the IBM/HP OLC card?

The HOTLink Evaluation Board is intended to allow easy evaluation of Cypress HOTLink parts and is not intended to replace the IBM® OLC card as a system interface (although it is capable of performing

this function). The OLC compatibility offered with these boards allows a familiar interface for those systems already compatible with the IBM cards.

OLC system interface signals in JP4 have the same timing and logical levels as the OLC card. Drive and loading are similar, but not identical. The function of the CY9266 Byte-Sync output differs from that of the OLC card when Sync-Enable is LOW. The OLC card will hold Byte-Sync LOW if Sync-Enable is LOW, while the CY9266 will set Byte-Sync HIGH for each byte containing a K28.5. When Sync-Enable is HIGH both boards will behave as the CY9266 does. The CY9266 behavior is convenient for implementing a simple "out of lock" indicator using timers that detect the interval between K28.5s (when Sync-Enable is LOW, a misframed K28.5 does not cause a Byte-Sync indication).

The CY9266 serial interface is incompatible with the IBM OLC card serial interface. The IBM OLC interface uses an 850-nm short wave laser and detector. The HOTLink Evaluation board uses off-the-shelf 1300-nm LED transmitters and detectors or copper transmission line interfaces. These various types are not compatible. For an operational link, use two compatible serial interfaces (i.e., two CY9266 boards of the same type, either −C, −T, or −F) for the two ends of the transmission link.

Note: The active signal level of the LOOPBACK signal, as implemented on the CY9266, is opposite that of an actual OLC-266 card. If this signal is under software control, it should be programmed to allow signal loopback when the signal is active LOW. For hardware controlled systems an external signal inversion is necessary, or the signal may be jumpered at JP1 for operation from the S1-7 DIP switch.

The physical size of the HOTLink Evaluation Board was chosen to be compatible with the two-channel version of the IBM OLC card. The X−Y dimensions are identical to those of the IBM product, but the thickness and the protrusion of the serial interface hardware is different from the IBM product.

The IBM OLC card includes plastic card guides and attachment clips that facilitate its use in production systems. The HOTLink Evaluation Board has none of these components since it is not intended for the same function.

5. Where can I get additional fiber-optic cables and accessories? Where can I get additional coaxial cables or STP cables?

We have located the following vendors of fiber-optic cables and accessories. You may contact them to receive further information about their offerings. The lists below represent only some of the available sources.

Fiber Instrument Sales Inc. 315-736-2206 315-736-2285 FAX	Nu-Power Optics 619-471-7131	FIBERTRON Tel: 714-871-3344 Fax: 714-871-5616	Belden Wire and Cable 800-BELDEN-1 order 317-983-5200

Additional coaxial and STP cables and other accessories may be found through:

Pasternack Enterprises 714-261-1920	First Source 408-371-1470	Newark 312-784-5100	Digi-Key Tel: 800-DIGI-KEY

6. How do I use this board to do bit-error-rate (BER) tests?

- Connect the board(s) with a suitable length of transmission line or fiber from the TX port of one board to the RX Port on another (or itself).
- Place the receiving board's Receiver in BIST mode by setting the RCV_BISTEN signal LOW. Ground the external pin marked RCV_BISTEN or set switch S1-5 to ON.

- Place the transmitting board's Transmitter in BIST Transmit mode by setting the XMIT_BISTEN signal LOW. Ground the external pin marked XMIT_BISTEN or set switch S1-1 to ON.
- Press the white reset button on the receiving board. The display should initially show a **.0.** . As the receiver finds an error in the data stream, it will show this with an increasing count. As the count exceeds 100, the overflow indicator will light up.
- The BER may be approximated by: 1 error/hour $\approx$ a BER of 1.1×10^{-12} using the 25.0-MHz oscillator shipped with the board.

7. How do I use this board to do transmitter jitter tests?

To achieve the best possible and most accurate transmit jitter measurements, the external environment of the HOTLink chips needs to have the lowest possible jitter to start. Common oscilloscopes and sources have so much jitter as to obscure the contribution of the transmitter. Additional sources of jitter on this board include:

- For the −C and −T versions: the transformer's frequency characteristics. For the −F version: the optical module.
- Layout of these boards has not been optimized for this testing, and does not have specific test connections built in.

With these items understood, a set-up to do an adequate test requires a quiet clock source and a digital oscilloscope such as the Tek 11801 or the HP 54720. The −F version without an optical module has the most convenient connections. Making connections to the −F board at location U4, all differential PECL signals, will allow the best measurements possible. (See the "HOTLink Jitter Characteristics" application note for information on how to measure jitter.)

Note: Transmit Jitter measured out of a −C or −T board includes significant crosstalk from the receive channel, coupled through the transformer. Ideally, measure Transmit Jitter with a quiet receive channel.

8. How do I use this board to do receiver jitter tolerance tests?

The ultimate performance of any serial link is determined by the performance of the receiver. The function of the receiver is to recover data from a (seemingly arbitrary) serial data stream. This data stream is translated several times, coupled to and though several non-linear devices and subjected to all manner of distortion. The receiver must accept this serial pulse train and recover a high-speed bit-synchronous clock, de-jitter it, and then separate the DATA from the CLOCK. Jitter tolerance is the typical term for the ability of the receiver to correctly recover the DATA and CLOCK in the presence of these many distortions. HOTLink Receiver jitter tolerance can be measured by connecting a suitable transmission media between the transmitter and receiver, and inserting a jitter generation source similar to that shown in the "HOTLink Jitter Characteristics" application note. By inserting measured jitter amplitudes and watching the RVS output of the receiver, jitter tolerance can be measured. Further details on the fabrication of the jitter generator and the measurement techniques required for accurate measurement of this injected jitter is beyond the scope of this note, but are covered in detail in the "HOTLink Jitter Characteristics" and "HOTLink Built-In Self-Test (BIST)" application notes.

9. How do I use this board to do HOTLink power supply noise immunity tests?

The layout and design of this board makes it difficult to test the power supply immunity of these parts. Power supply noise immunity testing requires injecting a signal into the power supply pins and observing the effect of this injected signal on the link. This requires a different layout to allow access to the power supply pins of the HOTLink chips without affecting the operation of the other parts on the board.

10. How do I use this board to do transmission-line tests?

To check for the maximum transmission-line length over which the HOTLink Evaluation Board can communicate, it is only necessary to connect the selected transmission line between the TX and RX ports of the HOTLink Evaluation Board. Using one board with the cable returning to its own RX port or two boards and cables for simultaneous testing in both/either directions of the transmission line will work quite well. The HOTLink Transmitter and Receiver BIST function serves the purpose of generating and testing the data so the user can check for an acceptable error rate without extra test equipment. Transmission lines can be extended or modified until the BIST error count indicates an unacceptable error rate. An error rate of approximately 1 error/hour $\approx$ a BER of 1.1×10^{-12} using the 25.0-MHz oscillator shipped with the board.

11. How do I use this board to do receiver-PLL acquisition-time tests?

Two kinds of receiver acquisition are measurable using this board. One kind shows how fast the receiver can recover from a phase hop, and the other shows how fast the receiver can acquire a datastream once the device is powered up with a stable REFCLK.

To measure the receiver recovery from a phase hop, connect a loopback cable with a delay just large enough to delay the data by almost one half a bit time ($\approx$2 ns for the shipped oscillator) with respect to the OUTC+ line that goes between the CY7B923 and the CY7B933. Then arrange a delayed synchronous switch signal into the A/$\overline{\text{B}}$ Select input of the receiver. Trigger this delay from $\overline{\text{RP}}$ and delay this pulse to a point in the data stream where the data stays HIGH for several bit times. By switching between the delayed and fast signal path, a phase hop can be created at the input to the receiver. Increase the delay until the receiver shows an RVS pulse during BIST testing. The receiver will properly recover data with a phase hop as large as $\pm 170°$. Invert the A/$\overline{\text{B}}$ select signal to get the other polarity of phase hop.

To observe the receiver recovery from a "lost" data stream, arrange the evaluation board to have an external REFCLOCK 0.1% faster or slower than the on-board oscillator. Configure the transmitter to only send K28.5s by either deasserting both the $\overline{\text{ENN}}$ and $\overline{\text{ENA}}$ signals, or constantly transmitting a C5.0 character in Encoded mode. With a clean pulse, switch the A/$\overline{\text{B}}$ select line to the B input. This will cause the receiver to see a lost and then found data stream. Using a delayed trigger, watch the CKR output with respect to the transmit clock. The two clocks will match frequency and stabilize in phase difference in less than 60 µs.

12. How do I use this board to do min/max frequency tests?

- Arrange the jumpers on the board so that the CKW and REFCLK use the same external clock input. Do this by removing the jumpers across pins IX−IY and GY−HY, then jumpering pins GX−GY and HX−IX. Apply an external reference clock to the XMITCLOCK pin on any of the interface connectors. Loopback the board either externally or by closing S1-7, which loops the board back on itself.
- Now enable the both the XMIT and RCVR BIST functions and the transmitter. The LED display should now show a stable number. Clear the count by pressing the RESET button S2.
- With the board set up as above, vary the frequency of the external reference clock from a nominal 20 MHz downward. As you approach the limits of operation, the board will start to indicate errors on the display. Clear the errors after setting a new frequency by pressing S2 again. The point in frequency where you do not see any BIST errors marks the edge of the frequency range. Change your frequency source upward toward 33 MHz and again clear the error indications until you achieve stable operation just below the high frequency limit.

Typical boards will operate as high as 40 MHz and as low as 12.5 MHz.

CY9266 HOTLink™ Evaluation Board User's Guide

Overview

This document describes the construction, interfaces, and operation of the CY9266–F (optical fiber), CY9266–T (shielded twisted pair/twinax), and CY9266–C (coaxial cable) HOTLink™ Evaluation Boards. These boards implement a complete bi-directional parallel-to-serial and serial-to-parallel communications link, capable of operation at serial rates of 160 to 330 Mbits/second (16 to 33 Mbytes/second). The supported rate of communication may be limited by the specific type and speed-grade of optical module or copper cable type used.

The CY9266 Evaluation Boards are optically, electrically, and mechanically compatible with the ANSI X3T11 Fibre Channel Interface, as documented in the ANSI standard ANS X3.230–1994. It provides three different methods of access for the TTL parallel interface and supervisor functions, for testing or exercising the serial data link.

Block Diagram

The block diagram in *Figure 1* illustrates the major functional blocks contained in the CY9266. These include:

- 10-bit TTL parallel transmit data input
- 10-bit TTL parallel receive data output
- Selectable Encoded or Bypass operation modes
- On-board oscillator
- Selectable internal/external clocking
- Selectable carrier-detect polarity
- Selectable local loopback
- Power supply voltage monitor
- Built-in self-test (BIST) pattern generation and checking hardware with error/status display

Board Connectors

This board offers three primary methods of TTL-level access:

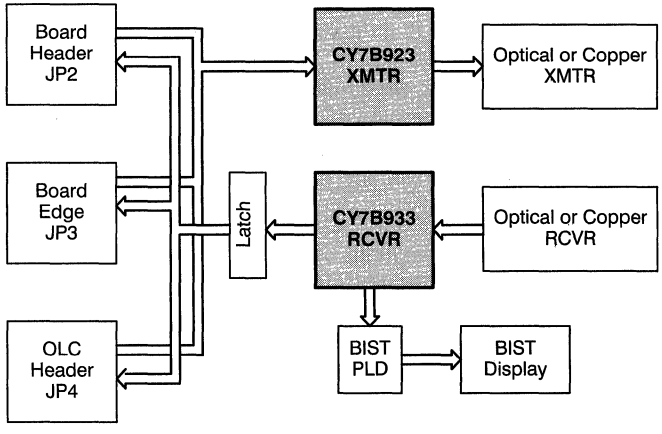


Figure 1. HOTLink Evaluation Board Block Diagram

- JP2—A 58-position (2 x 29) set of holes, capable of accepting a 0.025″ sq. pin-header on the top or bottom of the board
- JP3—A 60-position (2 x 30) 0.1″ spaced board-edge finger stock
- JP4—A 48-position (4 x 12 matrix) 0.025″ sq. pin-header mounted on the bottom of the board

Connectors JP2 and JP3 provided access to all data input and output buses as well as all BIST, control, and clocking signals for the HOTLink Transmitter and Receiver. These connectors may be used individually or together since all signals present on JP2 are also present on JP3. Power for the board is also brought in through these same connectors.

Connector JP4 is positioned and pinned to match up with the connector and signals present on other industry standard Fibre Channel modules. Unlike these other modules (which may contain two full-duplex channels), this evaluation board only provides a single full-duplex channel. While sufficient room exists to build a board with two channels, other functionality was added (on-board oscillator, BIST PLD and display, etc.) in this space to allow better testing and demonstration of the enhanced capabilities present in the Cypress HOTLink parts.

An additional jumper block (JP1) is used to configure three of the operating characteristics of the board: clock sourcing, serial output enable (FOTO), and local loopback control.

Optical Modules

The CY9266–F Evaluation Board is designed to operate with industry-standard footprint optical modules. The evaluation board contains low-profile socket pins so the user may select and test optical modules from different vendors. This board accepts both the four-row DIP and the single-row endfire types of modules.

These modules are available from multiple vendors with either ST- or SC-type optical fiber connectors. Because these modules are all LED-based, they are not required to meet many of the safety standards (ANSI Z136.1 and Z136.2, F.D.A. regulation 21 CFR subchapter J, and IEC 825) necessary for laser-based modules. These modules should be used with 62.5/125-μm multimode graded-index fiber.

Coaxial Cables

The CY9266–C Evaluation Board is configured to support 75Ω coaxial cables that attach through BNC/TNC connectors. Other cable impedances may be used with the board by changing the value of the termination and driver bias resistors on the board.

Shielded-Pair Cables

The CY9266–T Evaluation Board is configured to support 150Ω shielded twisted-pair or twinaxial cable that attaches through a 9-pin D-sub connector. Other cable impedances may be used with the board by changing the value of the termination and driver bias resistors on the board.

BIST Support

The CY9266 contains an on-board control PLD and a two-digit error-count display that are used in conjunction with the BIST (built-in self-test) capability of the Cypress Semiconductor HOTLink Transmitter and Receiver. This capability allows the parts, and any serial link, to be exercised and monitored at their full data rate without the use of expensive external test equipment.

The BIST PLD (CY7C344) contains a simple state machine that monitors the HOTLink Receiver BIST state, and an error-counter that drives an external display. The complete contents of this PLD are documented in Appendix C.

This BIST PLD also drives the four decimal point LEDs on the displays. These indicators are used to present additional status information about the state of the board, the BIST state machine, and the serial link.

Design Criteria

The CY9266 Evaluation Board was designed as a low-cost demonstration vehicle for the Cypress Semiconductor HOTLink family of data communications parts. The goals of this board are to:

- Present a Fibre Channel interface board that is fully compliant with the mechanical, electrical,

optical, coding, and protocol specifications in levels 0 and 1 of the ANSI Fibre Channel standard

- Allow full data rate testing of the serial link without expensive test equipment
- Allow the user to exercise all modes of operation of the receiver and transmitter
- Offer various parallel attachment methods for simplified system interfacing
- Offer various media types for evaluation
- Allow simple interfacing to existing OLC-compatible test platforms

Because of the flexibility inherent in the HOTLink parts, these goals were easily achieved.

Three electrical connection methods are provided: a 60-pin board-edge connector, a 58-pin (2 x 29) 0.025″ square pin-header, and a 48-pin (4 x 12) 0.025″ square pin-header. These different connectors allow the user to select the connector form that best suits their desired mode of attachment.

The HOTLink Transmitter and Receiver contain a BIST capability. This capability was designed into the HOTLink parts to allow high-speed serial testing without expensive test equipment. All hardware necessary to exercise and monitor the BIST function is present on the CY9266 board. This hardware allows a bit-error-rate (BER) test to be performed without additional equipment.

The BIST capability of the HOTLink Transmitter and Receiver allows offline testing of the transmitter, receiver, and serial link, by performing a byte-by-byte comparison of the data while a 511-byte pseudo-random byte stream is repeatedly sent, received, and checked.

Through use of either JP2 or JP3, users may exercise all modes of operation of the parts. JP4 is configured as a functional system interface, and thus does not include all the mode, clock, and special control signals present on JP2 and JP3, all of which may be selected or controlled in JP1 or S1.

Connector Pin Numbering

JP2—58-Position Pin-Header

The 58-position pin-header (JP2) holes are located next to the board-edge connector. Pin 1 of this connector area is identified on the board by a square solder pad. The remaining pin locations use a round solder pad.

The connector hole pattern is made to accept 58 0.025″ square pins soldered into the board. The numbering for this connector is shown in *Figure 2*.

Note: The numbering of this connector is specified to match up with standard 0.050″ centerline flat cable connectors. Because of the location of pin 1 of this hole pattern, the mating pins for this connector should normally be on the bottom of the board. If a connector is instead attached to the top side of the board, the even- and odd-numbered pins of the connector are effectively swapped. This means that conductor 1 of a cable attached to the top side of the

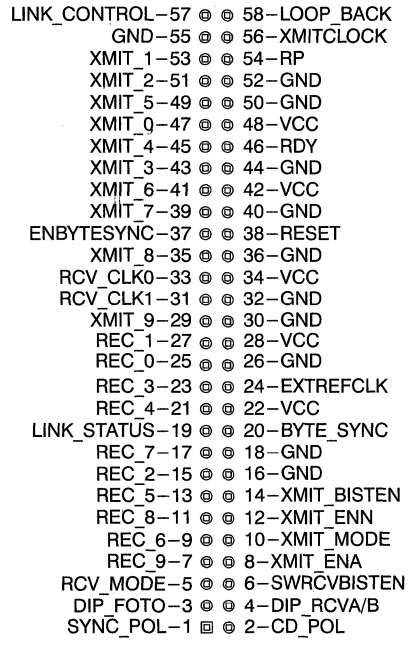


Figure 2. JP2 Pin Numbering, Top Side of Board View

board is in reality connected to the signal listed for pin 2 in *Table 1*.

JP3—60-Position Board-Edge

The 60-position board-edge connector (JP3) is a section of gold plated 0.062″ board finger-stock that connects to the same signals as JP2. Contact centerline for this connector is 0.1″, with even- and odd-numbered signals on opposing sides of the board.

To prevent the evaluation board from being plugged into a mating connector backwards (and possibly damaging it), a 0.040″ x 0.450″ keying slot is present between contacts 3/4 and 5/6. The pin numbering for this connector is shown in *Figure 3*.

Note: The numbering of this connector is specified to match up with standard 0.050″ centerline flat-cable connectors. Because of the location of pin 1 of this board-edge connector, the mating connector would normally be a mass-terminate board-edge to flat-cable type connector. If a standard board-edge connector is used instead, the even and odd numbered pins of the connector are effectively swapped. This means that pin 1 of a standard board-edge connector is in reality connected to the signal listed for pin 2 in *Table 1*.

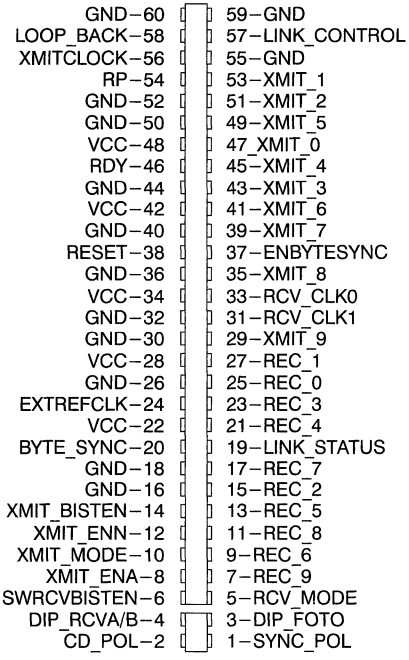


Figure 3. JP3 Pin Numbering, Edge of Board

JP4—OLC-Compatibility Connector

The JP4 (OLC-compatibility) connector is located on the bottom (passive-component) side of the board. Pin 1 of this connector is identified on the board by a square solder pad. The remaining pins use a round solder pad.

For the CY9266 Evaluation Board, pins of sufficient length are present so that analysis equipment may be attached to these signal pins on the top (active-component) side of the board while it is plugged into a mating connector. The numbering sequence for the JP4 connector pins is shown in *Figure 4*.

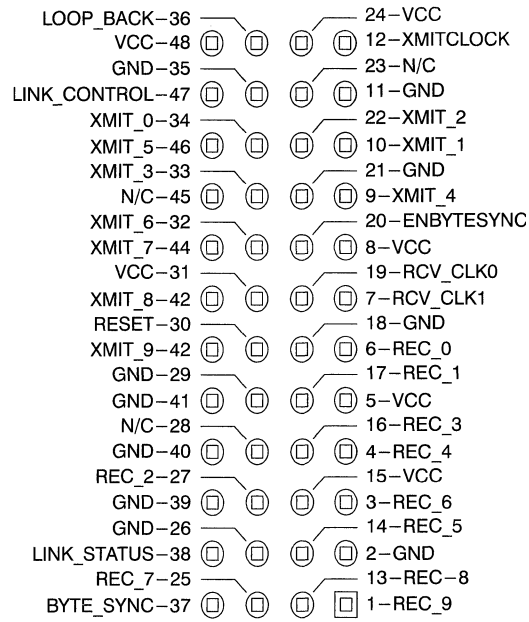


Figure 4. JP4 Pin Numbering, Top Side of Board View (Pins Are On the Bottom)

The connector is made from 48 0.025″ square pins soldered into the board. To allow full mating with an OLC-compatible connector, these pins must extend at least 0.250″ beyond the bottom surface of the board.

Connector Pinouts

The CY9266 provides three interface connectors to the user: JP2, JP3, and JP4. *Table 1* shows which signal is present on each connector pin.

Table 1. I/O Connector Pinouts

Pin No.	JP3	JP2	JP4	Pin No.	JP3	JP2	JP4
1	SYNC_POL	SYNC_POL	REC_9	31	RCV_CLK1	RCV_CLK1	VCC
2	CD_POL	CD_POL	GND	32	GND	GND	XMIT_6
3	DIP_FOTO	DIP_FOTO	REC_6	33	RCV_CLK0	RCV_CLK0	XMIT_3
4	DIP_RCVA/B	DIP_RCVA/B	REC_4	34	VCC	VCC	XMIT_0
5	RCV_MODE	RCV_MODE	VCC	35	XMIT_8	XMIT_8	GND
6	SWRCVBISTEN	SWRCVBISTEN	REC_0	36	GND	GND	LOOP_BACK
7	REC_9	REC_9	RCV_CLK1	37	ENBYTESYNC	ENBYTESYNC	BYTE_SYNC
8	XMIT_ENA	XMIT_ENA	VCC	38	RESET	RESET	LINK_STATUS
9	REC_6	REC_6	XMIT_4	39	XMIT_7	XMIT_7	GND
10	XMIT_MODE	XMIT_MODE	XMIT_1	40	GND	GND	GND
11	REC_8	REC_8	GND	41	XMIT_6	XMIT_6	GND
12	XMIT_ENN	XMIT_ENN	XMITCLOCK	42	VCC	VCC	XMIT_9
13	REC_5	REC_5	REC_8	43	XMIT_3	XMIT_3	XMIT_8
14	XMIT_BISTEN	XMIT_BISTEN	REC_5	44	GND	GND	XMIT_7
15	REC_2	REC_2	VCC	45	XMIT_4	XMIT_4	N/C
16	GND	GND	REC_3	46	RDY	RDY	XMIT_5
17	REC_7	REC_7	REC_1	47	XMIT_0	XMIT_0	LINK_CONTROL
18	GND	GND	GND	48	VCC	VCC	VCC
19	LINK_STATUS	LINK_STATUS	RCV_CLK0	49	XMIT_5	XMIT_5	
20	BYTE_SYNC	BYTE_SYNC	ENBYTESYNC	50	GND	GND	
21	REC_4	REC_4	GND	51	XMIT_2	XMIT_2	
22	VCC	VCC	XMIT_2	52	GND	GND	
23	REC_3	REC_3	N/C	53	XMIT_1	XMIT_1	
24	EXTREFCLK	EXTREFCLK	VCC	54	RP	RP	
25	REC_0	REC_0	REC_7	55	GND	GND	
26	GND	GND	GND	56	XMITCLOCK	XMITCLOCK	
27	REC_1	REC_1	REC_2	57	LINK_CONTROL	LINK_CONTROL	
28	VCC	VCC	N/C	58	LOOP_BACK	LOOP_BACK	
29	XMIT_9	XMIT_9	GND	59	GND		
30	GND	GND	RESET	60	GND		

Table 2. Transmit Bus Signal Name Map

Transmit Bus Input Pin Name	HOTLink Transmitter Pin Name	
	Encoded Mode	Bypass Mode
XMIT_0	SC/$\overline{D}$	Da
XMIT_1	D0	Db
XMIT_2	D1	Dc
XMIT_3	D2	Dd
XMIT_4	D3	De
XMIT_5	D4	Di
XMIT_6	D5	Df
XMIT_7	D6	Dg
XMIT_8	D7	Dh
XMIT_9	SVS	Dj

Signal Naming Conventions

There are three types of signal names used throughout this document: I/O connector pin names, onboard signal names, and HOTLink Transmitter and Receiver pin names. Except for the transmit and receive data buses, these names are unique.

The names used for the transmit and receive data bus pins on connectors JP2, JP3, and JP4 are different from the signal names present on the HOTLink Transmitter and Receiver. The functional names for these signals also change depending on the current operating mode of the HOTLink Transmitter or Receiver. *Table 2* lists the transmit data bus signals and the names mapped to them in each transmitter mode.

The output data bus from the HOTLink Receiver is pipelined with a single register stage between the receiver outputs and the board output pins. *Table 3* lists the receive data bus signals and the names mapped to them in each receiver mode.

Table 3. Receive Bus Signal Name Map

Receive Bus Output Pin Name	HOTLink Receiver Pin Name	
	Decode Mode	Bypass Mode
REC_0	SC/$\overline{D}$	Qa
REC_1	Q0	Qb
REC_2	Q1	Qc
REC_3	Q2	Qd
REC_4	Q3	Qe
REC_5	Q4	Qi
REC_6	Q5	Qf
REC_7	Q6	Qg
REC_8	Q7	Qh
REC_9	RVS	Qj

Signal Descriptions

The I/O signals listed in *Table 1* fall into six groups: power, switched control, control, status, clock, and data. These signals are described in *Table 4*.

Table 4. I/O Signal Descriptions

Signal Name	Group	Description
V_{CC}	Power	+5 VDC @ 1.0A typical
GND	Power	Ground
XMIT_BISTEN	Input, Switched Control	**Transmitter BIST Enable (S1-1).** When this signal is LOW, the HOTLink Transmitter is placed into its BIST mode. Exact operation of the transmitter is also determined by the settings of the $\overline{ENA}$ (S1-4) and $\overline{ENN}$ (S1-3) signals. With both $\overline{ENA}$ and $\overline{ENN}$ HIGH, the transmitter outputs an alternating 0−1 pattern (D10.2 or D21.5). If either $\overline{ENA}$ or $\overline{ENN}$ is LOW, the transmitter sends a repeating 511-character test sequence. The receiver contains a matching mode that allows this transmitter BIST mode to be used to test the entire serial link without external hardware. The transmitter BIST enable is kept separate from the receiver BIST enable on this board to allow each component to be tested with external patterns that are not part of the BIST sequence.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
XMIT_MODE	Input, Switched Control	**Encoder Mode Select (S1-2).** This signal is used to select whether pre-encoded (10-bit) or non-encoded (8-bit) data is clocked into the HOTLink Transmitter. When LOW (Encoded mode), this input enables the internal 8B/10B encoder and accepts 8-bit parallel data from the transmitter data bus (D0−D7 as listed in *Table 2*). When HIGH (Bypass mode), the encoder is bypassed and a 10-bit pattern is accepted (Da−Dj as listed in *Table 2)*.
XMIT_ENN	Input, Switched Control	**Enable Next Parallel Transmitter Data (S1-3).** This signal is used to control when data is loaded into the HOTLink Transmitter. When this signal is LOW at the rising edge of CKW, the data present on the transmitter inputs at the next rising edge of CKW is loaded, processed, and sent. When this signal is HIGH, the transmitter ignores the data present on its inputs at the next rising edge of CKW and instead inserts a SYNC character (K28.5) to fill in the data stream. When $\overline{\text{ENA}}$ is used for data control, the $\overline{\text{ENN}}$ signal should be tied HIGH, but may be used to enable BIST mode.
XMIT_ENA	Input, Switched Control	**Enable Parallel Transmitter Data (S1-4).** This signal is used to control when data is loaded into the HOTLink Transmitter. When LOW at the rising edge of CKW, the data present on the transmitter inputs is loaded, processed, and sent. When this signal is HIGH, the transmitter ignores the data present on its inputs and instead inserts a SYNC character (K28.5) to fill in the data stream. When $\overline{\text{ENN}}$ is used for data control, the $\overline{\text{ENA}}$ signal should be tied HIGH, but may be used to enable BIST mode.
SWRCVBISTEN	Input, Switched Control	**Receiver BIST Enable (S1-5).** When this signal is LOW, the HOTLink Receiver monitors the data stream for the BIST loop initialization character (D0.0). This signal also enables the BIST PLD (CY7C344-U8), which is used to monitor the progress and status of the BIST loop through the receiver $\overline{\text{RDY}}$ and RVS outputs. When the receiver detects the initialization character, it begins comparing received data with a built-in data sequence that can be used to verify the proper functionality of the transmitter, receiver, and the serial link connecting them. The receiver BIST enable is kept separate from the transmitter BIST enable on this board to allow each component to be tested with external patterns that are not part of the BIST sequence.
RCV_MODE	Input, Switched Control	**Receiver Mode Select (S1-6).** This signal is used to select whether encoded (10-bit) or non-encoded (8-bit) data is output from the receiver. When LOW (Decode mode), this input enables the internal 10B/8B decoder and outputs 8-bit parallel data (Q0−Q7 as listed in *Table 3*). When HIGH (Bypass mode), the decoder is bypassed and a 10-bit pattern is output (Qa−Qj as listed in *Table 3*).

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
DIP_RCVA/B	Input, Switched Control	**DIP-Switch Controlled Receiver A/$\overline{B}$ Port Select (S1-7).** This signal is used to determine which port (INA± or INB±) the receiver uses for the input serial data stream. When LOW, this signal selects the receiver B port that is directly connected to the C port on the transmitter. When HIGH, this signal selects the receiver A port that is connected to the optical receiver output. This signal is also routed through jumper block JP1. In order for this signal to control the port selection of the receiver, it is necessary to have a shorting jumper across the X and Y pins of JP1-C. To allow the LOOP_BACK signal on the I/O connectors (JP2, JP3, and JP4) to control the A/$\overline{B}$ port selection, this jumper should be moved to JP1-B.
DIP_FOTO	Input, Switched Control	**DIP-Switch Controlled FOTO (S1-8).** This signal is used to enable the A and B differential output drivers of the HOTLink Transmitter. When this signal is LOW, the differential outputs are allowed to follow the pattern of the data serialized by the transmitter. When this signal is HIGH, the A and B differential outputs of the transmitter are driven to a logic zero state (+ output is logic HIGH, − output is logic LOW). This places an attached optical transmitter in a state where no light is output. This signal is also routed through jumper block JP1. In order for this signal to control the FOTO (fiber-optic transmitter-off) enable on the transmitter, it is necessary to have a shorting jumper across the X and Y pins of JP1-E. To allow the LINK_CONTROL signal on the I/O connectors (JP2, JP3, and JP4) to control the FOTO enable, this jumper should be moved to JP1-F.
CD_POL	Input, Switched Control	**Carrier-Detect Polarity Select (S1-9).** This input selects the output polarity of the LINK_STATUS signal. When LOW, the LINK_STATUS signal is HIGH when a valid carrier is present. When HIGH, the LINK_STATUS signal is LOW when a valid carrier is present.
SYNC_POL	Input, Switched Control	**Byte Sync Polarity Select (S1-10).** This input, in conjunction with the HOTLink Receiver MODE input, selects the active level of the BYTE_SYNC signal. When LOW with the receiver in Bypass mode, the BYTE_SYNC signal is LOW when a K28.5 SYNC character is present on the receive data bus. When HIGH with the receiver in Bypass mode, the BYTE_SYNC signal is HIGH when a K28.5 SYNC character is present on the receive data bus. When LOW with the receiver in Decode mode, the BYTE_SYNC output remains HIGH for strings of K28.5 SYNC characters, or while awaiting the first K28.5 SYNC character after being placed into Reframe mode (RF is set HIGH). When HIGH with the receiver in Decode mode, the BYTE_SYNC output remains LOW for strings of K28.5 SYNC characters, or while awaiting the first K28.5 SYNC character after being placed into Reframe mode (RF is set HIGH).

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
LOOP_BACK	Input, Control	**Loopback Control.** This signal is used to determine which port (A or B) the HOTLink Receiver uses for the input serial data stream. When LOW, this signal selects the receiver B port that is connected directly to the transmitter C port. When HIGH, this signal selects the receiver A port that is connected to the optical receiver output. This signal is also routed through jumper block JP1. In order for this signal to control the port selection of the receiver, it is necessary to have a shorting jumper across the X and Y pins of JP1-B. To allow the DIP_RCVA/B signal (S1-7, also present on JP2 and JP3) to control the $\overline{A}/\overline{B}$ port selection, this jumper should be moved to JP1-C.
ENBYTESYNC	Input, Control	**Enable Byte Sync Detect.** This signal controls when the HOTLink Receiver is allowed to reframe to the incoming serial data (e.g., acquire byte sync). When this signal is HIGH, each K28.5 SYNC character received in the shifter will frame the data that follows. When this signal is LOW, the framing logic in the receiver is disabled. Because the CKR output of the receiver must line up with the reframed data, it is possible to generate significant phase jumps in the CKR clock. To prevent the generation of very short high or low pulses on the CKR output (which could cause timing violations in downstream logic) the Cypress HOTLink Receiver uses look-ahead hardware to prevent these short pulses. Instead, a portion of the clock period for the character preceding the reframed data is lengthened.
LINK_ CONTROL	Input, Control	**Link Control.** This signal is used to enable the A and B differential output drivers of the HOTLink Transmitter. When this signal is LOW, the differential outputs are allowed to follow the pattern of the data serialized by the transmitter. When this signal is HIGH, the A and B differential outputs of the transmitter are driven to a logic zero state (+ output is logic HIGH, − output is logic LOW). This places an attached optical transmitter in a state where no light is output. This signal is also routed through jumper block JP1. In order for this signal to control the FOTO enable on the transmitter, it is necessary to have a shorting jumper across the X and Y pins of JP1-F. To allow the DIP_FOTO signal on the I/O connectors (JP2 and JP3) to control the FOTO enable, this jumper should be moved to JP1-E.
RESET	Output, Status	**Reset/Power OK.** This output is used to emulate the voltage monitor function present on the OLC card. It remains active (LOW) until the V_{CC} input to the board is above 4.65 VDC. This output also becomes active when the BIST RESET switch (S2) is pressed.
LINK_STATUS	Output, Status	**Link Status.** This signal operates as a carrier-detect status for the serial interface. The polarity of this signal is determined by the CD_POL input (S1-9). When CD_POL is LOW, LINK_STATUS drives HIGH when a carrier is present. When CD_POL is HIGH, LINK_STATUS drives LOW when a carrier is present.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
RP	Output, Clock	**Read Pulse.** This is a 60% LOW duty-cycle pulse train suitable for clocking data out of Cypress's CY7C42X family of asynchronous FIFOs. This pulse is generated by the HOTLink Transmitter in response to the XMIT_ENA input being active at the rising edge of CKW. For repeated pulses the $\overline{RP}$ period is the same as CKW, yet is totally independent of the duty cycle of CKW. When the transmitter is in BIST mode, the $\overline{RP}$ signal remains HIGH for all but the last byte of the BIST loop, where it pulses LOW.
XMITCLOCK	Input, Output, Clock	**Transmitter External Clock.** This is the external byte-rate clock input. This clock is used to drive the transmitter CKW input. To allow for operation using the on-board oscillator, the XMITCLOCK signal is run through jumper block JP1. To operate using an external HOTLink Transmitter clock source, a shorting jumper should be placed across pins X and Y of JP1-G. To use the on-board oscillator instead, this shorting jumper should be moved to connect pin JP1-GY to JP1-HY. When operated from XMITCLOCK, the receiver REFCLK may also be set to use this same clock. This is done by placing a shorting jumper across pins JP1-GX and JP1-HX. To allow the receiver REFCLK to operate from the on-board oscillator, this jumper should be moved to connect the X and Y pins of JP1-I. The on-board oscillator may also be driven out on the XMITCLOCK line by placing a shorting jumper across pins X and Y of JP1-H.
EXTREFCLK	Input, Output, Clock	**External Reference Clock.** This byte-rate clock is used to drive the HOTLink Receiver REFCLK from an external source other than XMITCLOCK. This input may be used to test the tracking and capture range of the receiver PLL. It may also be used to operate the receiver at a different data rate from the transmitter. To allow the receiver PLL to properly lock to the received serial stream, this clock must be within 0.1% of the clock used to generate the received serial data. To drive the receiver REFCLK from this clock source, a shorting jumper should be placed across pins JP1-IX and JP1-JX. The on-board oscillator may also be selected to drive the EXTREFCLK line by placing a shorting jumper across pins X and Y of JP1-J. With this jumper in place it is still possible to drive the receiver REFCLK input from the on-board oscillator by placing a shorting jumper across the X and Y pins of JP1-I.
RCV_CLK0	Output, Clock	**Receive Clock 0.** This is the byte-rate recovered clock used for received data. The period of this clock is determined by the serial data rate entering the HOTLink Receiver. The duty-cycle of this signal is determined by the receiver and is fixed at 50%. This clock may experience a large phase jump when reframing to a serial data stream. The phasing on this clock is such that the rising edge of the clock occurs coincident with the start of each interval where a character is present on the output received data bus. This signal is a buffered form of the HOTLink Receiver CKR clock.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
RCV_CLK1	Output, Clock	**Receive Clock 1.** This is the byte-rate recovered clock used for received data. The period of this clock is determined by the serial data rate entering the HOTLink Receiver. The duty-cycle of this signal is determined by the receiver and is fixed at 50%. This clock may experience a large phase jump when reframing to a serial data stream. The phasing on this clock is such that the rising edge of the clock occurs near the center of each interval where a character is present on the output received data bus. This signal is a buffered and inverted form of the HOTLink Receiver CKR clock.
RDY	Output, Clock	**$\overline{\text{RDY}}$ (Ready).** This signal is used both as a HOTLink Receiver data output clock and a status indicator for the receiver when in BIST mode. This is an unbuffered output from the receiver. It is normally used to clock valid data from the receiver data bus into asynchronous FIFOs. Because of the additional pipeline register in the data bus (added for OLC compatibility) this signal will operate one byte *prior* to the data being available at the I/O connectors.
BYTE_SYNC	Output, Data	**Byte Sync Detected.** This signal is a pipelined form of the receiver $\overline{\text{RDY}}$ output. This additional pipeline stage for the $\overline{\text{RDY}}$ signal (and the rest of the receiver data bus) was added to match the specific timing of the OLC Byte Sync signal. The active level of this output is determined both by the operating mode of the HOTLink Receiver and by the state of the SYNC_POL input. With the HOTLink Receiver in Bypass mode, the BYTE_SYNC signal is used as a K28.5 SYNC character indicator. With SYNC_POL LOW, BYTE_SYNC is LOW when a K28.5 SYNC character is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH when a K28.5 SYNC character is present on the receive data bus. With the receiver in Decode mode, the BYTE_SYNC signal is used as a valid data indicator. With SYNC_POL LOW, BYTE_SYNC is LOW whenever a usable data byte is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH whenever a usable data byte is present on the receive data bus.
REC_9	Output, Data	**RVS(Qj).** This signal is a series-terminated, pipelined form of the HOTLink Receiver RVS(Qj) signal. This termination and additional pipeline stage for the RVS(Qj) signal (and the rest of the receive data bus) was added to match the specific timing and signal characteristics of the OLC card.
REC_8	Output, Data	**Q7(Qh).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q7(Qh) signal.
REC_7	Output, Data	**Q6(Qg).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q6(Qg) signal.
REC_6	Output, Data	**Q5(Qf).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q5(Qf) signal.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
REC_5	Output, Data	**Q4(Qi).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q4(Qi) signal.
REC_4	Output, Data	**Q3(Qe).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q3(Qe) signal.
REC_3	Output, Data	**Q2(Qd).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q2(Qd) signal.
REC_2	Output, Data	**Q1(Qc).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q1(Qc) signal.
REC_1	Output, Data	**Q0(Qb).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q0(Qb) signal.
REC_0	Output, Data	**SC/$\overline{D}$(Qa).** This signal is a series-terminated, pipelined form of the HOTLink Receiver SC/$\overline{D}$(Qa) signal.
XMIT_9	Input, Data	**SVS(Dj).** This signal is the SVS(Dj) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_8	Input, Data	**D7(Dh).** This signal is the D7(Dh) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_7	Input, Data	**D6(Dg).** This signal is the D6(Dg) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_6	Input, Data	**D5(Df).** This signal is the D5(Df) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_5	Input, Data	**D4(Di).** This signal is the D4(Di) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_4	Input, Data	**D3(De).** This signal is the D3(De) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_3	Input, Data	**D2(Dd).** This signal is the D2(Dd) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_2	Input, Data	**D1(Dc).** This signal is the D1(Dc) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_1	Input, Data	**D0(Db).** This signal is the D0(Db) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_0	Input, Data	**SC/$\overline{D}$(Da).** This signal is the SC/$\overline{D}$(Da) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.

Power Signals

The CY9266 Evaluation Board is designed to operate from a single +5V ±10% DC supply capable of delivering 1.0A (typical). All V_{CC} and GND pins on JP2, JP3, and JP4 are (respectively) common to each other. There are no distinctions made for separate supplies pins for the different logic sections.

Switched Control Signals

The CY9266 Evaluation Board contains a 10-position DIP switch (S1). This switch is connected in parallel with a number of control signals on JP2 and JP3. Each of these control signals is pulled-up by a 5-kΩ resistor through R-pack R20. None of these Switched Control signals are available at the JP4 connector.

The signals present in this group are:

- XMIT_BISTEN (S1-1)
- XMIT_MODE (S1-2)
- XMIT_ENN (S1-3)
- XMIT_ENA (S1-4)
- SWRCVBISTEN (S1-5)
- RCV_MODE (S1-6)
- DIP_RCVA/B (S1-7)
- DIP_FOTO (S1-8)
- CD_POL (S1-9)
- SYNC_POL (S1-10)

To allow these signals to be controlled through the external connectors (JP2 and JP3), the corresponding S1 switch must be in the off (open) position. Care should be taken when driving these signals, as any switch inadvertently left in the closed position will present a direct short to ground for an attached driver.

Control Signals

In addition to the Switched Control signals that are only present on JP2 and JP3, three additional control inputs are present that connect to JP2, JP3, and JP4.

These control signals are:

- LOOP_BACK
- ENBYTSYNC
- LINK_CONTROL

These control inputs are connected directly to the HOTLink Transmitter or Receiver. Because the HOTLink parts contain internal pull-up resistors on their TTL compatible inputs, these signals may be driven with either open-collector buffers, CMOS, or TTL drive levels.

Status Signals

Two status output signals (RESET and LINK_STATUS) are provided at all three I/O connectors. The RESET signal is a slow-speed signal and does not require the series termination used with LINK_STATUS.

Clock Signals

Six signals are available at the I/O connectors that are used as clocks in some form. Two of these (XMITCLOCK and EXTREFCLK) are input/output clocks that are routed through the JP1 jumper block, and three are output clocks.

These clock signals are:

- XMITCLOCK
- EXTREFCLK
- $\overline{RP}$
- $\overline{RDY}$
- RCV_CLK0
- RCV_CLK1

Of the output clocks, the $\overline{RP}$ and $\overline{RDY}$ signals are only available at JP2 and JP3. The $\overline{RP}$ signal is generated in the HOTLink Transmitter and is used for reading data from asynchronous FIFOs, while the $\overline{RDY}$ signal is generated in the HOTLink Receiver and is used for writing data into asynchronous FIFOs. When interfacing to clocked FIFOs (CY7C44X, CY7C45X), the $\overline{RP}$ signal is not normally used. Because these signals are not present in JP4, they are not series terminated.

The other two output clocks (RCV_CLK0 and RCV_CLK1) are a buffered form of the recovered CKR clock from the receiver. The RCV_CLK1 signal is an inverted form of RCV_CLK0.

Data Signals

The CY9266 Evaluation Board has two data buses: one input (to the HOTLink Transmitter) and one output (from the HOTLink Receiver).

The input data bus consists of ten parallel transmit data signals that are sampled at the rising edge of the HOTLink Transmitter CKW clock. In addition to these ten signals, $\overline{\text{ENN}}$ and $\overline{\text{ENA}}$ (while part of the Switched Control signals) may also be considered part of the data bus as they are also sampled at this same time. While the XMIT_BISTEN input is also sampled at this same time, it is not normally used to transfer data and is therefore not considered part of the input data bus.

The output data bus is comprised of ten parallel received data signals that are synchronous to the HOTLink Receiver CKR clock. To meet specific timing requirements for OLC compatibility, there is also an external pipeline register between the HOTLink Receiver data bus output, and the received data bus connected to JP2, JP3, and JP4.

One other signal, BYTE_SYNC, is also clocked through this pipeline register and is thus considered part of the data bus.

All signals on this output bus are series-terminated with a 22Ω inline resistor to minimize transmission line ringing.

Configuration Settings

The CY9266 board may be user-configured to allow many modes of operation. This configuration is performed through the jumper block JP1 and the option select switch S1.

JP1 Jumper Block

The JP1 jumper block is used for configuring those options of the CY9266 that are (primarily) either to protect the board from signal contention, or for those signals having multiple sources and destinations. These functions are:

- Receiver Mode Select
- Receiver Loopback Source Select
- Transmitter Mode Select
- Transmitter FOTO Source Select
- Transmitter Clock (CKW) Source Select
- Receiver Reference Clock (REFCLK) Source Select

JP1 exists as a 2 x 10 matrix of 0.025″ square pins on the top of the board. The rows in this matrix are identified on the top silk screen as A through J. The columns are identified as X and Y. A drawing of the JP1 jumper block is shown in *Figure 5*.

Receiver Mode Select

This jumper ties pins X and Y of JP1-A together. It is used to connect the receiver's MODE select pin to the option select switch (S1-6), and to allow the HOTLink Receiver mode to be set to the clock Test mode (see *Figure 13*). The three modes of receiver operation are:

- Decode Mode—S1-6 ON (closed)
- Bypass Mode—S1-6 OFF (open)
- Test Mode—JP1-A, X and Y open

Because this clock Test mode is not normally used for communications testing, the jumper (JP1-A) is

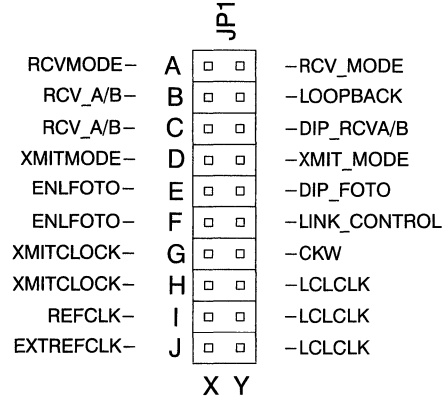


Figure 5. JP1, Top Side View

permanently wired in place with a foil trace on the bottom of the board. For those users who wish to actually place the receiver in Test mode, it may be necessary to cut this foil on the back of the board.

Once this foil has been cut, it will be necessary to use a shorting jumper across pins X and Y of JP1-A to allow the two data modes of the receiver to be set by the option select switch (S1-6) and the RCV_MODE signal on JP2 and JP3.

Receiver Source Loopback Select

This function uses two positions (JP1-B and JP1-C) of the jumper block to select the source of the HOTLink Receiver loopback signal. Because this jumper is used to select between one of two sources, only one of these two positions (JP1-B or JP1-C) may contain a shorting jumper at any one time (see *Figures 10* and *11*).

By placing a shorting jumper across pins X and Y of JP1-B, the receiver loopback ($A/\overline{B}$) input is then controlled by the LOOP_BACK signal on JP2, JP3, and JP4. If this shorting jumper is moved to JP1-C, then the receiver loopback input is controlled by the option select switch (S1-7) and the RCV_MODE signal on JP2 and JP3. If a jumper is not present in either position, the INA± path is selected (external serial data).

Transmitter Mode Select

This jumper ties pins X and Y of JP1-D together. It is used to connect the transmitter MODE select pin to the option select switch, and to allow the HOTLink Transmitter mode to be set to the clock Test mode (see *Figure 7*). The three modes of transmitter operation are

- Encode Mode—S1-2 ON (closed)
- Bypass Mode—S1-2 OFF (open)
- Test Mode—JP1-D, X and Y open

Because this clock Test mode is not expected to be used for normal data communications testing, the jumper (JP1-D) is permanently wired in place with a foil trace on the bottom of the board. For those users who wish to actually place the transmitter in Test mode, it may be necessary to cut this foil on the back of the board.

Once this foil has been cut, it will be necessary to use a jumper across JP1-D to allow the two data modes of the transmitter to be set by the option select switch (S1-2) and the XMIT_MODE signal on JP2 and JP3.

Transmitter FOTO Source Select

This function uses two positions (JP1-E and JP1-F) of the jumper block to select the source of the HOTLink Transmitter FOTO signal. Because this jumper is used to select from one of two sources, only one of these two positions (E or F) may contain a jumper at any one time (see *Figures 8* and *9*).

By placing a shorting jumper across pins X and Y of JP1-F, the HOTLink Transmitter FOTO signal is then controlled by the LINK_CONTROL signal on JP2, JP3, and JP4. If this shorting jumper is moved to JP1-E, then the transmitter FOTO signal is controlled by the option select switch (S1-8) and the DIP_FOTO signal on JP2 and JP3. If a jumper is not present in either position, the transmitter OUTA± and OUTB± differential drivers are placed in a mode where a differential logic 0 is driven.

Transmitter Clock Source Select

The HOTLink Transmitter CKW clock can be sourced from two different signals: LCLCLK from the on-board oscillator and XMITCLOCK from JP2, JP3, and JP4 (see *Figure 7*).

To select the on-board oscillator, a shorting jumper should be placed across pins JP1-GY and JP1-HY. To select the XMITCLOCK signal, this shorting jumper should be moved to connect pins X and Y of JP1-G. To allow the transmitter to operate, it is necessary for a jumper to be in one (and only one) of these two positions.

Receiver Reference Clock Source Select

The HOTLink Receiver REFCLK signal can be sourced from three different signals: LCLCLK from the on-board oscillator, XMITCLOCK (from JP2, JP3, and JP4), and EXTREFCLK (from JP2 and JP3) (see *Figure 13*).

To select the on-board oscillator, a shorting jumper should be placed across the X and Y pins of JP1-I. To select the XMITCLOCK signal, this shorting jumper should be moved to connect pin X of JP1-I to pin X of JP1-H. To select the EXTREFCLK signal (used for PLL range testing), the shorting jumper should be placed across pin X of JP1-I and pin X of JP1-J. To allow the receiver to operate it is necessary for a jumper to be in one (and only one) of these three positions.

S1 Option Select Switch

The S1 Option Select Switch is used for configuring those options of the CY9266 that may be changed on a regular basis or are used to operate the board in a standalone mode. These functions are

- Transmitter BIST Enable
- Encoder Mode Select
- Enable Next Parallel Transmitter Data
- Enable Parallel Transmitter Data
- Receiver BIST Enable
- Receiver Mode Select
- Receiver A/B Port Select
- Transmitter FOTO Enable
- Carrier-Detect Polarity
- Byte Sync Polarity

S1 exists as a 10-position DIP switch. The switch positions (numbered 1 through 10) are identified on the top of the switch. When a switch is on (closed), the signal connected to that switch is tied directly to ground. When a switch is off (open), the signal on that switch is pulled up through a 5-kΩ resistor in R-pack R20.

These signals are also connected to pins on JP2 and JP3 to allow external logic to control these functions. A drawing of the S1 option select switch is shown in *Figure 6*.

Transmitter BIST Enable

Switch S1-1 (XMIT_BISTEN) is used to enable the HOTLink Transmitter BIST function. When this switch is on (closed), the $\overline{\text{BISTEN}}$ input to the transmitter is pulled LOW, placing the transmitter into its BIST loop. The exact patterns transmitted are determined by the levels on the XMIT_ENN and XMIT_ENA signals, located on S1-3 and S1-4 respectively (see *Figure 7*).

Encoder Mode Select

Switch S1-2 (XMIT_MODE) is used to select the data encoding mode of the HOTLink Transmitter. When this switch is on (closed), the internal 8B/10B encoder is enabled and the 8-bit data characters are encoded into 10-bit transmission characters. When this switch is off (open), the encoder is bypassed and the transmitter accepts 10-bit patterns for direct serialization (see *Figure 7*).

Enable Next Parallel Transmitter Data

Switch S1-3 (XMIT_ENN) is used, along with S1-1 (transmitter BIST enable) and S1-4 (XMIT_ENA), to select which data patterns are sent during HOTLink Transmitter BIST operations (see *Figure 7*).

If BIST is enabled (S1-1 on and S1-4 off), setting this switch off (open) causes the transmitter to send an alternating 1−0 pattern (D10.2 or D21.5). When turned on (closed), it enables an internal pattern generator in the transmitter that generates a repeating sequence of 511 10-bit patterns.

For normal data transfer operations this switch should remain off, with the XMIT_ENN signal controlled externally through JP2 and JP3.

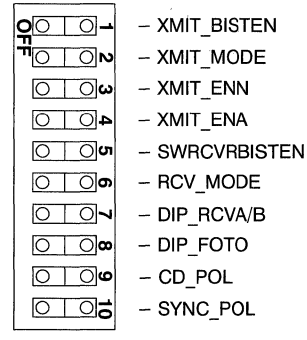


Figure 6. S1 Option Select Switch

Enable Parallel Transmitter Data

Switch S1-4 (XMIT_ENA) is used, along with S1-1 (transmitter BIST enable) and S1-3 (XMIT_ENN), to select which data patterns are sent by the HOTLink Transmitter during BIST operations (see *Figure 7*).

If BIST is enabled (S1-1 on and S1-3 off), setting S1-4 off (open) causes the transmitter to send an alternating 1−0 pattern (D10.2 or D21.5). When turned on (closed), it enables an internal pattern generator in the transmitter that produces a repeating sequence of 511 10-bit patterns.

For normal data transfer operations this switch should remain off, with the XMIT_ENA signal controlled externally through JP2 and JP3.

When operated from the JP4 system connector, this switch should be turned on (closed), because the system hardware is required to provide a valid 10-bit transmission character or data byte for each CKW clock.

Receiver BIST Enable

Switch S1-5 (SWRCVBISTEN) is used to enable the HOTLink Receiver BIST function (see *Figure 13*). When this switch is on (closed), the receiver awaits a D0.0 transmission character (sent once per BIST loop). When this character is detected the BIST state machine in the receiver begins matching the following received transmission characters with its internal pattern generator. This pattern generator follows the same sequence of patterns as those sent by the HOTLink Transmitter when sending its BIST sequence.

When this switch is off (open), the HOTLink Receiver operates in one of its two data modes (Decode or Bypass).

Receiver Mode Select

Switch S1-6 (RCV_MODE) is used to select the data decoding mode of the HOTLink Receiver (see *Figure 13*). When this switch is on (closed), the internal 10B/8B decoder is enabled and the received 10-bit transmission characters are decoded into 8-bit data characters. When this switch is off (open), the decoder is bypassed and the receiver outputs 10-bit transmission characters directly to the output data and status pins.

Receiver A/$\overline{B}$ Port Select

Switch S1-7 (DIP_RCVA/B) is used to select which input port (A or $\overline{B}$) the HOTLink Receiver should use for receiving serial data (see *Figures 10* and *11*). While the A/$\overline{B}$ input of the receiver is a 100K ECL (emitter-coupled logic) compatible input, it is connected here to allow control from a switch or TTL driver. This requires use of an external resistor network, connected between that input and the select switch, to allow full rail-to-rail swings to be used.

When this switch is on (closed), the INB+ input to the HOTLink Receiver is selected. This input is directly connected to the OUTC+ output from the HOTLink Transmitter. This is the Local Loopback mode for the CY9266 evaluation board that allows the transmitter and receiver to be tested without an external serial data cable or optical module.

When this switch is off (open), the INA± differential input of the receiver is enabled to accept data from the optical module (U4) or copper cable.

Transmitter FOTO Enable

Switch S1-8 (DIP_FOTO) is used to enable the OUTA± and OUTB± differential output drivers of the HOTLink Transmitter. When this switch is on (closed), the differential outputs are allowed to follow the pattern of the data serialized by the transmitter (see *Figures 8* and *9*). When this switch is off (open), the OUTA± and OUTB± differential outputs of the transmitter are driven to a logic zero state (+ output is logic LOW, − output is logic HIGH). This places an attached optical transmitter in a state where no light is output, or presents no transitions on a copper cable.

Carrier-Detect Polarity

Switch S1-9 is used to control the active level of the carrier-detect output signal, LINK_STATUS. When this switch is on (closed) LINK_STATUS is driven HIGH when a carrier is present and LOW when one is not. When this switch is off (open) these levels are reversed (see *Figure 13*).

The carrier-detect status is also displayed on one of the decimal point indicators of the two-digit BIST display. When the indicator is on, a carrier is present. The state of S1-9 has no affect on the operation of this indicator.

Byte Sync Polarity

Switch S1-10 is used to control the active level of the BYTE_SYNC output signal. This level is also affected by the operating mode of the HOTLink Receiver (S1-6) (see *Figure 13*).

With the HOTLink Receiver in Bypass mode, the BYTE_SYNC signal is used as a K28.5 SYNC character indicator. With SYNC_POL LOW, BYTE_SYNC is LOW when a K28.5 SYNC character is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH when a K28.5 SYNC character is present on the receive data bus.

With the receiver in Decode mode, the BYTE_SYNC signal is used as a valid data indicator. With SYNC_POL LOW, BYTE_SYNC is LOW whenever a usable data byte is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH whenever a usable data byte is present on the receive data bus.

CY9266 Schematic

The complete schematic for the CY9266–F Evaluation Board is shown in Appendix A, and the schematic for the CY9266–C and CY9266–T Evaluation Boards is shown in Appendix B.

Sheet 1 of the top-level schematic contains four functional blocks, which are detailed on the remaining pages of the schematic.

Sheet 2 contains the power-supply filtering and bypass capacitors. It also contains a sacrificial Zener diode that is used to protect the components on the board in case of over voltage or incorrect connection of the power supply.

Sheet 3 contains the BIST PLD and the error/status displays.

Sheet 4 of Appendix A contains the HOTLink Transmitter and Receiver, as well as the optical interface module. It also contains the on-board oscillator and option-select DIP switch.

Sheet 4 of Appendix B contains the HOTLink Transmitter and Receiver, as well as the copper interface and carrier-detect circuit. It also contains the on-board oscillator and option-select DIP switch.

Sheet 5 contains the parallel interface connectors, the voltage monitor/reset generator, and the OLC-compatibility registers.

Theory of Operation

The CY9266 Evaluation Board operation is broken down into five functional sections:

- Transmitter Parallel Interface
- Transmitter to Optical Module or Copper Serial Interface
- Optical Module or Copper to Receiver Serial Interface
- Receiver Parallel Interface
- BIST and Support Hardware

Transmitter Parallel Interface

The purpose of the transmitter parallel interface is to load parallel data from an external source and move that data to the shifter inside the transmitter. This portion of the design consists of three parts: the transmit data bus, transmitter control signals, and transmitter clocks. A simplified schematic of this interface is shown in *Figure 7*.

Transmit Data Bus

The transmit data bus is composed of the ten signals named XMIT_0 through XMIT_9. This bus may be driven from any of three possible sources: JP2, JP3, or JP4. The data present on this bus is sampled by the HOTLink Transmitter (U1-CY7B923) at the rising edge of CKW.

The information present on the transmit data bus is interpreted by the HOTLink Transmitter in one of two ways, based on the setting of the MODE input

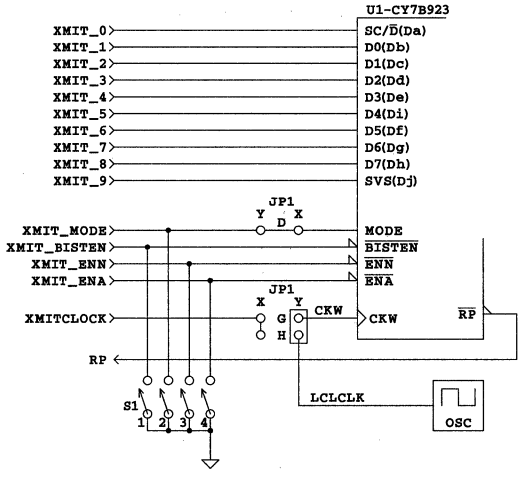


Figure 7. Transmitter Parallel Interface

to the transmitter. When MODE is HIGH (Bypass mode), all ten signals are accepted as the actual data to be transmitted and are fed directly to the shifter. The letter form (Da–Dj, as illustrated in *Figure 7*) of the bit identifiers is followed for this setting. These designators specify which encoded data bit is connected to a specific XMIT_0 to XMIT_9 signal. In this mode the user must encode the data into the 10-bit patterns used to send data across the serial interface. While it is not necessary to use the 8B/10B code described in the HOTLink datasheet, it is advised that this code be used for simplicity. If another code is used, its is the user's responsibility to insure that sufficient transitions are present in the serial data stream to allow the receiver to properly phase-lock to the serial data stream. For the HOTLink Receiver to provide byte framing and synchronization, the K28.5 pattern must be used for framing initialization.

When the MODE input is LOW (Encode mode), the internal 8B/10B encoder is enabled. In this mode the ten input bits are partitioned into eight data bits (D0–D7) and two data-modifier bits (SC/$\overline{D}$ and SVS). For transmitting normal data patterns, both the SVS and SC/$\overline{D}$ pins must be LOW. In this setting the 8-bit data character present on D0–D7 is latched at the rising edge of CKW and presented to the encoder. The encoder then converts the data character into the appropriate 10-bit transmission character. Following conversion, the transmission character is loaded into the shifter.

The two data-modifier bits, SC/$\overline{D}$ (Special Character/Data Select) and SVS (Send Violation Symbol), are used to send transmission characters other than those used to represent data. When the SC/$\overline{D}$ input is HIGH, the normal 8B/10B encoding of the data characters present on D0–D7 is changed. Now special control codes are generated (see listing in the CY7B923/CY7B933 datasheet). These control codes are used to send framing, control, status, and other supervisory functions across the interface.

The SVS pin is used for diagnostic purposes. When this input is HIGH, the HOTLink Transmitter shifter is loaded with a 10-bit pattern that is *not* a valid 8B/10B transmission character. When the HOTLink Receiver detects this encoding violation it responds with its RVS (Received Violation Symbol) output.

Note: The SVS input is intended for diagnostic purposes *only*. If used within normal message traffic, it may cause unexpected receive errors.

Transmitter Control Signals

In addition to the transmit data bus, four other signals are used to control the serial data stream generated by the HOTLink Transmitter. Two of these signals ($\overline{BISTEN}$ and MODE) control operating modes of the transmitter. The other two signals ($\overline{ENN}$ and $\overline{ENA}$) are used to specify when valid data is present on the transmit data bus.

Unlike the transmit data bus, these control signals are not connected to JP4, but are instead connected to JP2, JP3, and separate switches of S1. These switches allow the control inputs to be set LOW or HIGH when an external controller is not present. These switches are used both to control BIST mode for standalone applications and to set the proper operating characteristics for systems which only connect to JP4.

The $\overline{BISTEN}$ and MODE inputs are used to control which transmission characters are generated by the HOTLink Transmitter. Setting $\overline{BISTEN}$ LOW places the HOTLink Transmitter into one of two auto pattern-generation modes.

When $\overline{\text{BISTEN}}$ is LOW and both $\overline{\text{ENN}}$ and $\overline{\text{ENA}}$ are HIGH, the HOTLink Transmitter sends an alternating 1−0 pattern (D10.2 or D21.5). This pattern provides the highest baseband output frequency that the transmitter can generate, and is equal to 5x the frequency of CKW. This pattern may be useful to test or characterize various serial link components (i.e., fiber-optic modules, jitter tests, etc.).

When $\overline{\text{BISTEN}}$ is LOW and either $\overline{\text{ENN}}$ or $\overline{\text{ENA}}$ is also LOW, the HOTLink Transmitter begins a repeating test sequence that allows the transmitter and receiver to work together to test the functionality of the entire serial link. The repeating sequence is 511 characters in length and includes all standard codes as well as patterns that are normally considered code violations. This sequence may also be useful for performing serial link margin tests.

The MODE input pin is used to select both how the data on the transmit data bus is interpreted (encoded or non-encoded) and to place the HOTLink Transmitter into a clock Test mode. This input is capable of selecting one of these three possible modes from a single pin by use of an internal three-level comparator. These modes are

- Encode Mode—S1-2 ON (closed)
- Bypass Mode—S1-2 OFF (open)
- Test Mode—JP1-D, X and Y open

When the MODE input is LOW (Encode mode), the internal 8B/10B encoder is enabled. This allows the transmit data bus to be interpreted as an 8-bit data bus (D0−D7) with two control bits (SC/$\overline{\text{D}}$ and SVS). When the MODE input is HIGH (Bypass mode), the internal encoder is bypassed. This allows the data bus to be interpreted as a 10-bit bus (Da−Dj). Either of these modes may be set from JP2, JP3, or S1-2.

The clock Test mode is accessed by allowing the MODE input pin to float. Through use of an internal bias network in the transmitter, the MODE input pin is placed at $V_{CC}/2$. This clock Test mode can be accessed two ways on the board. The easiest is to cut the foil on the bottom of the board that shorts the X and Y pins of JP1-D together. Once cut it will be necessary to place a shorting jumper across these pins to allow JP2, JP3, or S1 to place the transmitter into one of its normal data modes.

The other method of accessing this mode is to actively bias the XMIT_MODE pin on JP2 or JP3 to $V_{CC}/2$. When doing so, keep in mind that this input also has a 5-kΩ pull-up resistor attached to this signal.

The $\overline{\text{ENN}}$ (enable next parallel data) and $\overline{\text{ENA}}$ (enable parallel data) inputs are normally used to specify when valid data is present on the transmit data bus. Both of these inputs are sampled on the rising edge of CKW at the same time as the 10-bit transmit data bus.

If $\overline{\text{ENA}}$ is LOW and $\overline{\text{ENN}}$ is HIGH at the rising edge of CKW, the data present on the transmit data bus is loaded, processed, and sent to the shifter. If both $\overline{\text{ENA}}$ and $\overline{\text{ENN}}$ are HIGH at the rising edge of CKW, the latched data is ignored and a K28.5 SYNC code is sent in its place.

If $\overline{\text{ENN}}$ is LOW and $\overline{\text{ENA}}$ is HIGH at the rising edge of CKW, the data present on the transmit data bus at the **next** rising edge of CKW is loaded, processed, and sent to the shifter. If both $\overline{\text{ENN}}$ and $\overline{\text{ENA}}$ are HIGH at the rising edge of CKW, the data latched on the next rising edge of CKW is ignored and a K28.5 SYNC code is sent in its place.

These two enable control signals are used to allow different hardware interfaces to be implemented with the least amount (usually none) of additional data pipelining hardware. When one of these enable inputs is used for enable control, the other is usually tied HIGH, but may be used in conjunction with BISTEN for link testing without affecting the data path controller.

Transmitter Clocks

The transmitter interface operates with both an input clock (CKW) and an output clock ($\overline{\text{RP}}$). The input clock is used to generate both the internal shifter clock and the output clock.

The CKW input clock can be sourced from either the on-board oscillator or from the XMITCLOCK signal. This selection is made through jumper block JP1.

All internal operations of the HOTLink Transmitter are based on the rising edge of the CKW clock. The

CKW clock must be generated from a crystal-based source. While the duty cycle of the CKW clock source is relatively unimportant, it must still meet certain minimum pulsewidth times as listed in the CY7B923/CY7B933 datasheet.

The $\overline{RP}$ output clock pulse is a modified duty cycle pulse whose HIGH and LOW components are set for operation with asynchronous FIFOs (CY7C42X family). The phase relationship of this clock pulse to CKW, and its duty cycle (both set by the internal PLL), are positioned to have valid data on the transmit data bus at the rising edge of CKW.

This $\overline{RP}$ clock pulse may be directly connected to the read control pin ($\overline{R}$) of an attached FIFO. Because the presence of this pulse signifies a FIFO read operation, it is only generated in response to the $\overline{ENA}$ input being pulled LOW.

Transmitter to Optical Module Serial Interface

The transmitter has three differential output pairs that each output the same serial data stream from the shifter. Because of the switching speeds used for these serial outputs (and for compatibility with optical interface modules) they are all implemented using positive-referenced 100K ECL-compatible drivers. A simplified schematic of the interface present on the CY9266–F is shown in *Figure 8*.

The normal mode of ECL operation is for all signaling to be done at voltages below ground. Because the ground point for ECL is only a reference, the same signaling can also be implemented above ground. When this is done the reference point changes from ground to V_{CC}. When operated in this mode ECL is often referred to as PECL (positive-ECL). This is the mode of operation for the serial outputs on the transmitter.

Two of the differential outputs (OUTA± and OUTB±) are also controlled by a TTL-level enable pin called FOTO (fiber-optic transmitter-off). This control input is used to disable all light output from the optical module. While not specifically necessary for LED-based optical modules, the ability to disable all light output is a safety requirement for all laser-based links (ANSI Z136.1 and Z136.2, F.D.A. regulation 21 CFR subchapter J, and IEC 825).

When FOTO is HIGH, the OUTA± and OUTB± differential pairs are forced to a logic 0 state (OUT+ is LOW and OUT– is HIGH). When FOTO is LOW, the OUTA± and OUTB± differential outputs are allowed to follow the serial data pattern from the shifter.

The FOTO pin on the HOTLink Transmitter may be configured to be controlled from either the JP2, JP3, or JP4 connectors (LINK_CONTROL) or from S1-8 (DIP_FOTO). To avoid possible signal contention from these sources, this signal is first run through jumper block JP1.

Placing a shorting jumper across the X and Y pins of JP1-F allows the transmitter FOTO pin to be controlled from the LINK_CONTROL signal. Moving this jumper to JP1-E allows this selection to be made through S1-8 or through the DIP_FOTO signal on JP2 and JP3. If the jumper is omitted from the board, the OUTA± and OUTB± outputs are placed in the disabled state.

The OUTC± differential output is not controlled by FOTO. This output continues to follow the serial shifter data at all times. Because it is never disabled, this signal is used for the local loopback. While this signal is available differentially, it is connected to

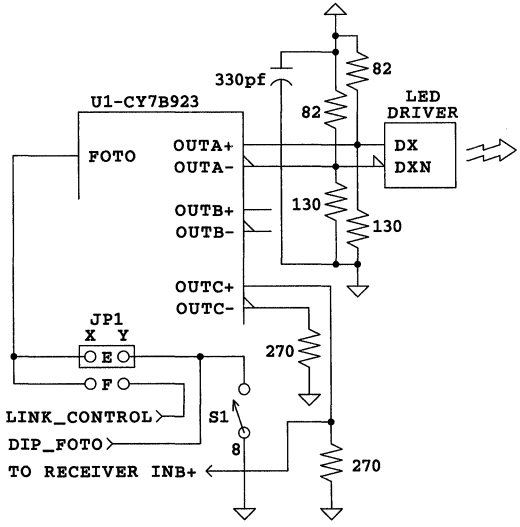


Figure 8. HOTLink Transmitter-to-Optical Serial Interface

the receiver single-ended. This allows the INB− input on the receiver to be used as an ECL-to-TTL translator for the receive optical module's carrier-detect signal.

Because ECL signals are only *active* in one direction, it is necessary to provide a bias/load network of some type for the signals to properly switch. The typically specified load for ECL signals is 50Ω connected to V_{CC} − 2V (i.e., +3V for PECL).

This type of load can be created in many ways. For large ECL systems a separate power supply is usually present to generate this bias voltage. This provides the lowest power dissipation. For small systems (like this one), a simpler method is to use two resistors to create a network whose Thévenin equivalent is this same 50Ω connected to V_{CC} − 2V. This is used for the OUTA± differential pair. The capacitor present across the Thévenin pair is necessary to produce an AC short between the power and ground planes.

The OUTB± output pair is not used on this evaluation board. While normal ECL drivers left in this mode would still dissipate a significant amount of power, the HOTLink ECL outputs contain additional internal structures to sense if an output is used or left open, and disables the internal current sources of unused output drivers. This results in a current savings of approximately 5 mA (25 mW) for each unused output pair.

The OUTC± output pair is biased to V_{CC} − 5V (ground) through 270Ω resistors. This bias arrangement is used here to reduce the overall component count. This type of load may be used for short connections because it provides a similar current load to a Thévenin termination but, due to asymmetric rise and fall times, it induces more jitter into the data. This type of biasing should not be considered as a type of line termination. If the switching speeds and length of circuit traces dictate that the line should be terminated, a Thévenin bias network should be used to match the line impedance.

Even in those cases where the connection to the optical modules is short and a 270Ω resistor to V_{CC} − 5V may seem to be usable, it should not be used. While this type of connection may work for very short optical cable lengths, the jitter introduced by the bias network reduces the overall system jitter margin.

Transmitter to Copper Cable Serial Interface

On the CY9266−C and CY9266−T boards, the transmitter output is configured to drive either a coaxial or shielded-pair cable. A simplified schematic of this interface is shown in *Figure 9*.

The copper-based CY9266−C and CY9266−T boards use a transformer-coupled interface. Transformer coupling is called out in the ANSI Fibre Channel standard for copper-based interfaces. Its primary advantages are excellent common mode rejection, balanced-to-unbalanced conversion (for coaxial cables), and DC isolation (2 kV hi-pot tested).

The CY9266−C and CY9266−T boards are designed to allow other modes of line biasing and coupling to be used for presenting a signal into the cable. Pads are present on the board to allow a Thévenin bias to be used on OUTA±. These resistors are identified as R72 and R73 on Sheet 4 of the CY9266−C/T schematic (see Appendix B).

The CY9266−C and CY9266−T are designed to operate with cable systems providing a reflection coefficient of zero. This means that the receiving end

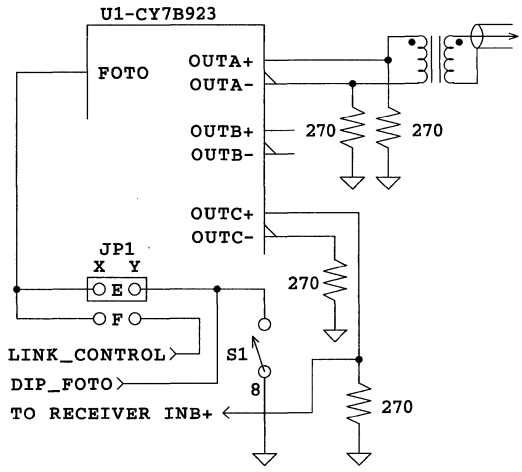


Figure 9. HOTLink Transmitter to Copper Serial Interface

of the cable should be terminated in the characteristic impedance of the cable.

Pads are also present to allow both source termination and AC coupling to the transformer. These components are identified as R54, R55, C25, and C26 on Sheet 4 of the CY9266–C/T schematic (see Appendix B). To use parts in these locations it is necessary to remove the foil shorts across these component pads on the circuit board.

The control signal inputs for copper-based interfaces operate identically to those of the optical interface. The difference in operation is that when the OUTA± outputs are disabled through the use of the FOTO signal, instead of disabling all light, all output transitions are disabled.

Optical Module to Receiver Serial Interface

The HOTLink Receiver has two differential input pairs (INA± and INB±) that can both be used to receive the high-speed serial data streams generated directly by the transmitter or as output from an optical receiver. These serial inputs are also PECL and are directly compatible with the HOTLink Transmitter. ECL was chosen for these signals for the same reasons (speed, low noise, compatibility with optical modules) it was used for the transmitter.

A separate PECL input signal ($A/\overline{B}$) is used to select which input pair (INA± or INB±) is actually fed to the receiver shifter and PLL. A simplified schematic of the optical module-to-receiver serial interface on the CY9266–F is shown in *Figure 10*.

Optical Module Signals

The optical receiver generates two signals; a 100K PECL differential received data signal, and a single-ended carrier-detect signal. While the DIP package form of the optical module does provide both + and – forms of the carrier-detect signal, only the + form is available on the endfire package. To allow the same circuitry to be used with either module type, only the + carrier-detect signal is used.

Receiver Data Inputs

The HOTLink Receiver differential INA and INB inputs are similar, but not identical. While the INA± inputs must always operate as a differential pair, the INB± signals do not. This allows the INB± inputs to be split into two separate ECL inputs: INB+, which feeds the shifter and PLL, and INB–, which feeds an ECL-to-TTL translator.

The configuration of the INB± inputs is controlled by the SO output of the translator. While technically an output, the SO pin on the HOTLink Receiver also contains sense circuits that monitor the voltage level on the pin during power-up. If the SO output is connected to V_{CC}, the INB– input becomes part of the INB± differential serial input. If the SO output is normally loaded (no resistive pull-up to V_{CC}), the INB+ input becomes a single-ended serial data receiver and the INB– input becomes part of a PECL-to-TTL translator.

This split mode is used on the CY9266 Evaluation Board. It allows the INB– input to be used to convert the PECL carrier-detect output of the optical module (SIGO) to the TTL-level signal needed on the receiver parallel interface.

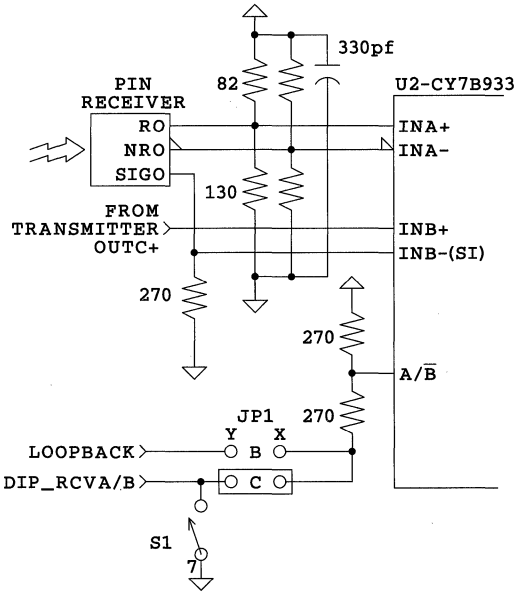


Figure 10. Optical-to-HOTLink Receiver Serial Interface

Receiver Port Select

The HOTLink Receiver uses a single-ended PECL input (A/$\overline{B}$) to control which serial input is fed to the shifter and PLL. When the A/$\overline{B}$ input is HIGH, the differential INA± pair is connected to the shifter and PLL. When the A/$\overline{B}$ input is LOW, the INB+ input is fed to the shifter and PLL. Because the INB+ input is directly connected to the OUTC+ output from the HOTLink Transmitter, this LOW setting is used for a local loopback and allows the transmitter and receiver to communicate without using an optical module.

The A/$\overline{B}$ input is a PECL input and normal TTL or CMOS logic swings will not work to control it. This input uses PECL (or larger) signal swings. These can still be achieved in a TTL environment through use of a resistive divider network.

Using this network, a TTL LOW level on the input to the divider creates a PECL LOW at the A/$\overline{B}$ input to the receiver. With a TTL (or CMOS) HIGH into the divider, the A/$\overline{B}$ input is placed at (or above) a PECL HIGH. While standard 100K ECL inputs should never be taken above V_{CC} – 700 mV, the ECL inputs on the HOTLink Receiver may be connected directly to V_{CC} without degradation or damage.

The divider network on this evaluation board may be configured to be controlled from either the JP4 connector (LOOP_BACK) or from S1-7 (DIP_RCVA/B). To avoid possible signal contention from these sources the signal is first run through jumper block JP1.

Placing a shorting jumper across the X and Y pins of JP1-B allows the receiver port selection to be controlled from the LOOP_BACK signal. Moving this jumper to JP1-C allows this selection to be made through S1-7 or through the DIP_RCVA/B signal on JP2 and JP3. If the jumper is left off the board, the A± pair is selected.

Copper to Receiver Serial Interface

The CY9266–C and CY9266–T Evaluation Boards replace the optical module with a transformer coupled electrical interface. The transformer

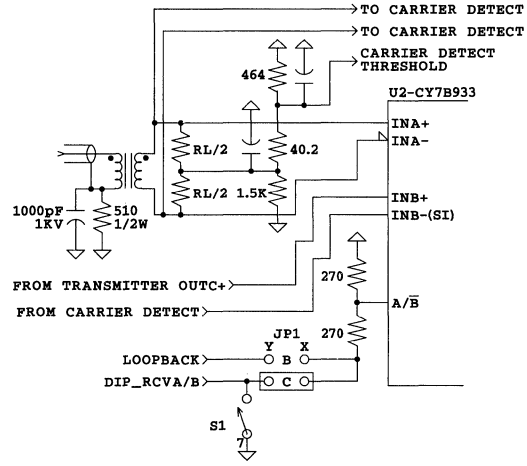


Figure 11. Copper-to-HOTLink Receiver Serial Interface

used here provides the same functionality as the one used at the transmit end of the cable. A simplified schematic of the copper cable-to-receiver serial interface on the CY9266–C/T is shown in *Figure 11*.

The output side of the transformer connects to two resistors. These resistors provide the line termination for the transmission line connected to the transformer. Two resistors are used for the termination network to allow a reference voltage to be set for the center of the received signal. This reference point is set by an external 3-resistor divider, and is set in this circuit to V_{CC} – 1.3V. This is near the center of the common mode range of the MC10H116 ECL receiver that is used to build a carrier detection circuit. If this carrier-detect circuit is not used, it would be better to bias this point at V_{CC} – 1.5V, the center of the HOTLink Receiver's common mode range.

Both of these reference points must be bypassed to allow them to remain stable under dynamic signal conditions.

Unlike the optical receiver, which outputs a logic zero in the absence of light (INA+ = 0, INA– = 1), the AC-coupled interface used for copper connections does not. When the signal is removed, the INA+ and INA– inputs to the HOTLink Receiver are set to the same voltage. Because of the high gain present in the HOTLink Receiver to allow use with

long cables (low amplitude received data), the HOTLink Receiver will probably oscillate. This oscillation under a no-signal condition can be corrected by forcing an offset between the INA+ and INA− inputs, but this offset will induce more jitter into the data stream and limit the usable length of a copper-based serial link. Rather than compromise operational length, a carrier detection circuit can be added to validate the received data (in addition to the validation mechanisms present in the data itself).

The CY9266−C and CY9266−T boards also contain the pads and routing necessary for implementing an equalizer to allow longer cables to be used. The function of an equalizer is to present a frequency selective attenuation to the received signal that brings the amplitude and phase of the frequency components in that signal into the same amplitude and phase. Because signals transmitted over copper cables are effectively run through a high-frequency attenuator, the equalizer used for copper cables is a form of low-frequency attenuator (high-pass filter).

The equalizer used is implemented in a bridged-H configuration that is designed for balanced line operation. It is shown on Sheet 4 of the CY9266−C schematic in Appendix B and is constructed using R64, R65, R66, R67, R68, R69, R70, R71, C29, C30, and L1. To implement this equalizer it is necessary to remove the foil shorts across R64 and R71.

Copper Carrier-Detect

The carrier-detect circuit used on the CY9266−C and CY9266−T boards is shown in *Figure 12*. This circuit uses two ECL differential receivers as level comparators to detect the presence of 1- and 0-level pulses on the incoming signal. The gate connected to the top side of the transformer (shown in *Figure 11*) detects the presence of received 1 pulses while the gate connected to the bottom of this transformer detects the presence of received 0 pulses. The input capacitance of these comparators is isolated from the actual received signal through 100Ω resistors to prevent this additional load from distorting the received signal.

The input signal amplitude necessary to detect either a 1 or a 0 is set by the resistor divider shown in *Figure 11*. To prevent the 10H116 gate from oscillating it is recommended that this threshold be set to a minimum of 50 mV above the termination reference voltage.

The outputs of these two gates are then wire-ORed together to charge a capacitor. Because of the low on resistance of the emitter follower output transistors of the 10H116 gates, the capacitor can be charged quite quickly. In the absence of 1 or 0 transitions above the set threshold level, this capacitor is discharged both by a bleeder resistor to V_{EE}, and through the input of the third gate.

The third gate is configured as a comparator with feedback to form a Schmitt trigger. This feedback is necessary because of the slow transition rate of the input signal to this gate. If feedback was not used, this gate would oscillate as the input signal slowly passes through the the threshold region of the gate. The output of this Schmitt trigger is then connected to the HOTLink Receiver INB− input, which is configured as a PECL-to-TTL translator.

Receiver Parallel Interface

The receiver parallel interface is used to move the character framed in the HOTLink Receiver to the external world where it can be used. This portion of the design consists of five sections: receiver parallel data output, OLC-compatibility registers, receiver clocks, receiver control inputs, and receiver status outputs. A simplified schematic of this interface is shown in *Figure 13*.

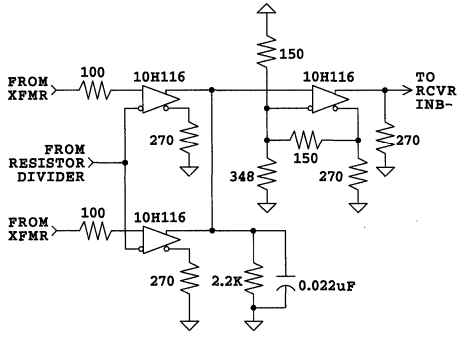


Figure 12. Copper Interface Carrier-Detect

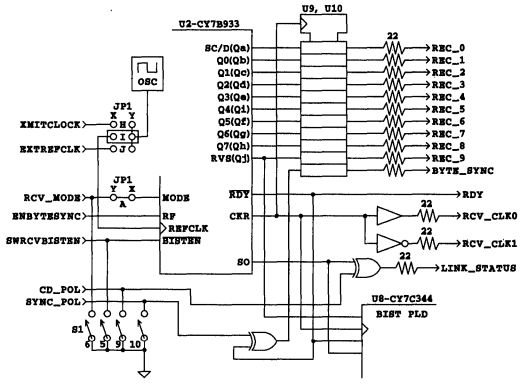


Figure 13. HOTLink Receiver Parallel Interface

Receiver Parallel Data Output

The receiver data bus is composed of ten signals named REC_0 through REC_9. This bus drives all three I/O connectors (JP2, JP3, and JP4). Due to the external register in the data path, these outputs change coincidental with the rising edge of RCV_CLK0 (CKR).

The information placed on the receiver data bus is determined by the HOTLink Receiver MODE select pin. When MODE is HIGH (Bypass mode), all ten outputs are the ten bits that were received and framed. The letter form (Qa−Qj, as illustrated in *Figure 13*) of the bit identifiers is followed for this setting. These designators specify which encoded data bit is connected to a specific REC_0 to REC_9 signal. In this mode the user must decode the data from the 10-bit patterns used to send the data across the serial interface.

While it is not necessary to use the 8B/10B code described in the HOTLink datasheet, it is advised that this code be used for simplicity. If another code is used, it is the user's responsibility to insure that sufficient transitions are present in the data stream to allow the HOTLink Receiver to properly phase-lock to the serial data stream.

For the HOTLink Receiver to maintain byte framing and synchronization, the K28.5 pattern must also be used for framing initialization. For those systems that perform their own framing (SONET, etc.), the HOTLink Receiver will phase-lock to a serial data stream without a K28.5 code present and clock out a character every 10 bit-clocks. These systems must operate in Bypass mode as the HOTLink Receiver decoder requires operation with the 8B/10B code and must acquire byte sync to recover valid data. These systems must provide external byte framing.

When the HOTLink Receiver MODE input is LOW, the internal 10B/8B decoder is enabled. In this mode, the ten output bits from the shifter are sent to the decode register once every ten bit-clocks, as determined by the framer. The 8-bit output from this decoder is then placed on the receiver output data bus bits Q0−Q7, along with the two data status bits SC/$\overline{D}$ and RVS.

When receiving normal data patterns both the RVS and SC/$\overline{D}$ pins are LOW. In this setting, the 8-bit data character present on Q0−Q7 is latched at the rising edge of CKR into the external register and presented to the output of the board.

The two status bits, SC/$\overline{D}$ (special character/data select) and RVS (received violation symbol), are used to indicate reception of characters other than those used to represent data. When the SC/$\overline{D}$ output is HIGH, special control codes (see listing in the CY7B923/CY7B933 datasheet) have been decoded. These control codes are used to indicate framing, control, status, and other supervisory functions across the interface.

The RVS pin is used for diagnostic purposes. When this output is HIGH, the HOTLink Receiver decoder has detected a 10-bit pattern that is **not** a valid 8B/10B transmission character or sequence. When the receiver detects this encoding violation, it asserts RVS and places information on the Q0−Q7 outputs to represent the type of error detected. Because all of these errors are represented with special codes (C0.7, C1.7, C2.7, and C4.7) the SC/$\overline{D}$ output is always HIGH whenever RVS is HIGH. These possible error-type codes are listed in the HOTLink datasheet.

OLC-Compatibility Registers

In order for this evaluation board to operate in an OLC-266 compatible system, the timing of the $\overline{RDY}$

signal had to be modified. This signal from the receiver is used for four functions: to indicate when a K28.5 SYNC character has been received, to indicate that valid data has been received, to clock valid data into an external asynchronous FIFO, and to indicate the end of a BIST loop.

To support these different functions from a single pin requires the addition of a single register to convert the waveform generated by the $\overline{RDY}$ signal into the BYTE_SYNC status signal the OLC card generates. Additional registers were then added to the data bits to keep them in the same byte-phase relationship as the BYTE_SYNC signal (which is now delayed one clock).

The 22Ω series termination present on these signals should not be necessary for most systems, but are added here to allow a flat-cable-type attachment to this card.

Figure 14 shows the relative timing relationships between the HOTLink Receiver data, the $\overline{RDY}$ signal, the BYTE_SYNC signal, and the output clocks. For $\overline{RDY}$ to operate in this fashion, the RF (Reframe enable) control input must be HIGH and the receiver must be in Bypass mode (receiver MODE is HIGH).

When RF is LOW, the $\overline{RDY}$ and BYTE_SYNC outputs operate the same as that shown in *Figure 14*. The difference is that the clocks are not allowed to change phase or width upon detection of a K28.5 SYNC character.

The functionality of the $\overline{RDY}$ (and thus BYTE_SYNC) signal changes when the receiver is in Decode mode (receiver MODE is LOW). Here the the $\overline{RDY}$ signal pulses LOW for every character received *including* the K28.5 SYNC character. When multiple consecutive SYNC characters are received, $\overline{RDY}$ is inhibited except for the last K28.5 character received. This is done to prevent overfilling a receiver FIFO with non-data information. *Figure 15* shows the relative timing relationships for this type of operation.

Because RF is LOW in *Figure 15*, the CKR clock (and thus RCV_CLK0 and RCV_CLK1) is not allowed to reframe on new K28.5 SYNC characters detected. When RF is HIGH in Decode mode, the HOTLink Receiver $\overline{RDY}$ output ceases pulsing until the first K28.5 SYNC code is detected, after which the behavior illustrated in *Figure 15* is resumed.

Receiver Clocks

The HOTLink Receiver parallel interface (see *Figure 13*) operates with a single input clock (REFCLK) and two output clocks (CKR and $\overline{RDY}$).

The REFCLK input clock does not directly clock anything in the receiver, but is used as a reference for the receiver PLL. This clock is required to be both stable and reasonably accurate. It must match the byte-rate frequency of the received data within ±0.1%. Unlike an OLC card, which requires a special sequencing of the LOCK_TO_REF signal to allow the receiver to track to a reference clock, the HOTLink Receiver PLL continuously operates in a mode that compares its frequency to that of the reference clock, even when valid data is being received.

If the frequency of the received data varies outside of specific fixed limits, the HOTLink Receiver stops

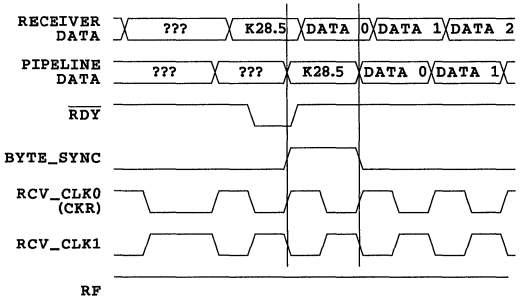


Figure 14. Receiver Data Timing, Bypass Mode, RF HIGH

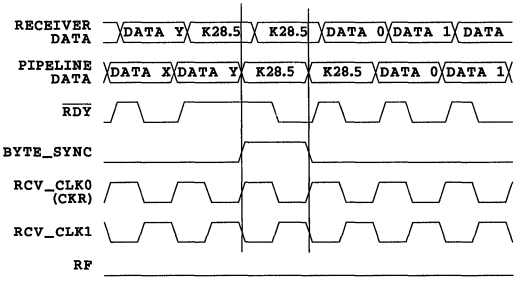


Figure 15. Receiver Data Timing, Decode Mode, RF LOW

locking to the serial data and reverts to the REFCLK. Once the received serial data stream returns to an acceptable frequency, the PLL again locks to the received data. Since it is likely that byte sync has been lost, a reframe cycle should be performed to allow the framer to lock up again. Detection of this and the recovery process is normally handled automatically by higher-level functions in the communications system.

The REFCLK input to the receiver can be sourced from three different signals on the evaluation board: the on-board oscillator, the XMITCLOCK input, or the EXTREFCLK input. Selection of the clock source can only be done through jumper block JP1.

The on-board oscillator is used primarily for standalone operation and testing using the BIST capabilities of the HOTLink parts. This clock is selected by placing a shorting jumper across pins X and Y of JP1-I.

The XMITCLOCK input is used for normal data transmit/receive functions and for OLC-compatibility mode. This clock is selected by placing a shorting jumper across pins JP1-HX and JP1-IX.

The EXTREFCLK input is used for those instances when the transmitter and receiver are to be clocked with different frequency clocks. This is expected to be used only to test for PLL capture/lock range testing of the receiver, or when the HOTLink Receiver is connected to a transmitter operating at a different frequency from the local HOTLink Transmitter. This clock is selected by placing a shorting jumper across pins JP1-JX and JP1-IX.

The CKR output clock is generated in the HOTLink Receiver and is based directly on the internal PLL frequency. This output is synchronous with the receiver output data bus and may be used to clock the data into an associated register (as is done on this board) or into synchronous FIFOs.

The period and duty cycle of the CKR output clock are fixed by the logic in the receiver. To achieve compatibility with OLC-type systems, the CKR signal is used to generate two new clock signals (RCV_CLK0 and RCV_CLK1) that are true and complement copies of the CKR clock. To keep matched delays and to minimize the number of additional logic packages on the board, these two clocks are generated using XOR gates.

When framing occurs, the CKR clock can experience large phase changes. These changes are exhibited by a lengthening of either the HIGH or LOW portion of the CKR waveform. This can be seen in the waveforms shown in *Figure 14*. While this functionality is not required by the ANSI Fibre Channel Standard, it is included in the HOTLink Receiver to protect downstream clocked logic from the narrow pulses or glitches that can occur otherwise.

The $\overline{\text{RDY}}$ output signal is used both as a status output and as a clock. Its use as a clock is primarily for clocking data present on the receiver data bus outputs into asynchronous FIFOs. The duty cycle of the $\overline{\text{RDY}}$ pulse and its position relative to the output data is such that it may be directly connected to the $\overline{\text{W}}$ (write) input on CY7C42X FIFOs.

Receiver Control Inputs

The receiver parallel interface is controlled by three input signals: RF (Reframe), MODE (Receiver Mode select), and $\overline{\text{BISTEN}}$ (BIST Enable).

The RF input is used to select when the HOTLink Receiver is allowed to reframe (acquire byte-sync) to the incoming serial data stream. This input is present to prevent the receiver from mis-framing on aliased K28.5 SYNC codes, which would cause long running decode errors.

When RF is LOW the framer is disabled; it does not change the starting bit location of each received character. Any received K28.5 SYNC code is treated as normal data and is clocked out with the CKR and $\overline{\text{RDY}}$ clocks. If this SYNC code is received across two character boundaries, the framer does not reframe. If the HOTLink Receiver is operating in Decode mode, the existence of such a non-aligned pattern may generate one or more characters in error.

When RF rises, the $\overline{\text{RDY}}$ output is inhibited. With RF held HIGH, the framer continuously monitors the serial data stream for either disparity form of the K28.5 SYNC character. When this character is detected, the bit counter used to count off serial

data bits and specify received character boundaries is asynchronously reset to properly frame the subsequently received bits on character boundaries.

If the receiver is set to Decode mode, the $\overline{RDY}$ output assumes its normal function of pulsing LOW for each byte after the first K28.5 SYNC code is detected. If the receiver is instead set to Bypass mode, the $\overline{RDY}$ signal pulses LOW only for the SYNC (K28.5) characters while RF is HIGH or LOW.

Because of characteristics of the 8B/10B code, it is possible to transmit legal character sequences that can cause incorrect framing (this requires sending control codes other than K28.5). These codes should be avoided while RF is HIGH. Once the framer is disabled (RF LOW) these sequences may be used to pass control information across the interface without causing the receiver to incorrectly frame the data that follows.

The MODE input pin on the HOTLink Receiver is used to select both how the received serial data is to be presented on the data bus (encoded 10-bit character or decoded 8-bit character), and to place the receiver into a clock Test mode. This input is capable of selecting one of these three possible modes from a single pin through use of an internal three-level comparator.

When the MODE input is LOW, the internal 10B/8B decoder is enabled (Decode mode). This allows the receiver output data bus to be interpreted as an 8-bit data bus (Q0–Q7) with two status bits (SC/$\overline{D}$ and RVS). When the MODE input is HIGH, the internal decoder is bypassed (Bypass mode). This allows the data bus to be interpreted as a 10-bit bus (Qa–Qj). Either of these modes may be set from JP2, JP3, or S1-6.

The clock Test mode is accessed by allowing the MODE input pin to float. Through use of an internal bias network in the receiver, the MODE input pin is placed at $V_{CC}/2$. This clock Test mode can be accessed two ways on the board. The easiest is to cut the foil on the bottom of the board that shorts the X and Y pins of JP1-A together. Following this, it will be necessary to place a shorting jumper across these pins to allow JP2, JP3, or S1-6 to place the receiver into one of its normal data modes.

The other method of accessing this mode is to actively bias the RCV_MODE pin on JP2 or JP3 to $V_{CC}/2$. When doing so, keep in mind that this input also has a 5-kΩ pull-up resistor attached to the signal.

The $\overline{BISTEN}$ input pin is used to place the HOTLink Receiver in a special pattern verification mode. This mode is designed to work in conjunction with a matching pattern generation mode in the transmitter. While not shown on the schematic in *Figure 13*, the $\overline{BISTEN}$ input is actually run through the BIST PLD (U8-CY7C344). This is not necessary but is done here to allow other conditioning of the $\overline{BISTEN}$ signal if desired.

When the HOTLink Receiver $\overline{BISTEN}$ input is set LOW, the receiver's BIST state machine is enabled and enters its self-test mode. At this point it sets $\overline{RDY}$ HIGH and begins looking for the BIST start-of-loop character (D0.0) in the serial data stream. Once this character is detected, the $\overline{RDY}$ output is driven LOW, where it remains until the end of the 511-character BIST loop. At this point $\overline{RDY}$ pulses HIGH for one character and starts the next 511-byte loop.

While BIST mode is enabled, the RVS output is used to indicate that a pattern mismatch has occurred. This means that the 10-bit pattern received did not *exactly* match the 10-bit pattern that was expected (expected code violations are not errors).

Receiver Status Outputs

The HOTLink Receiver parallel interface generates two status output signals: $\overline{RDY}$ and SO.

The $\overline{RDY}$ output is used both for status information and as a clock. As a status output, its information is valid at the rising edge of CKR. This means that the $\overline{RDY}$ signal must be registered to present its status information. For normal data transfer modes, the registered form of $\overline{RDY}$ is used to identify the presence of multiple K28.5 SYNC characters (HIGH at rising edge of CKR) and of data or control characters (LOW at the rising edge of CKR). This registered form of $\overline{RDY}$ generates the BYTE_SYNC signal.

The $\overline{RDY}$ signal is also used to identify what phase the HOTLink Receiver BIST mode is in. When

HIGH for two or more CKR clocks, the receiver is looking for the start character of the BIST loop. When LOW, the receiver is in the BIST loop. When HIGH for a single clock, the receiver has completed another BIST loop.

The SO output is used as part of an ECL-to-TTL translator to specify the current state of the carrier on the serial interface, and is used to drive the LINK_STATUS signal. When a valid carrier is present and S1-9 (CD_POL) is off (open), LINK_STATUS is LOW. This polarity is reversed by turning S1-9 on (closed) or pulling SYNC_POL LOW.

BIST and Support Hardware

The CY9266 Evaluation Board contains not only those components necessary to form a serial link, but also a few support components to enhance OLC compatibility and to support the BIST capability in the HOTLink Transmitter and Receiver. A simplified schematic of these additional components is shown in *Figure 16*.

The MAX707 is used to monitor the power-supply voltage and remove the RESET signal when V_{CC} is above 4.65V. This is a close approximation to the 4.75V RESET threshold specified for the OLC card. This part also supports an external mechanical switch input that also controls the RESET output. This input is controlled by the BIST reset push-button switch (S2). When this switch is depressed, the RESET output is driven LOW until 200 ms after the switch is released. This RESET signal is used to clear the BIST error-counter located in the BIST PLD (U8). The PWR ON indicator is extinguished as long as RESET is active.

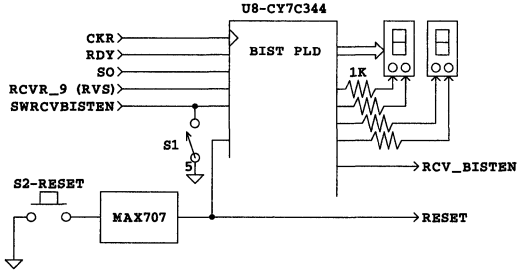


Figure 16. BIST Support Hardware

The BIST PLD is a Cypress CY7C344 MAX EPLD programmed with the counters and state machines necessary to monitor the status of the receiver outputs and count when BIST-compare errors are detected. This PLD also drives the decimal points on the attached displays to indicate four status signals. These status signals are:

- PWR ON—Lit when power is present and above the 4.65V sense threshold
- CAR DET—Lit when a valid carrier is present
- BIST WAIT—Lit when BIST is enabled but the receiver has not detected the start of the BIST loop
- BIST OVFL—Lit when the BIST error count exceeds 99

BIST State Machine

The BIST state machine has six states that control when a counter is enabled to count pattern-match errors. A bubble diagram of this state machine is shown in *Figure 17* while the MAX+PLUS source file for this state machine is listed in Appendix C.

This state machine controls when the error counter is enabled to count. It operates off of two input signals: BISTEN and $\overline{\text{RDY}}$. Whenever BISTEN is not present, the machine is returned to the WAIT0 state (while all state transition arrows are shown for these transitions, not all of them are labeled).

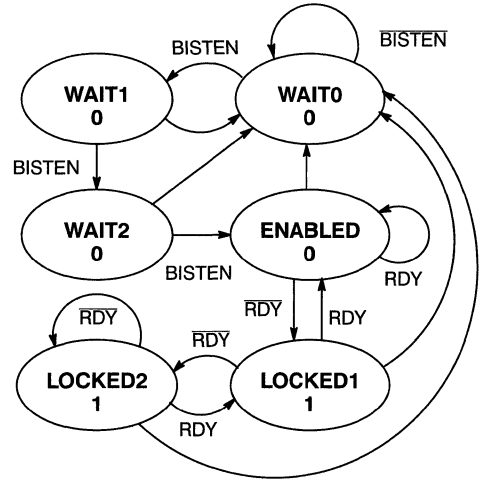


Figure 17. BIST State Machine Bubble Diagram

Once BISTEN becomes active, the machine goes through two secondary wait states (WAIT1 and WAIT2) before starting to look for $\overline{RDY}$ being active. These wait states are necessary to allow the receiver time to recognize the BISTEN signal and bring $\overline{RDY}$ high.

When the ENABLED state is reached, the machine remains in this state until $\overline{RDY}$ goes LOW, causing the machine to move to the first of the two LOCKED states. This signifies that the receiver has received the start-of-loop character (D0.0) and is now performing matching of the received data bits to its internal pattern generator.

In the LOCKED states, the external counter is enabled to count errors. The reason two LOCKED states are present is to allow for the single pulse on $\overline{RDY}$ that indicates the end of a BIST loop. If $\overline{RDY}$ is ever HIGH for more than one clock, the HOTLink Receiver has determined that it is no longer in sync with the transmitter and it starts looking for the start-of-loop character again.

Other BIST PLD Functions

The complete schematic for the BIST PLD is shown in Appendix C. Other than the BIST state machine, the other main logic functions present in the part are for driving the four status indicators and the actual error counter.

Error Display

The error display is made from two hexadecimal LED displays (TIL311). These displays are each capable of showing the entire hexadecimal character set (0–9, A–F) as well as having two independent decimal points. These decimal points are used as individual status indicators for the board.

External Serial Interface Connections

The primary difference between the CY9266 card types is in the external high-speed serial interface. Each of the card types operates with not only a different media type (optical, coaxial, shielded twisted pair), but also different connectors and cable types.

CY9266–F Serial Interface Connections

The CY9266--F HOTLink Evaluation Board implements a fiber-optic-based serial interface. This interface uses industry-standard LED-based fiber-optic modules that accept SC-type fiber-optic connectors.

Optical Modules

The CY9266–F HOTLink Evaluation Board is designed to operate using a *de facto* standard-footprint optical module. Any optical module meeting the pinout and dimensions of this *de facto* standard (established originally for FDDI) should operate with the CY9266–F.

Note: These standard-footprint optical modules are available in a wide range of operating data rates. Because the operating data rate for some of these modules may be outside the 160- to 330-Mbit/second operating range of the HOTLink Transmitter and Receiver, care should be exercised when selecting an optical module.

This footprint supports two types of optical modules: those with four rows of vertical pins, and those with a single row of pins along the bottom edge. In vendor literature these are referred to as DIP- and endfire-type packages.

While specified originally for FDDI, modules meeting this footprint are also available for Fibre Channel and ATM data rates, and meet all optical and mechanical specifications of the Fibre Channel Standard. *Figure 18* shows the mechanical footprint dimensions of this *de facto* standard package.

Both package types operate from a +5V supply and interface directly with 100K ECL/PECL. The biggest mechanical difference between them is that the endfire-type packages have two oversized pins (1 and 32) that are used only to hold the package in place. The main electrical difference between the packages types is that the DIP package drives the Signal Detect output differentially while the endfire package only provides the active HIGH output. *Table 5* lists the pinouts for this standard-footprint optical module.

The active signals listed in *Table 5* are

- SD—Signal Detect

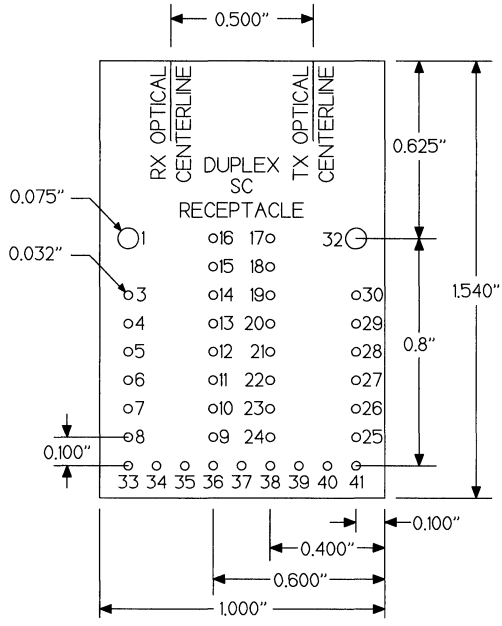


Figure 18. Optical Module, Top View Dimensions

- TD—Transmit Data
- RD—Receive Data
- Case—Outer Case of Module
- V_{CC}—Positive Supply Voltage
- V_{EE}—Negative Supply Voltage

Pins marked "Case" are not necessarily isolated pins. Because the optical module is used in the CY9266–F in a PECL mode, these Case pins are connected to the V_{EE} (ground) supply. When selecting an optical module, care should be taken to insure that the pins marked "Case" are either floating or are attached to the appropriate power supply rail.

To allow evaluation of different types of optical modules, the CY9266–F Evaluation Board is built using low-profile socket pins for the optical module. This allows the modules to be easily replaced. In addition, two slotted holes are provided for a cable-tie to hold the module in place.

Table 5. Optical Module Pinout

DIP Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
1	Case	2	No Pin
3	Case	4	V_{EE}
5	V_{EE}	6	+SD
7	–SD	8	Case
9	Case	10	–RD
11	+RD	12	V_{CC}
13	V_{CC}	14	V_{CC}
15	Case	16	Case
17	Case	18	Case
19	V_{CC}	20	V_{CC}
21	Case	22	+TD
23	–TD	24	Case
25	Case	26	V_{BB}
27	Case	28	Case
29	V_{EE}	30	V_{EE}
31	No Pin	32	Case
Endfire Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
33	V_{EE}	34	+RD
35	–RD	36	+SD
37	V_{CC}	38	V_{CC}
39	–TD	40	+TD
41	V_{EE}		

Fiber-Optic Connector

The optical modules specified for use on the CY9266–F HOTLink Evaluation Board (listed in Appendix A, item U4) are designed to accept SC-type fiber-optic connectors. These connectors are available in both simplex (single-fiber) and duplex (dual-fiber) versions. *Figure 19* shows a simplex SC fiber-optic connector. A duplex connector is formed either by joining two simplex connectors together with a clip (sometimes referred to as a "Z" clip) or by using a connector that supports two fibers in the same form factor. The standard optical fiber type used with these connectors and LED-based op-

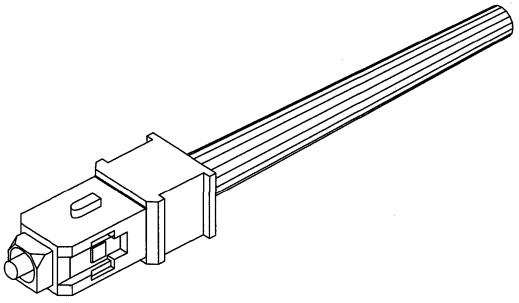


Figure 19. SC Simplex Fiber-Optic Connector

tical modules is 62.5/125-µm multimode graded-index fiber.

When using duplex connector cables, the cable construction controls which fiber is connected to the transmit LED and which is connected to the receive photodetector. When using simplex cables, this polarization control is left to the user. The transmit and receive connectors on the fiber-optic module are shown in *Figure 20.*

CY9266–C Serial Interface Connections

The CY9266–C HOTLink Evaluation Board implements a copper-based serial interface. This interface uses 75Ω coaxial cables having BNC- and TNC-type connectors.

Coaxial Board Connectors

The CY9266–C HOTLink Evaluation Board has two right-angle female coaxial cable connectors: a BNC (Bayonet Neil-Councilman) for the J1 transmit connector, and a TNC (Threaded Neil-Councilman) as the J2 receive connector. These connectors and their location on the board are shown in *Figure 21.*

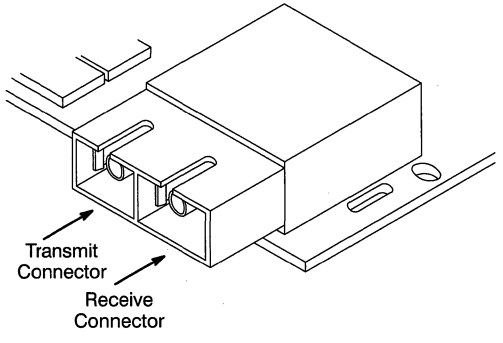


Figure 20. U4 Fiber-Optic Module Connectors

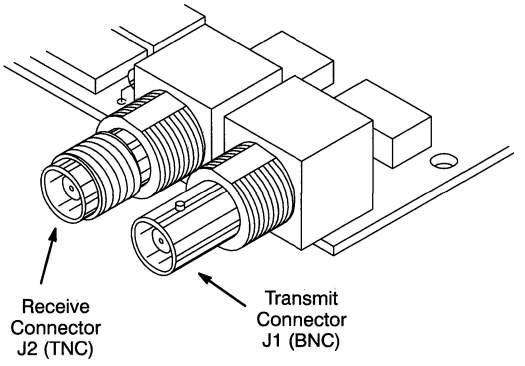


Figure 21. J1 and J2 Coaxial Board Connectors

Coaxial Cable Connectors

Many different coaxial cables may be used with the CY9266–C HOTLink Evaluation Board. The only requirements for the cable are 75Ω characteristic impedance and BNC/TNC connectors at each end to attach to the board. Other cable impedances may also be used, however, the termination (R40 and R41) and bias (R61 and R62) resistors on the board must then be changed for correct operation.

Coaxial cables for the CY9266–C should have a BNC connector on one end and a TNC connector on the other. This dual connector mechanism is specified by ANSI to prevent the inadvertent cabling of a transmitter to another transmitter, or a receiver to another receiver. When connecting cables to a CY9266–C board, the cable BNC connector always attaches to a transmit port (J1) and the cable TNC connector always attaches to a receiver port (J2). TNC/BNC dual-female barrel connectors (e.g., Amphenol #76400) are available to allow splicing of cables to evaluate multiple lengths of cable. *Figure 22* illustrates typical TNC and BNC connectors.

CY9266–T Serial Interface Connections

The CY9266–T HOTLink Evaluation Board implements a copper-based serial interface. This interface uses 150Ω shielded twisted-pair (STP)

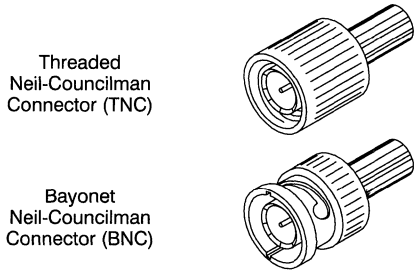


Figure 22. TNC/BNC Cable Connectors

cables with 9-pin male D-subminiature-type connectors.

STP Board Connectors

The CY9266–T HOTLink Evaluation Board has a right-angle female 9-pin D-subminiature connector. Unlike the coaxial cable version of the CY9266, which uses separate connectors for transmit and receive, the CY9266–T uses only a single connector (P1) for both. This connector and its location on the board is shown in *Figure 23*.

STP Cable Connector

There are presently two STP cable types identified by ANSI for use with Fibre Channel; both specify 150Ω differential characteristic impedance. These cable types are known as either EIA/TIA568 Type-1 and Type-2, or more generically as IBM® Type-1 or Type-2. Both of these cable types contain two individually shielded pairs of solid conductors. The Type-2 cable also contains four non-shielded con-

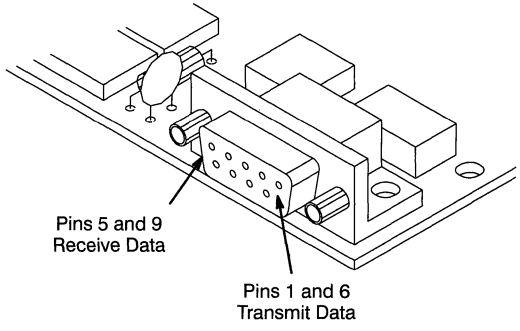


Figure 23. STP P1 Board Connector

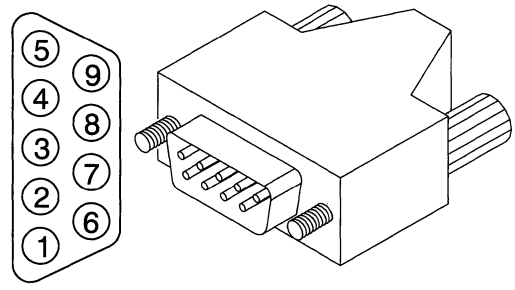


Figure 24. STP Cable Connector and Connector Pinout

ductors that are often used for either low-speed signaling or voice-grade communications.

For installations where the cables may see more flexing, a stranded conductor cable is available that meets the 150Ω impedance. This cable type is commonly known as IBM Type-6. Other cable types may also be used with the CY9266–T HOTLink Evaluation Board. The only requirements for the cable are 150Ω differential characteristic impedance and a properly wired (see *Figure 25*) 9-pin male D-subminiature connector at each end of the cable. Other cable impedances may also be used, however, the termination (R40 and R41) and bias (R61 and R62) resistors on the board must then be changed for correct operation.

Figure 24 shows an example of a compatible STP cable connector and how the pins in the connector are numbered. This is a 9-pin male D-subminiature connector. While connectors of this type are available with a plastic housing, proper operation with STP cables requires using connectors having a metal or conductive shell. When properly connected, as shown in *Figure 25*, the shield of each pair in the cable is attached to the conductive front shell of the connector. To maintain shielding effectiveness it is

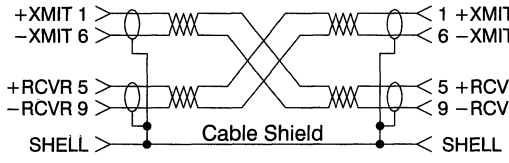


Figure 25. STP Cable Connections

recommended that the connector backshell/strain relief also be metallic or conductive.

The STP cable is wired in a crossover fashion where the transmit connections at one end of the cable are connected to the receive connections at the other end of the cable, as illustrated in *Figure 25*. The cable shields for both pairs are tied together and connected to the D-sub shell at each end.

OLC Mode Configuration

The CY9266 Evaluation Board may be configured to operate in an OLC-266 compatible system. This emulation is strictly at the TTL parallel interface level; the optical and electrical serial interfaces are not compatible. In addition, the CY9266 is only a single-channel board while the OLC-266 is available in either single- or dual-channel versions.

The TTL parallel interface attachment is provided through the JP4 connector. This connector is pinned and positioned to mate with host systems designed for the OLC-266 board.

The following configuration sets the CY9266 for 10-bit data and Bypass mode on both the transmitter and receiver. The transmitter and receiver are both clocked by the XMITCLOCK signal on JP4, and the receiver A/$\overline{B}$ selection is controlled by the LOOP_BACK signal on JP4.

JP1 Settings

The CY9266 jumper block JP1 controls many of the options on the board. For the CY9266 to operate in an OLC socket, jumper block JP1 must be configured with shorting jumpers as shown in *Figure 26*.

The shorting jumper across pins X and Y of JP1-B allows the LOOP_BACK signal in the JP4 connector to control the A/$\overline{B}$ input selection on the HOTLink Receiver. The jumper across pins X and Y of JP1-F allows the LINK_CONTROL signal to control the FOTO enable of the HOTLink Transmitter. The jumper connecting pins X and Y of JP1-G connects the XMITCLOCK input to the HOTLink Transmitter CKW clock. The jumper connecting pins JP1-HX to JP1-IX connects the XMITCLOCK input to the HOTLink Receiver REFCLK input.

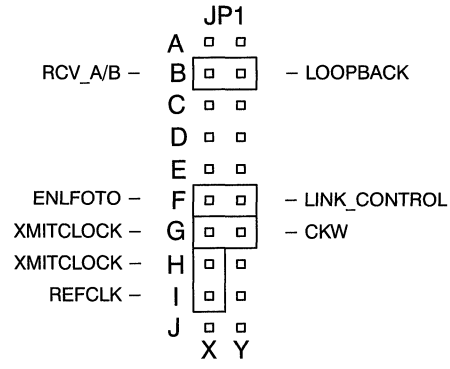


Figure 26. JP1 OLC-Compatibility Settings

Note: The active signal level of the LOOPBACK signal, as implemented on the CY9266, is opposite that of an actual OLC-266 card. If this signal is under software control, it should be programmed to allow signal loopback when the signal is active LOW. For hardware controlled systems an external signal inversion is necessary, or the signal may be jumpered at JP1 for operation from the S1-7 DIP switch.

S1 Settings

The S1 DIP switch is also used to configure many of the HOTLink Transmitter and Receiver options. The settings for these switches are listed in *Table 6*.

Table 6. S1 OLC-Compatibility Settings

DIP Switch Settings		
Sw #	**State**	**Controlled Signal**
1	Off	Transmitter BIST Enable
2	Off	Transmitter Mode Select
3	Off	Enable Next Parallel Xmit Data
4	On	Enable Parallel Xmit Data
5	Off	Receiver BIST Enable
6	Off	Receiver Mode Select
7	N/A	Switch Controlled Loopback
8	N/A	Switch Controlled FOTO
9	Off	Carrier-Detect Polarity Select
10	Off	BYTE_SYNC Polarity Select

The setting of switches S1-7 and S1-8 are not applicable when jumpers JP1-B and JP1-F are in place.

Assembly and Options

The design of the CY9266–F and CY9266–C/T Evaluation Boards offer many different assembly options for those users interested in making modifications for their own evaluation.

Optical Module

Optical module U4 on the CY9266–F is socketed for user evaluation of different optical modules. The hole pattern on the board supports direct soldering of the optical module to the board. This should not be attempted on a board that is already equipped with a socket for the module because removal of the socket pins may damage the board.

Transmitter

The HOTLink Transmitter B± differential output signals on the board are left open to conserve power. Pads are present on the bottom of the board (labeled R1, R2, R3, and R4) for bias/termination resistors for these outputs. While these resistors are present on the board schematic, they are not part of the delivered assembly. If the B± outputs are used for probing or test purposes, resistors must be added in these locations to enable the output drivers.

Oscillator

The on-board oscillator (U5) is used primarily for exercising the BIST capability of the board in a standalone mode. If the board is only used with an external clock, the oscillator does not need to be present. This part is socketed to allow the user to select the operating frequency.

When selecting an oscillator, care must be taken to insure the frequency stability and jitter characteristics of the oscillator are within the specifications of the HOTLink Receiver and Transmitter and the intended system application.

The hole pattern on the board supports direct soldering of the oscillator to the board. This should not be attempted on a board that is already equipped with a socket for the oscillator, as removal of the socket pins may damage the board.

BIST Support Hardware

The BIST support hardware does not interact with the functionality of the HOTLink Transmitter or Receiver and is not part of the communications link. If there is no requirement for BIST and display hardware, the following components may be removed from the board:

- U6 and U7—TIL311 Hex Displays
- U8—CY7C344 EPLD
- S2—Reset Switch
- R21, R22, R23, and R24—1 kΩ
- C13—0.022 µF
- C18—100 pF

Voltage Monitor

The voltage monitor (U11) is used as part of the BIST function and also drives the RESET signal on JP2, JP3, and JP4. If monitoring of the specific voltage is not necessary (and BIST capability is not used) this part may be removed.

If U11 is removed, it may be necessary to bias the RESET line to allow an external system controller to properly sense a high on the RESET output. This may be done by soldering a jumper wire from pin 7 of U11 to pin 2 of R20.

JP2

The area of the board labeled as JP2 provides a hole pattern designed to accept multiple types of headers and connectors. These connectors allow access to all the same signals present on JP4 and JP3.

The current pin 1 designation for JP2 assumes a pin-header connector designed for flat cable is attached to bottom of the board. If this type of connector is instead attached to the top of the board, the even and odd pins are effectively swapped in the connector and cable, from those listed in *Table 1*.

OLC-Compatibility Registers

The 74F174 hex D-registers (U9 and U10) are used to provide compatibility with OLC-266 sockets. For

those users not requiring this capability, or for those who wish to use the receiver $\overline{RDY}$ signal to clock received data into asynchronous FIFOs, these registers can be removed.

Once U9 and U10 are removed, it is necessary to short eleven adjacent pad pairs on U9 and U10 to allow the receiver data bus to connect to the output connectors. The pairs that must be shorted are listed in *Table 7.*

Table 7. OLC-Compatibility Register Bypass Connections

Register Pin Connections		
Part	**Pins**	**Signal Name**
U10	14, 15	RCVR_0
U10	12, 13	RCVR_1
U10	10, 11	RCVR_2
U10	6, 7	RCVR_3
U10	4, 5	RCVR_4
U10	2, 3	RCVR_5
U9	10, 11	RCVR_6
U9	12, 13	RCVR_7
U9	14, 15	RCVR_8
U9	6, 7	RCVR_9
U9	4, 5	RDY

Copper Cable Connectors

The CY9266–C and CY9266–T are assembled on the same substrate and may be configured for use with either coaxial or shielded-pair cables. Changing from coax to shielded-pair requires the removal of the J1 BNC and J2 TNC connectors and replacing them with a female 9-pin D-sub connector at location P1 (see Appendix B for manufacturer part numbers). Also, the foil traces that connect pins 6 and 9 of P1 to the shield of J1 and J2 (located on the bottom of the board) must be cut. Because the cable impedance used for shielded-pair cable is different from that of coax cable, the line termination resistors R40 and R41 must be replaced with 75Ω resistors, and coupling transformer T1 must also change to the higher inductance type.

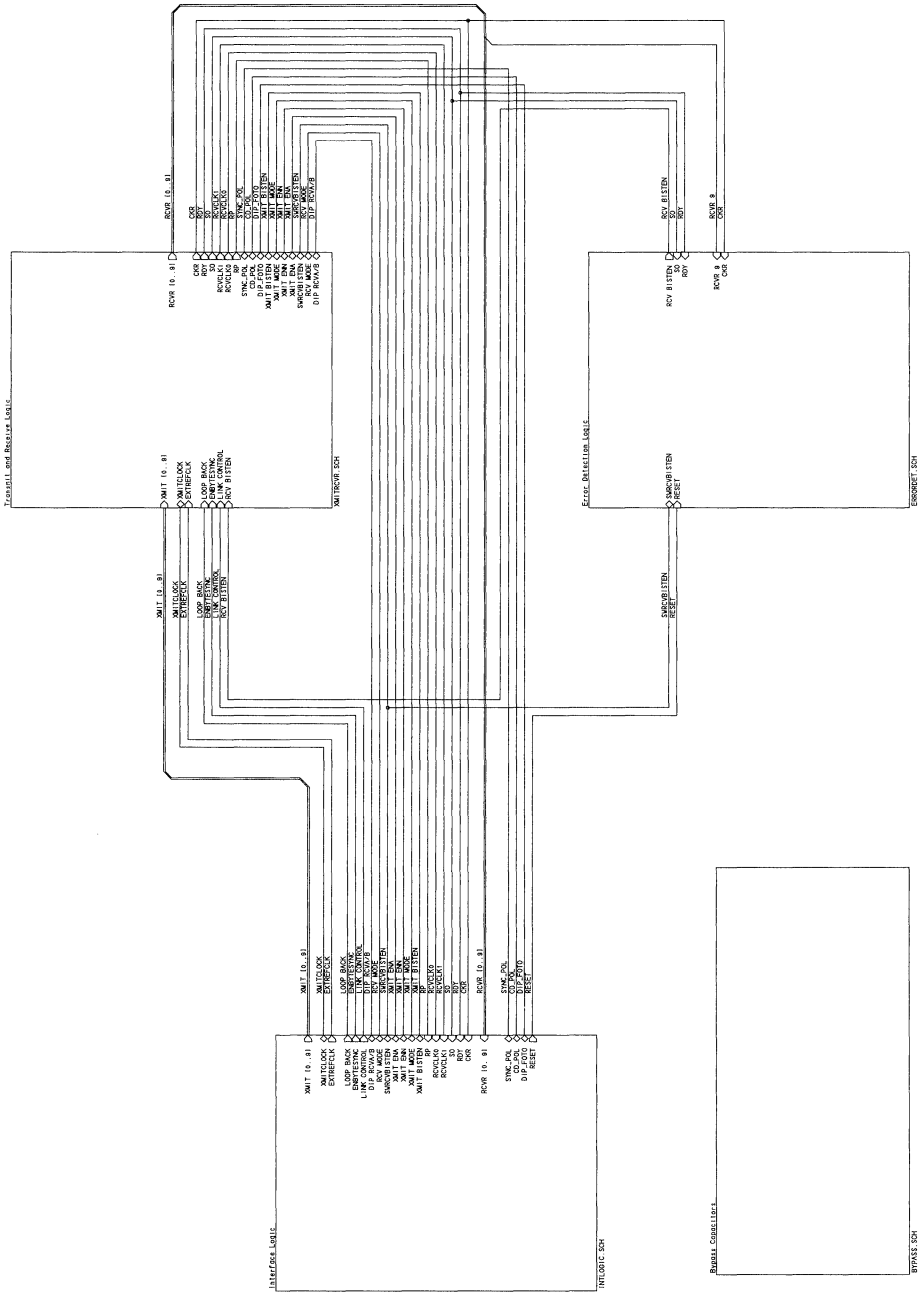
Changing from shielded-pair to coax requires removal of the P1 D-sub connector and the addition of connectors J1 and J2. It is necessary to connect pin 6 of the P1 pad set to the shield pin of J1, and pin 9 of P1 to the shield pin of J2. Because the cable impedance used for coax cable is different from that of shielded-pair cable, the line termination resistors R40 and R41 must be replaced with 37.4Ω resistors, and coupling transformer T1 must also change to the lower inductance type.

Redesign Capability

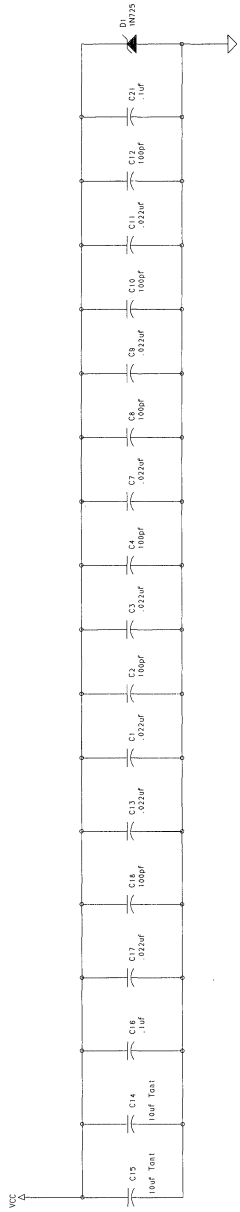
The CY9266–F, CY9266–C, and CY9266–T boards were designed strictly as a demonstration vehicle for the Cypress Semiconductor HOTLink family of communications parts. The designs shown here may not be optimal for most applications, as these are expected to be more specialized and may not require all the configuration and BIST demonstration hardware contained on these boards.

Examination of the evaluation boards will show that the components necessary for creating a serial link are all on one half of the board, while the components used for configuration and BIST support are located on the other half of the board. This placement of parts was intentional, and shows that two complete channels may be placed on a board of the same size as the CY9266 *without* placing active components on both sides of the board.

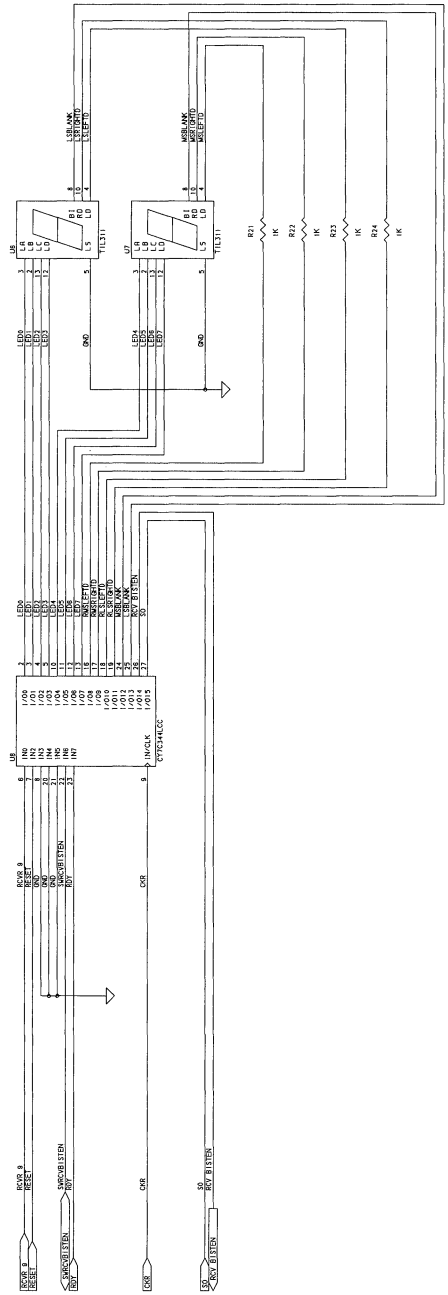
Appendix A. CY9266−F Schematic (Sheet 1 of 5)



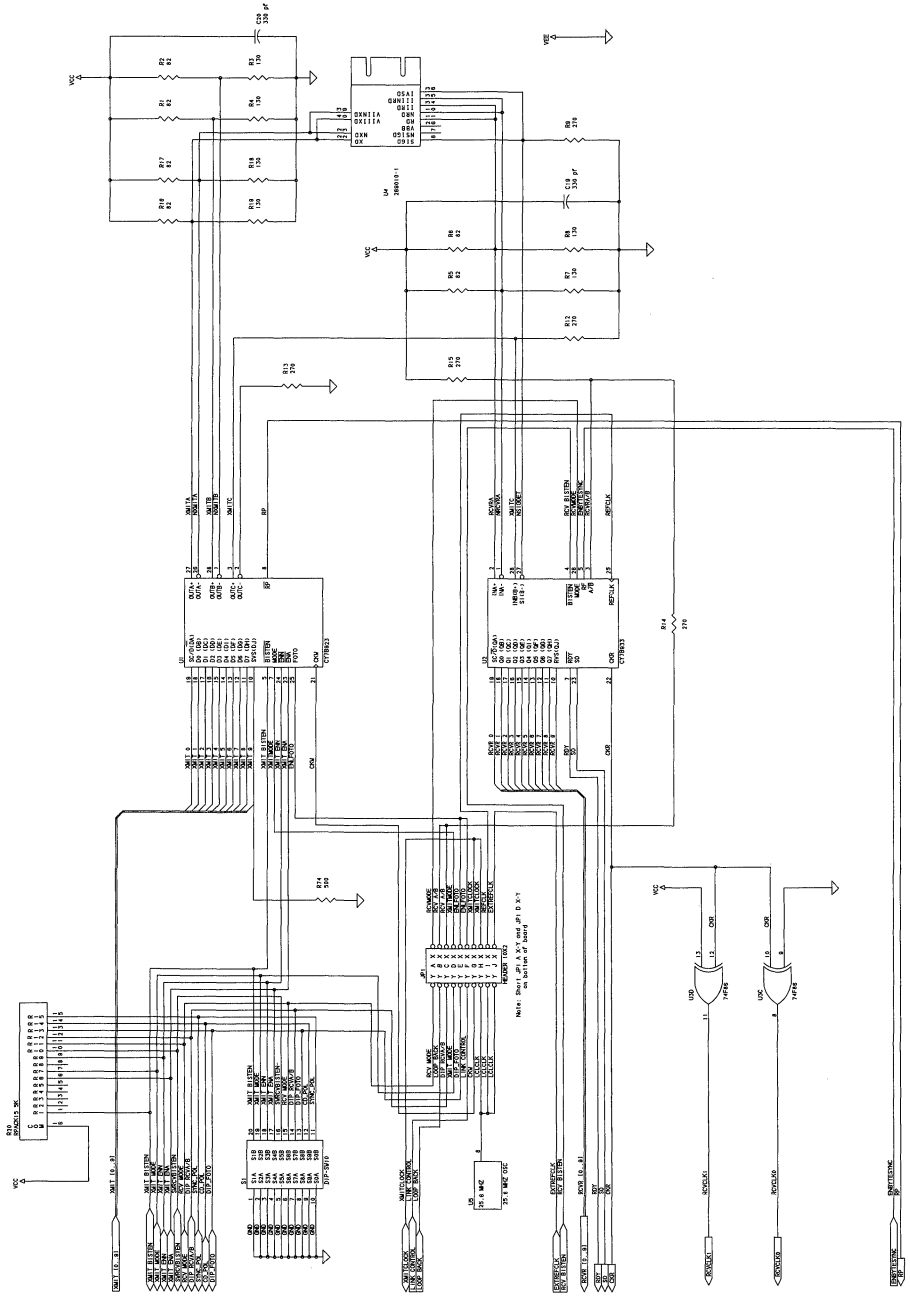
Appendix A. CY9266–F Schematic (Sheet 2 of 5)



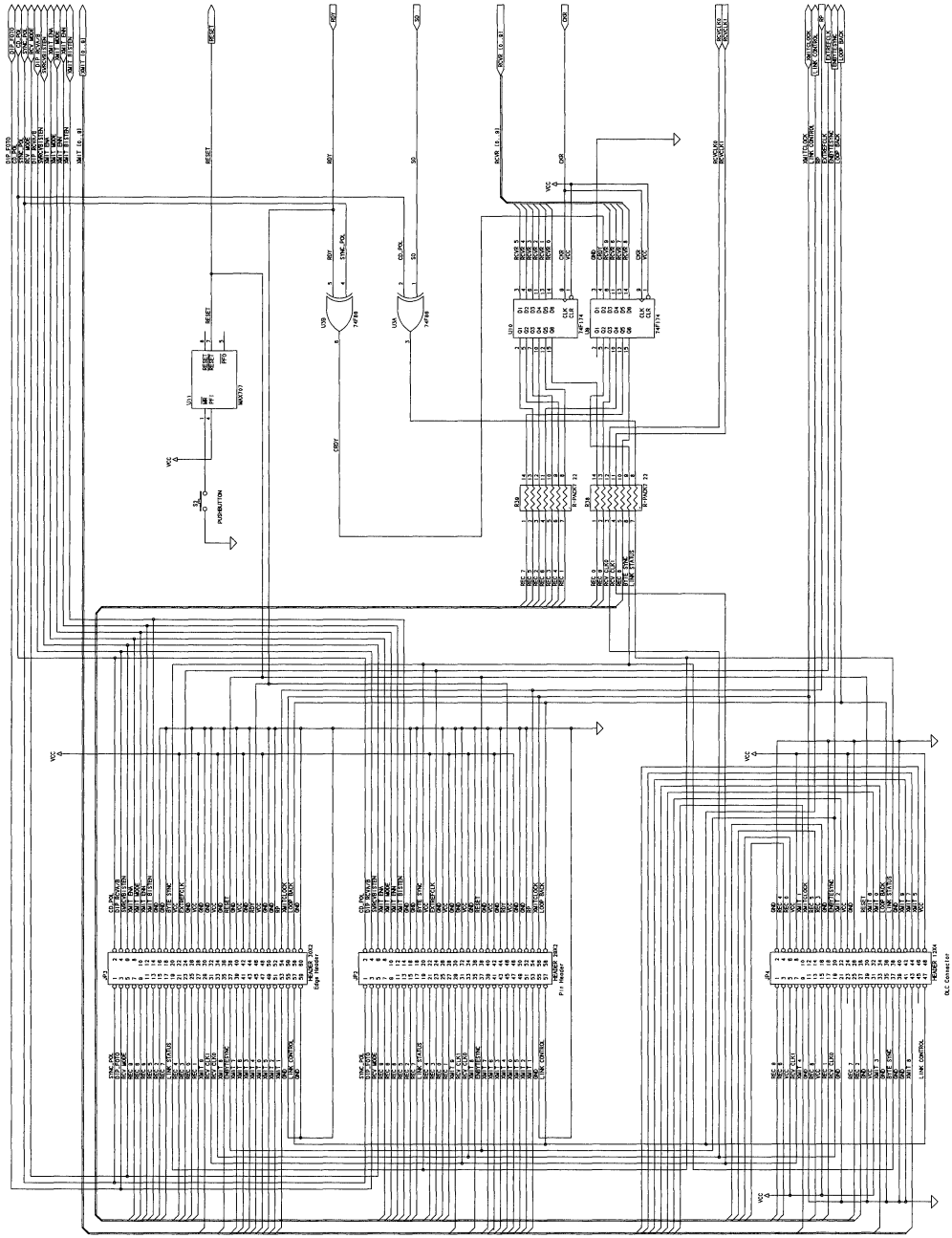
Appendix A. CY9266−F Schematic (Sheet 3 of 5)



Appendix A. CY9266−F Schematic (Sheet 4 of 5)



Appendix A. CY9266−F Schematic (Sheet 5 of 5)

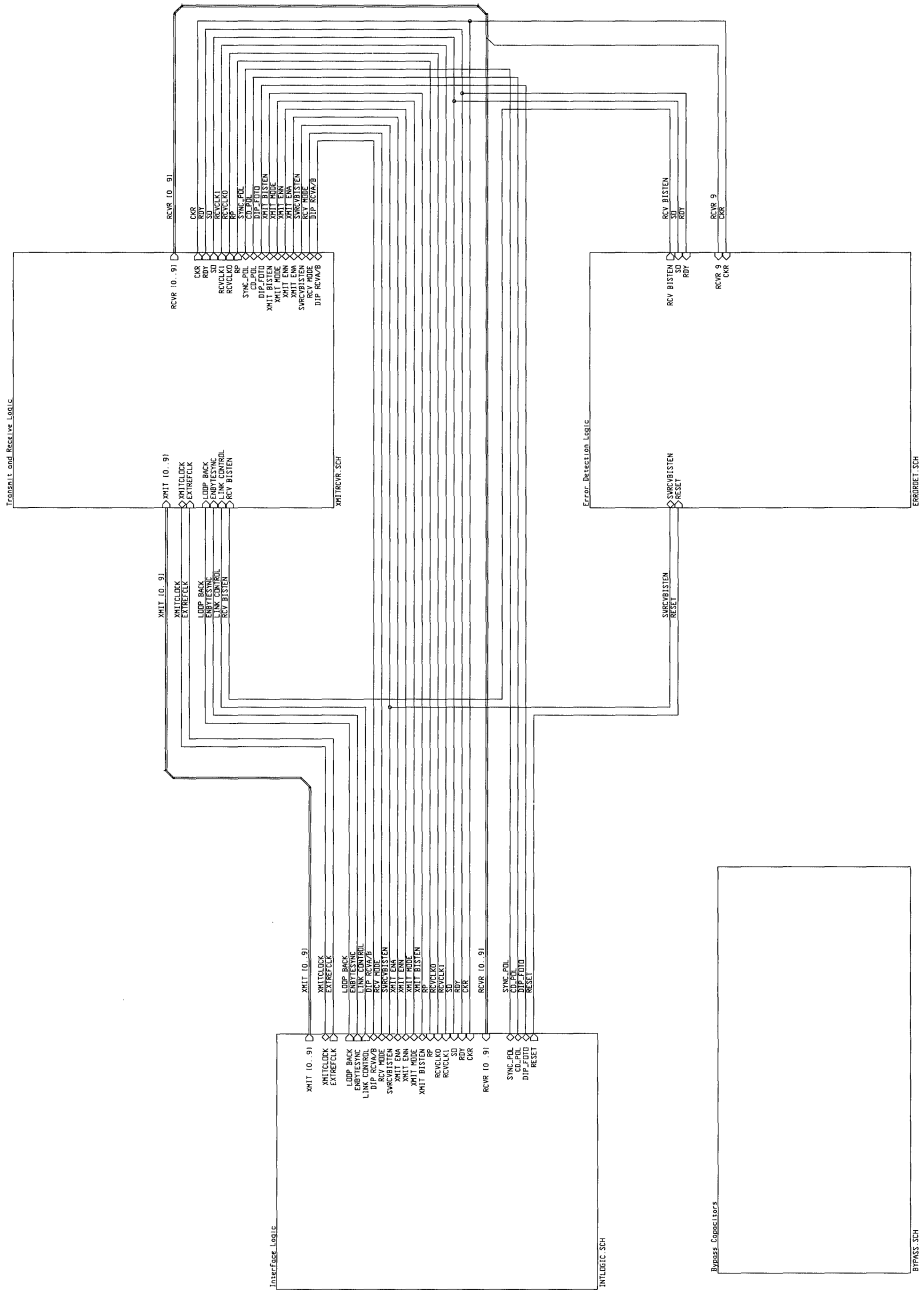


Appendix A. CY9266–F Parts List

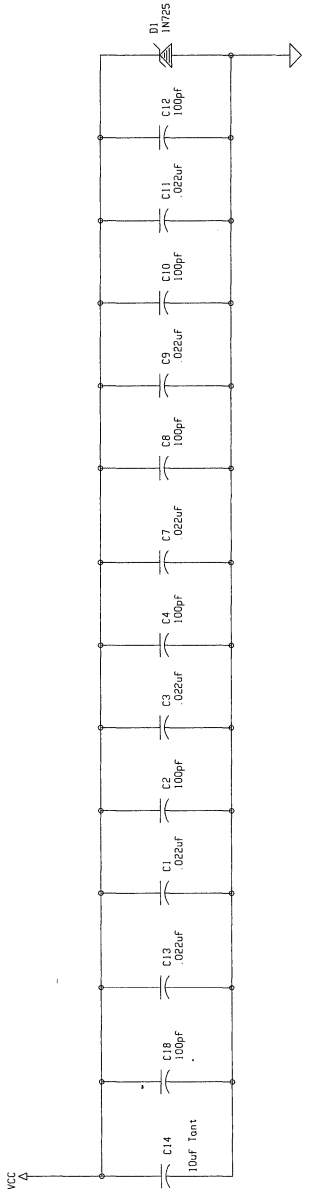
Instance	Part Number	Description
U1	Cypress CY7B923-JC	HOTLink Transmitter
U2	Cypress CY7B933-JC	HOTLink Receiver
U3	74F86	Quad XOR Gate SOIC Package
U4	AMP/Lytel 269063-1, Hewlett Packard HFBR-5302, Siemens TC-266C2EP, CTS 1408N, or Equivalent	266-MB/s 1300-nm LED Transceiver Module
U5*	CTS CTX126 or Equivalent	25-MHz TTL Clock Oscillator
U6*,U7*	TI TIL311	Hex Display With Logic
U8*	Cypress CY7C344-15HC	32-Macrocell MAX EPLD
U9,U10	74F174	Hex D-Register, SOIC Package
U11*	Maxim MAX707CSA or Equivalent	Voltage Monitor
D1*	1N4735A	1W, 6.2V Zener Diode
S1*	AMP 3-435668-0 or Equivalent	10-position DIP Switch
S2*	ECG 520-01-3 or Equivalent	Momentary Pushbutton Switch
JP1*	Sullins PZC10DAAN or Equivalent	2 x 10 Position 0.25″ Sq. Pin-Header
JP4	Sullins PZC12DFBN or Equivalent	2 – 2 x 12 Position 0.25″ Sq. Pin-Header
C1, C3, C7, C9, C11, C13, C17	0.022 µF MLC X7R	0805 Chip Cap
C2, C4, C8, C10, C12, C18	100 pF MLC NPO	0805 Chip Cap
C14, C15	10 µF 16V Tantalum	Electrolytic Cap
C16, C21	0.1 µF MLC X7R	1206 Chip Cap
C19, C20	330 pF MLC NPO	0805 Chip Cap
R5, R6, R16, R17	82Ω 1/8W	1206 Chip Resistor
R7, R8, R18, R19	130Ω 1/8W	1206 Chip Resistor
R9, R12, R13, R14, R15	270Ω 1/8W	1206 Chip Resistor
R21*, R22*, R23*, R24*	1-kΩ 1/8W, 5%	1206 Chip Resistor
R74	510Ω 1/8W, 5%	1206 Chip Resistor
R20	CTS 766-161-R512 or Equivalent	5.1-kΩ R-Pack-15 SO16
R38, R39	CTS 766-143-R220 or Equivalent	22Ω R-Pack-7 SO14
	AMP 645955-2 or Equivalent	41 – Low Profile Socket-Pin
	3M 929955-06 or Equivalent	4 – 0.1″ Centerline Shorting Jumper

* — Used only for supervisory functions. Not needed for communications.

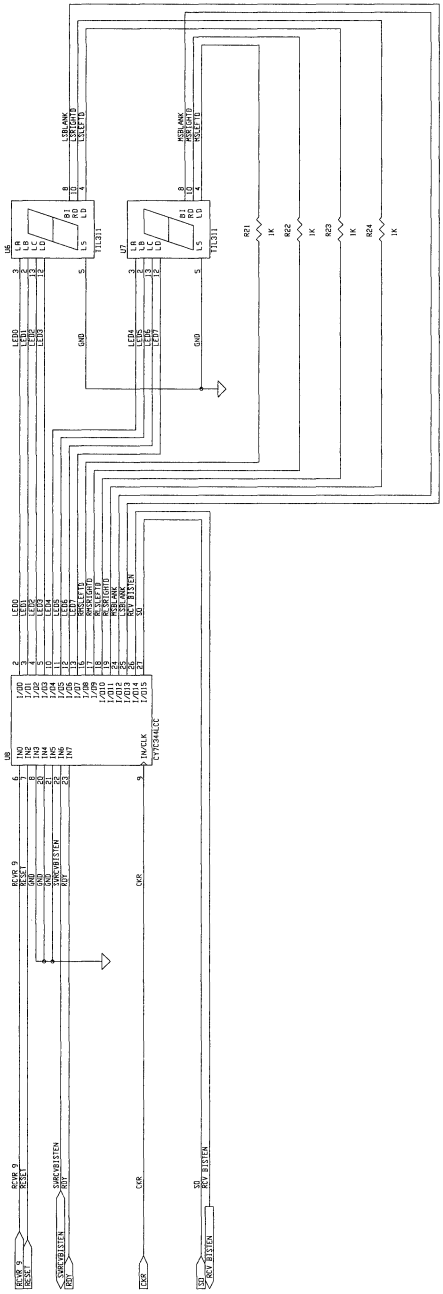
Appendix B. CY9266−C/T Schematic (Sheet 1 of 5)



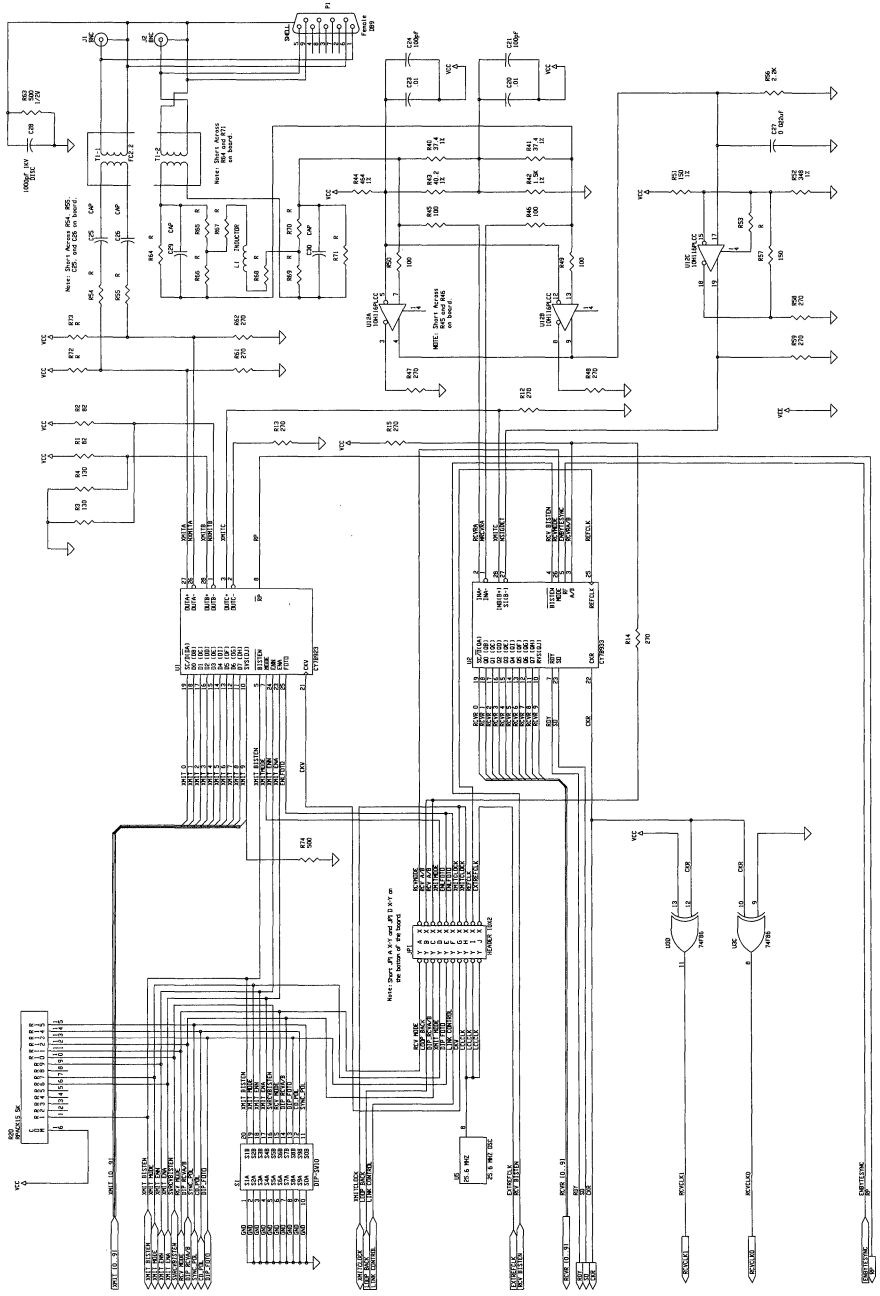
Appendix B. CY9266−C/T Schematic (Sheet 2 of 5)



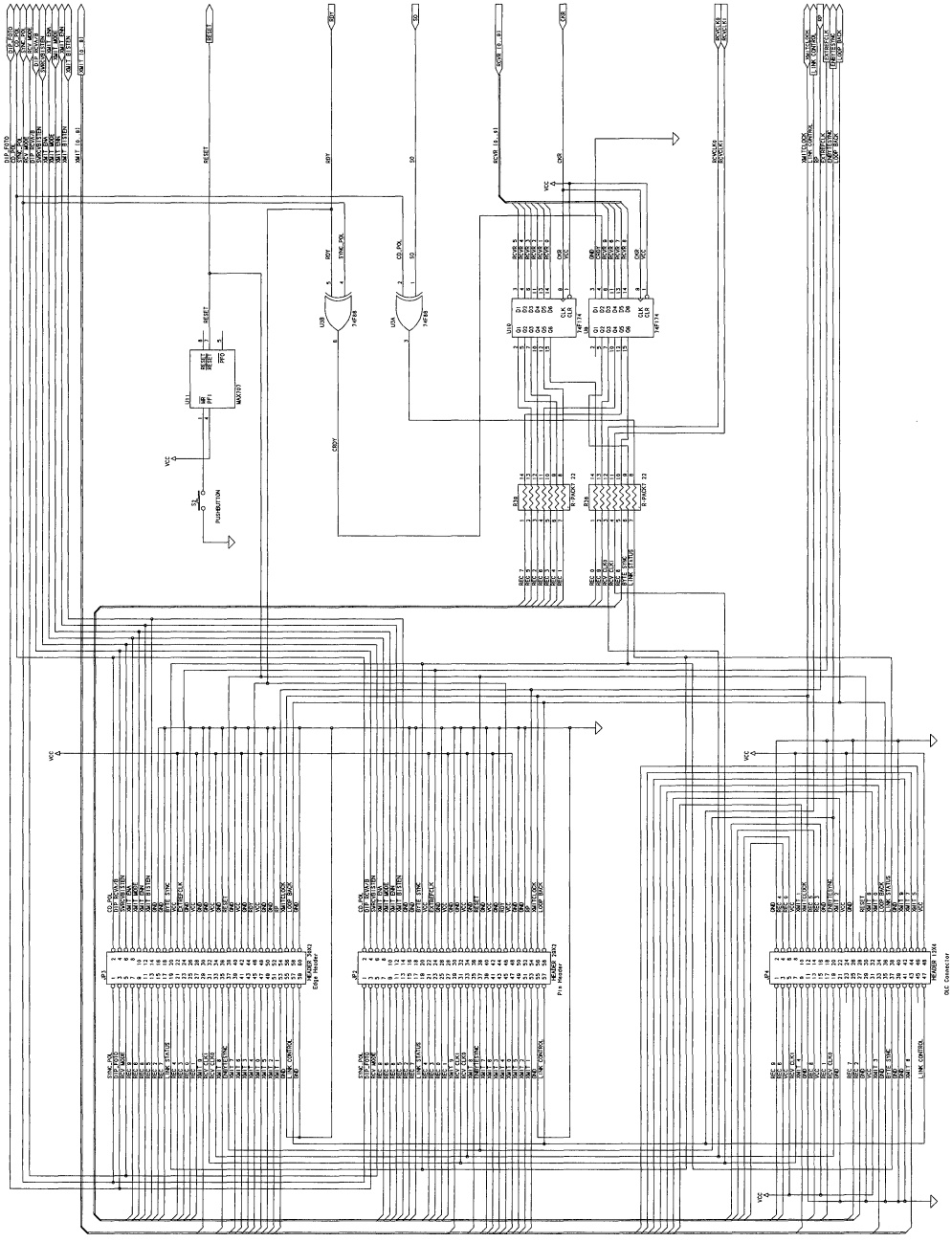
Appendix B. CY9266−C/T Schematic (Sheet 3 of 5)



Appendix B. CY9266−C/T Schematic (Sheet 4 of 5)



Appendix B. CY9266−C/T Schematic (Sheet 5 of 5)



Appendix B. CY9266–C/T Parts List

Instance	Part Number	Description
U1	Cypress CY7B923-JC	HOTLink Transmitter
U2	Cypress CY7B933-JC	HOTLink Receiver
U3	74F86	Quad XOR Gate SOIC Package
U5*	CTS CTX126 or Equivalent	25-MHz TTL Clock Oscillator
U6*,U7*	TI TIL311	Hex Display With Logic
U8*	Cypress CY7C344-15HC	32-Macrocell MAX EPLD
U9,U10	74F174	Hex D-register, SOIC Package
U11*	Maxim MAX707CSA or Equivalent	Voltage Monitor
U12*	Motorola MC10H116FN	ECL Tripple Line Receiver
D1*	1N4735A	1W, 6.2V Zener Diode
S1*	AMP 3-435668-0 or Equivalent	10-position DIP Switch
S2*	ECG 520-01-3 or Equivalent	Momentary Pushbutton Switch
J1	227161-3 or Equivalent	RA Female BNC Connector
J2	227818-1 or Equivalent	RA Female TNC Connector
JP1*	Sullins PZC10DAAN or Equivalent	2 x 10 Position 0.25″ Sq. Pin-Header
JP4	Sullins PZC12DFBN or Equivalent	2 – 2 x 12 Position 0.25″ Sq. Pin-Header
P1	747844-6 or Equivalent	RA Female 9-Pin D-Sub Connector
C14	10 µF 16V Tantalum	Electrolytic Cap
C1, C3, C7, C9, C11, C13, C27*	0.022 µF MLC X7R	0805 Chip Cap
C2, C4, C8, C10, C12, C18, C21, C24*	100 pF MLC NPO	0805 Chip Cap
C20, C23*	0.01 µF MLC X7R	0805 Chip Cap
C28	1000 pF 1 kV, Y5P	Disc Cap
T1	Pulse Engineering PE-65507 for STP Pulse Engineering PE-65508 for coax	Dual-Wideband Pulse Transformer
R12, R13, R14, R15	270Ω 1/8W, 5%	1206 Chip Resistor
R74	510Ω 1/8W, 5%	1206 Chip Resistor
R21*, R22*, R23*, R24*	1-kΩ 1/8W, 5%	1206 Chip Resistor
R40, R41	37.4Ω 1/10W, 1% for Coax 75.0Ω 1/10W, 1% for STP	0805 Chip Resistor
R43	40.2Ω 1/10W, 1%	0805 Chip Resistor
R49*, R50*	100Ω 1/10W, 5%	0805 Chip Resistor

Appendix B. CY9266–C/T Parts List (continued)

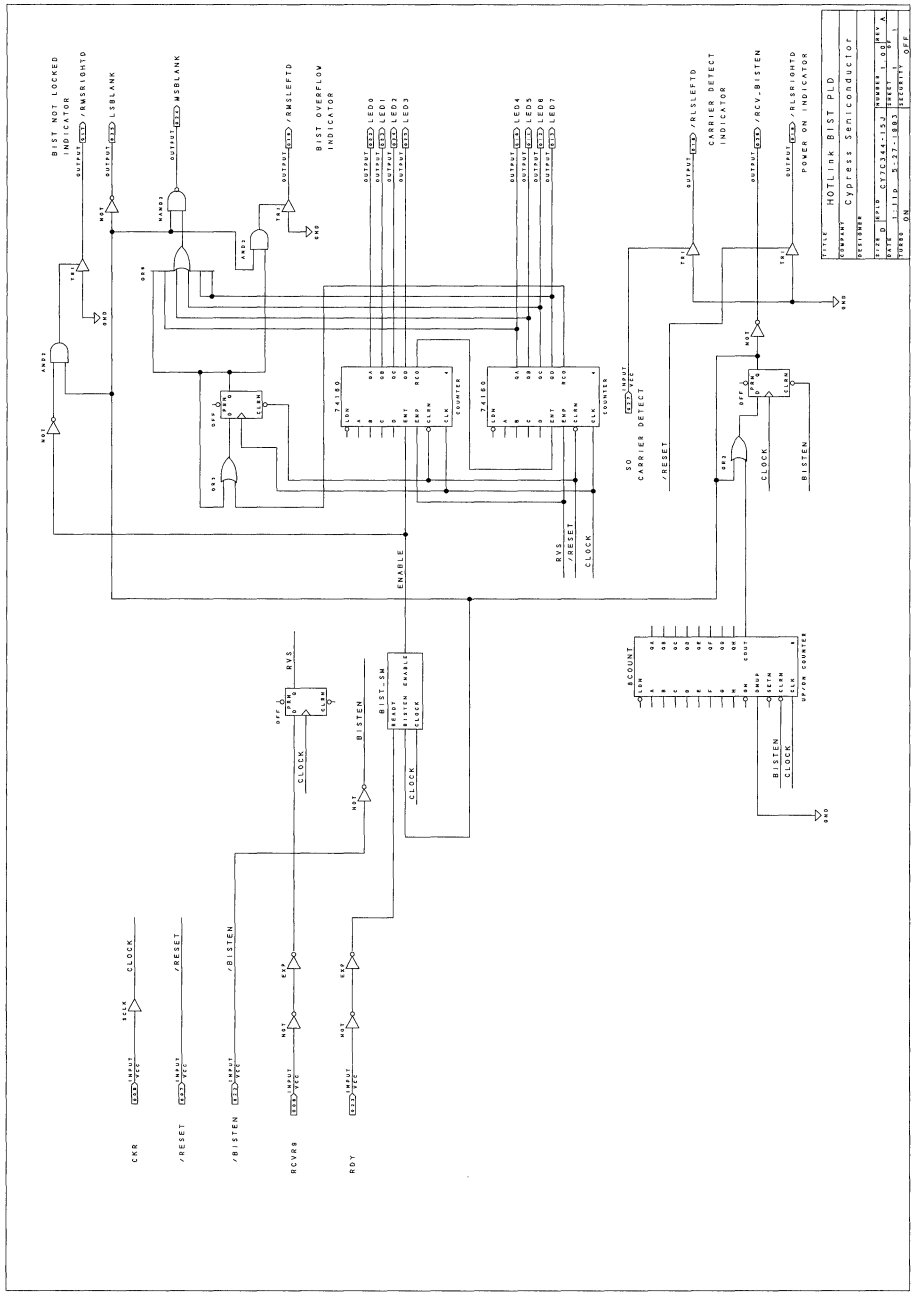
Instance	Part Number	Description
R51*, R57*	150Ω 1/10W, 1%	0805 Chip Resistor
R47*, R48*, R58*, R59*	270Ω 1/10W, 5% for 150Ω cable	0805 Chip Resistor
R61, R62	200Ω 1/10W, 5% of 75Ω cable	0805 Chip Resistor
R52*	348Ω 1/10W, 1%	0805 Chip Resistor
R44	464Ω 1/10W, 1%	0805 Chip Resistor
R42	1.5-kΩ 1/10W, 1%	0805 Chip Resistor
R56*	2.2-kΩ 1/10W, 5%	0805 Chip Resistor
R63	510Ω 1/2W	Axial Lead Resistor
R20	CTS 766-161-R512 or Equivalent	5.1-kΩ R-Pack-15 SO16
R38, R39	CTS 766-143-R220 or Equivalent	22Ω R-Pack-7 SO14
	AMP 645955-2 or Equivalent	4 – Low Profile Socket-Pin
	3M 929955-06 or Equivalent	4 – 0.1″ Centerline Shorting Jumper

* — Used only for supervisory functions. Not needed for communications.

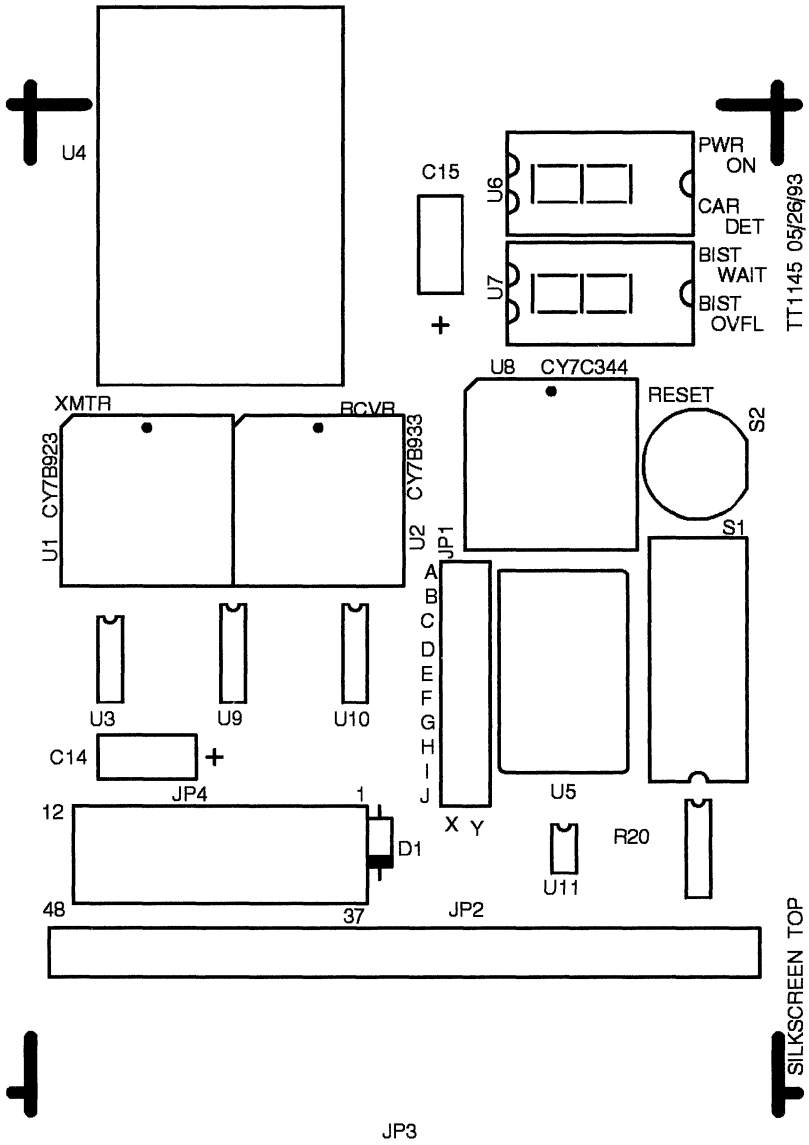
Appendix C. BIST PLD State Machine Source Code

```
SUBDESIGN bist_sm  (ready, bisten, clock : INPUT;
                    enable               : OUTPUT)


VARIABLE
       ss : MACHINE OF BITS  (enable_q)
                     %state    output%
            WITH STATES (wait0    =   0,
                         wait1    =   0,
                         wait2    =   0,
                         enabled  =   0,
                         locked1  =   1,
                         locked2  =   1);

BEGIN
       ss.clk = clock;              %assign machine clock%
       enable = enable_q;           %assign output of machine%

       TABLE
              %present    present            next %
              % state     inputs             state%
                 ss,   bisten, ready =>      ss;

       % define reset vectors %
                wait0,      0,   x    =>      wait0;
                wait1,      0,   x    =>      wait0;
                wait2,      0,   x    =>      wait0;
                enabled,    0,   x    =>      wait0;
                locked1,    0,   x    =>      wait0;
                locked2,    0,   x    =>      wait0;
       % define operational vectors %
                wait0,      1,   x    =>      wait1;
                wait1,      1,   x    =>      wait2;
                wait2,      1,   x    =>      enabled;
                enabled,    1,   1    =>      enabled;
                enabled,    1,   0    =>      locked1;
                locked1,    1,   1    =>      enabled;
                locked1,    1,   0    =>      locked2;
                locked2,    1,   1    =>      locked1;
                locked2,    1,   0    =>      locked2;
       END TABLE;
END;
```

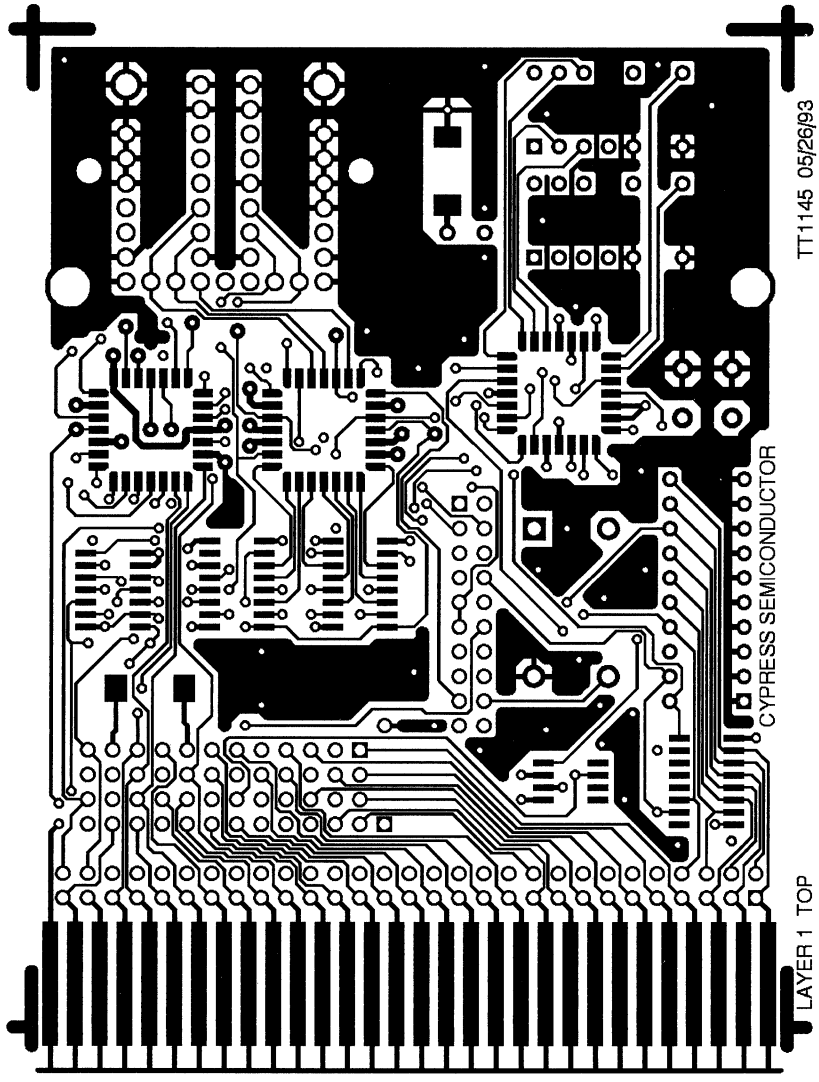
Appendix C. BIST PLD Logic Schematic



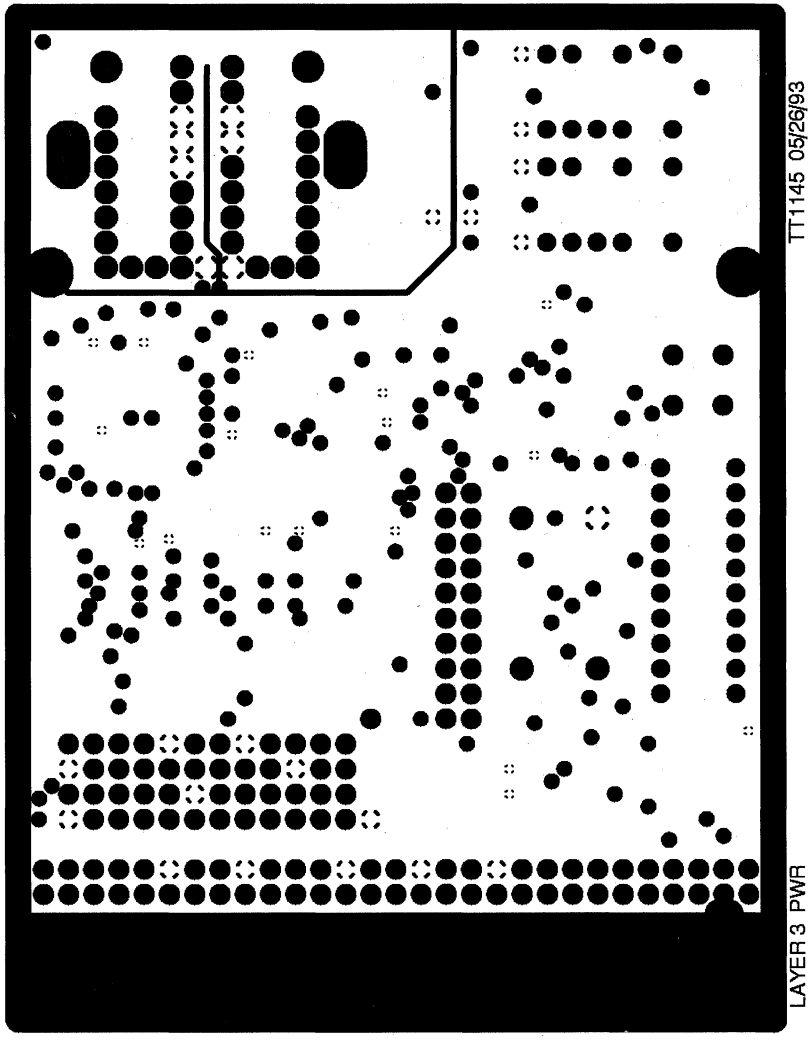
Appendix D. CY9266–F Artwork — Top Silkscreen



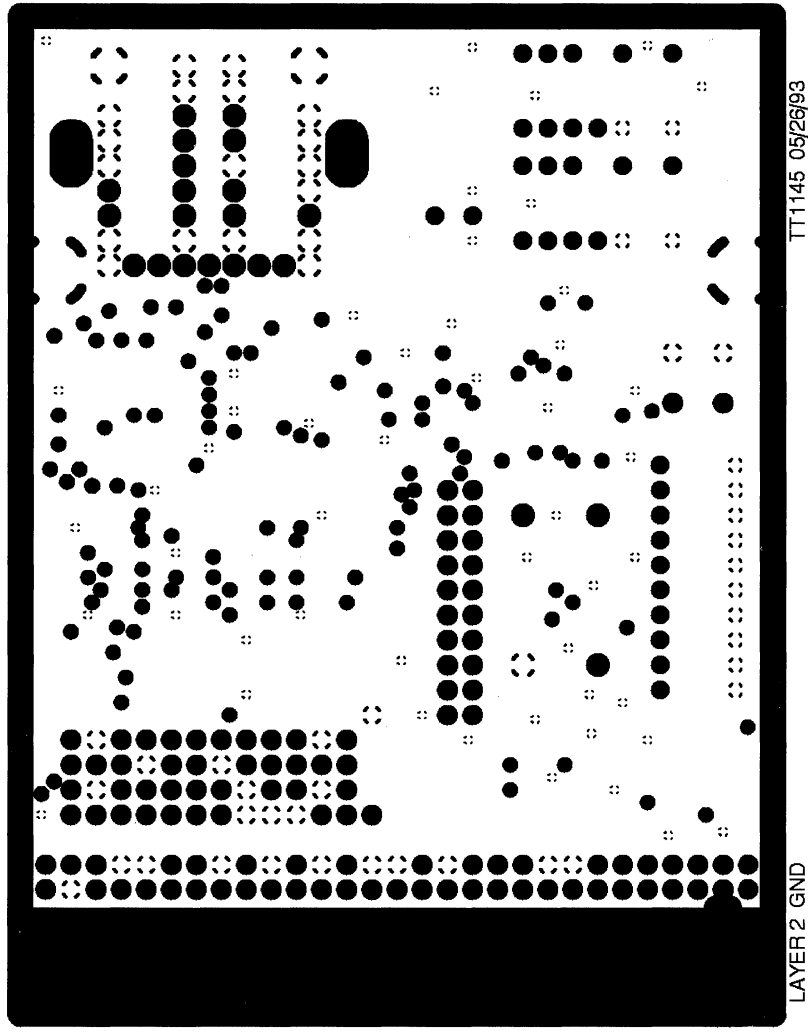
Appendix D. CY9266−F Artwork — Top Layer Copper



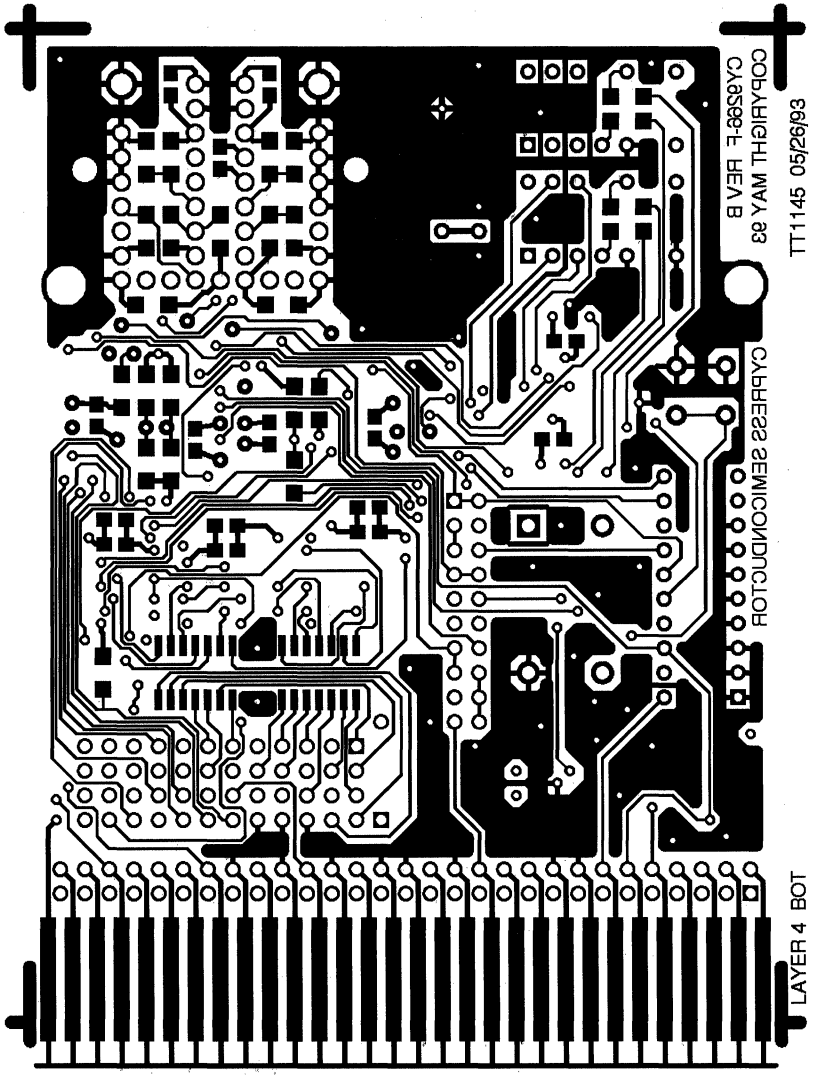
Appendix D. CY9266−F Artwork — Power Layer



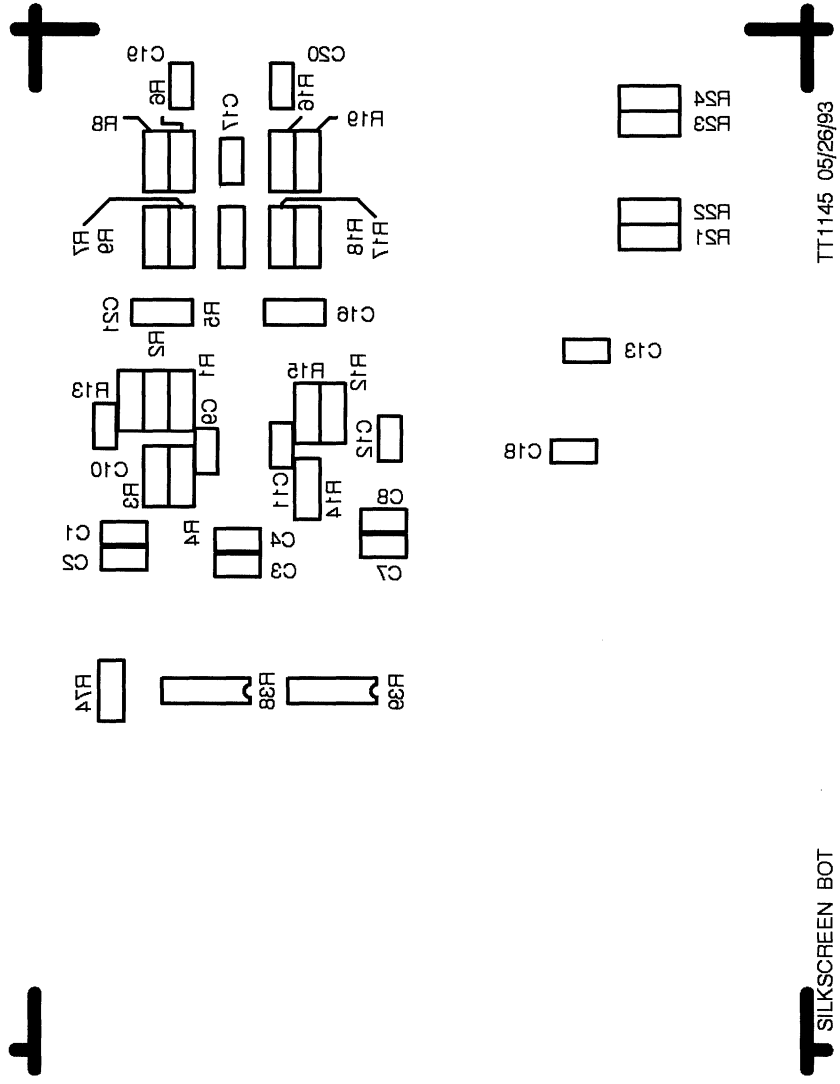
Appendix D. CY9266−F Artwork — Ground Layer



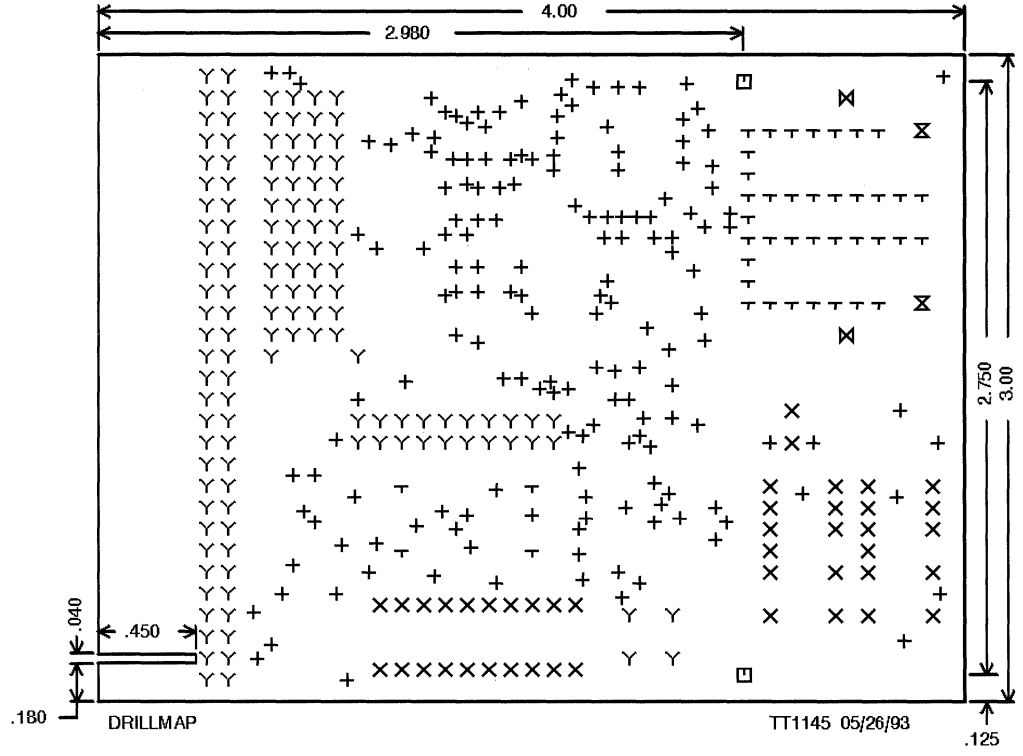
Appendix D. CY9266−F Artwork — Bottom Layer Copper



Appendix D. CY9266–F Artwork — Bottom Silkscreen



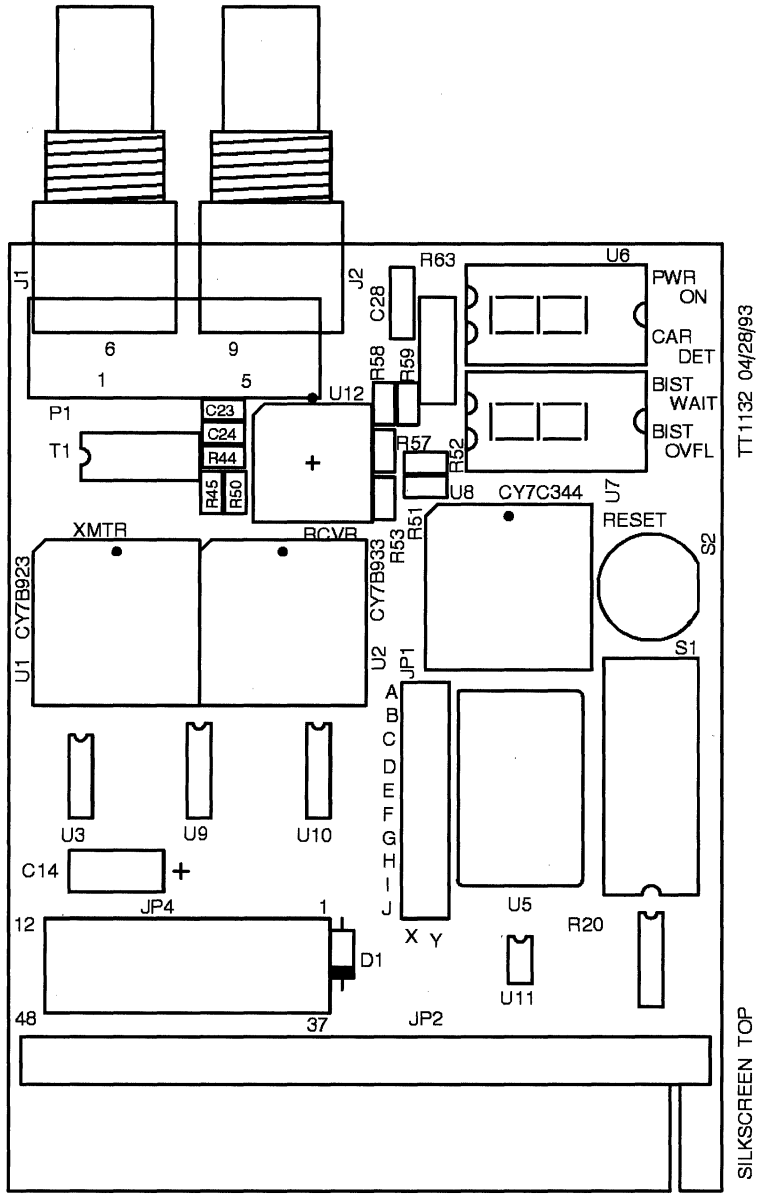
Appendix D. CY9266–F Artwork — Drill Chart



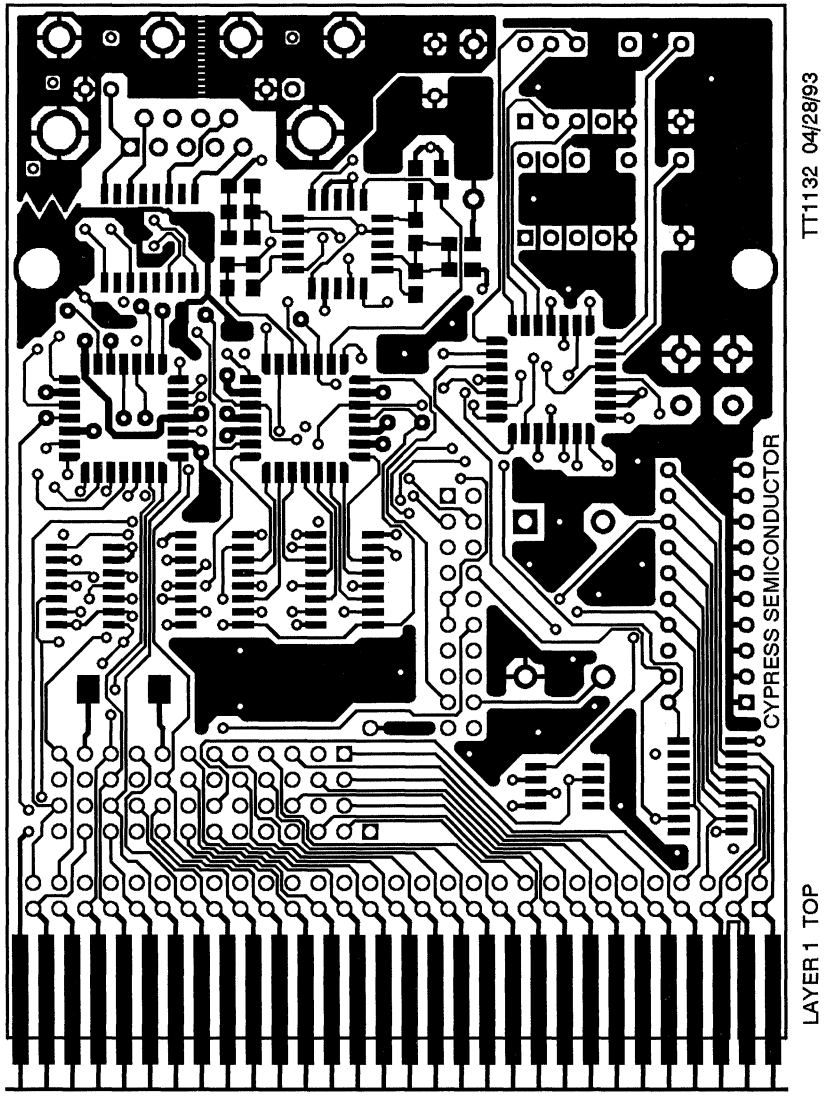
TOOL	SIZE	SYM	QTY	PLT
T01	.020	+	162	Y
T02	.032	X	44	Y
T03	.040	Y	132	Y
T04	.052	⊤	41	Y
T05	.085	⧖	4	Y
T06	.156	⊡	2	Y
T06	↓	⋈	2	Y

.200 X .100 OVAL HOLE

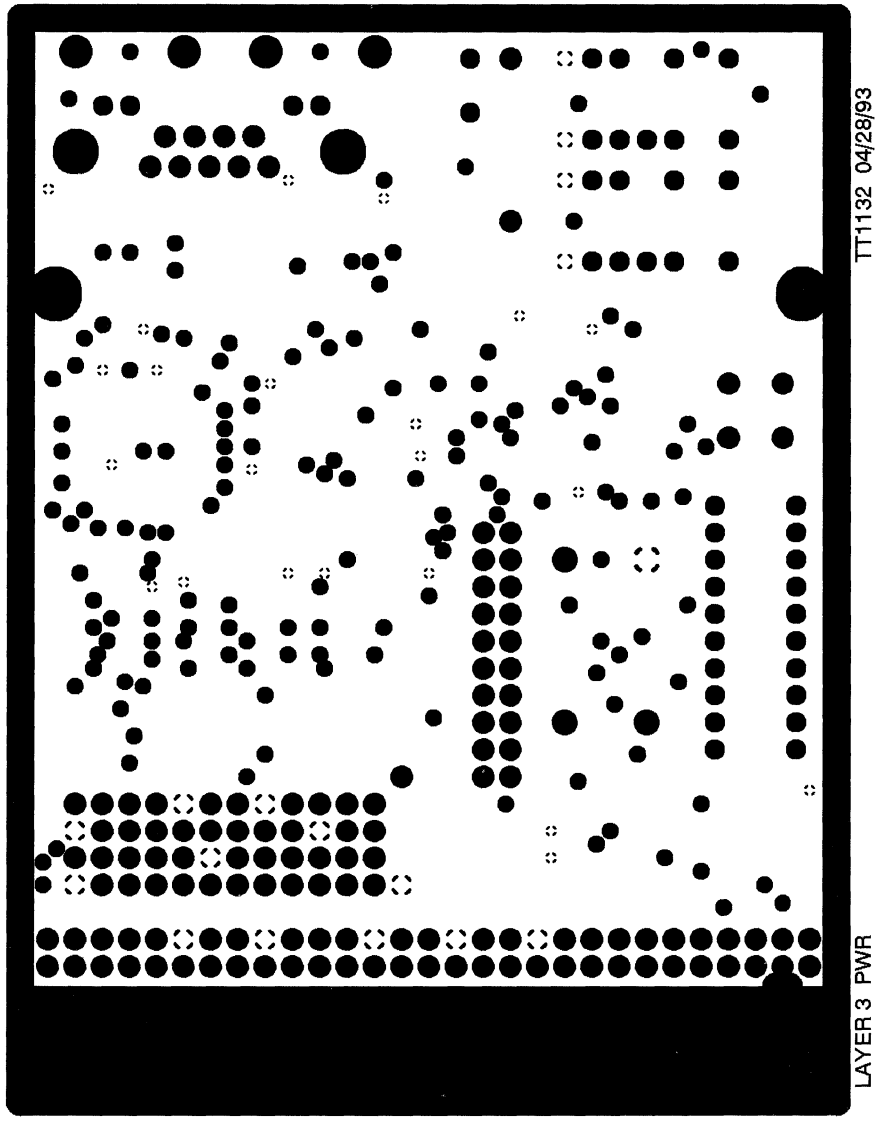
Appendix E. CY9266–C/T Artwork — Top Silkscreen



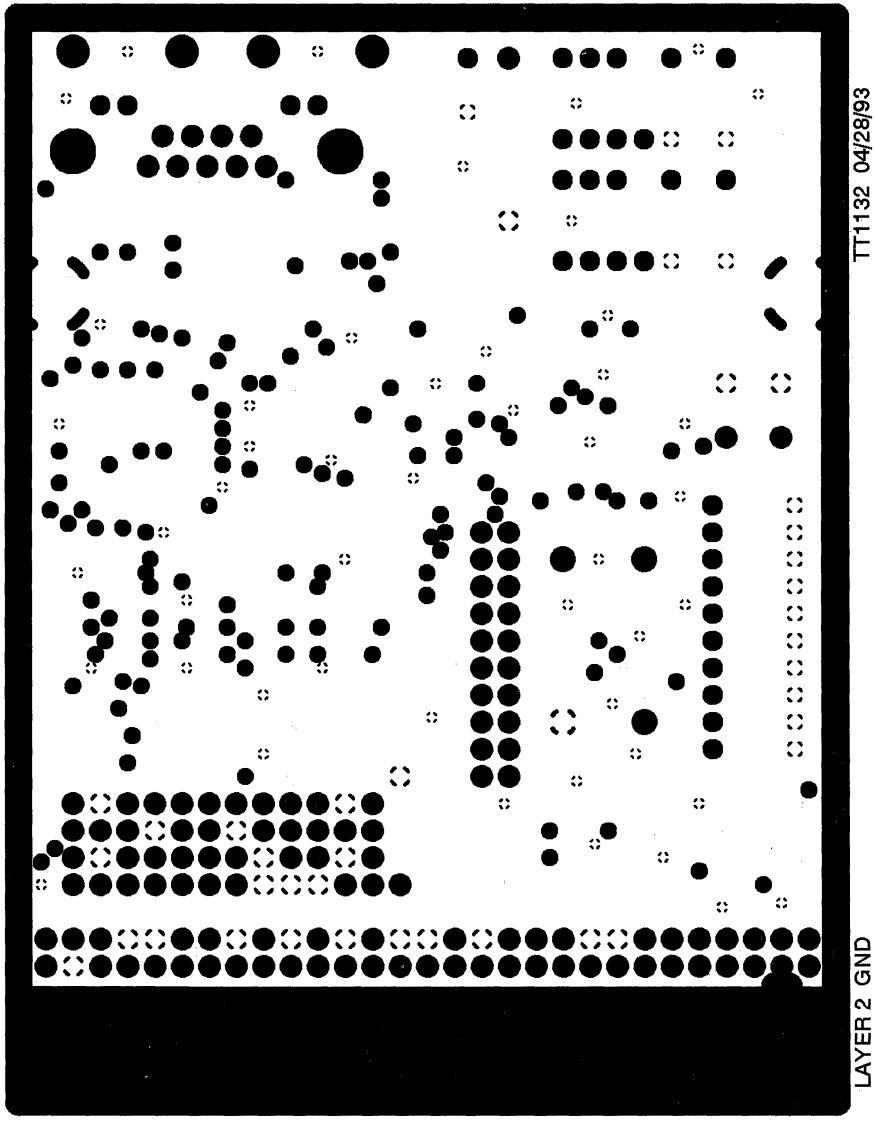
Appendix E. CY9266–C/T Artwork — Top Layer Copper



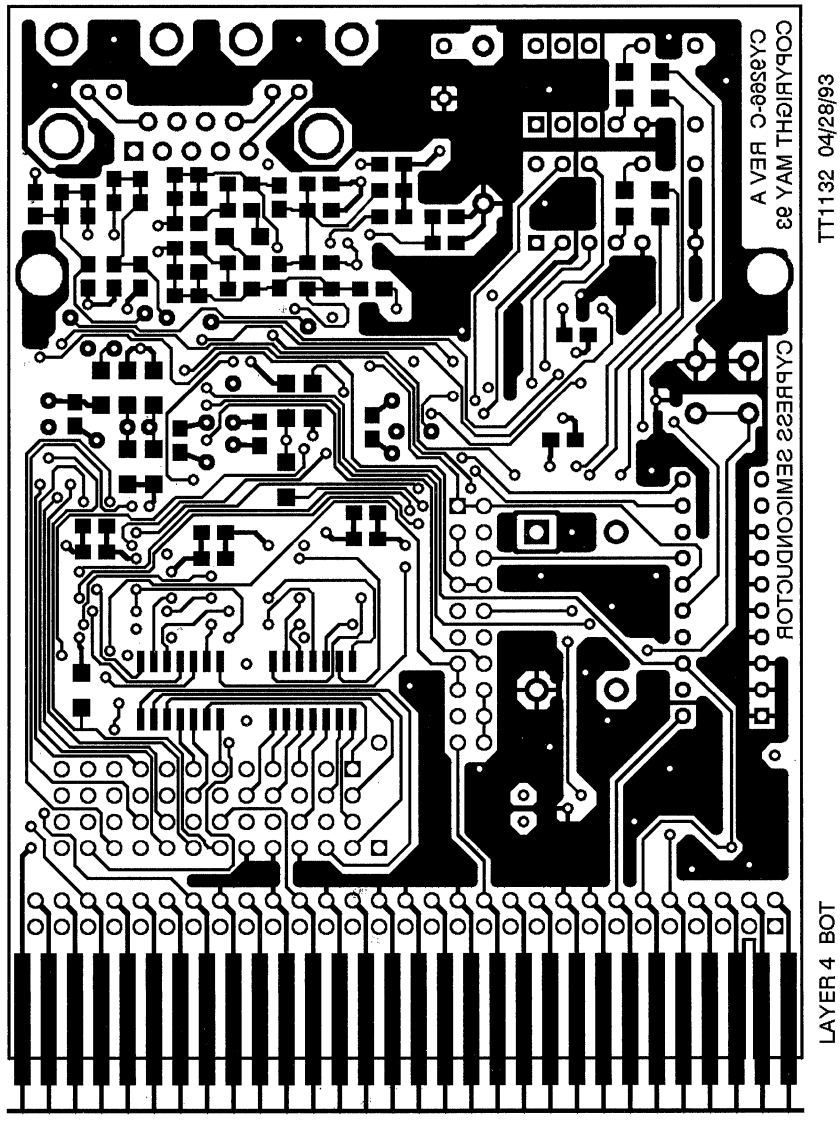
Appendix E. CY9266–C/T Artwork — Power Layer



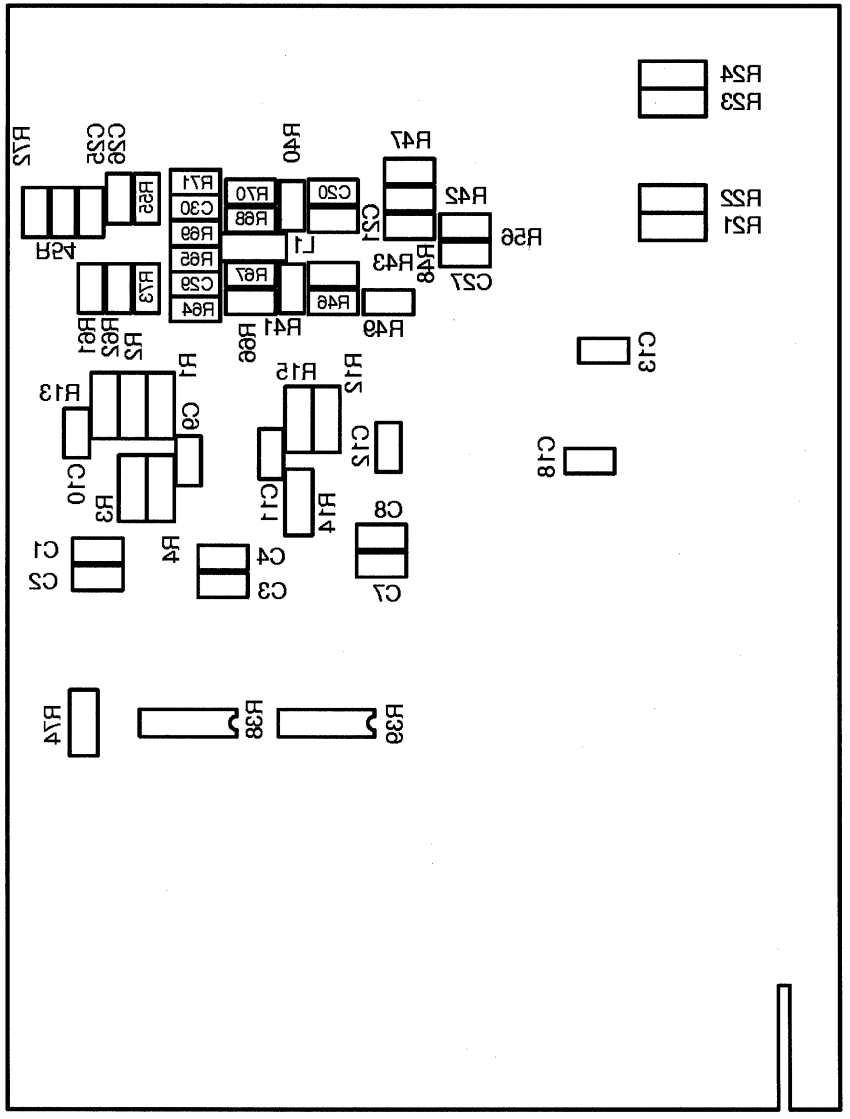
Appendix E. CY9266−C/T Artwork — Ground Layer



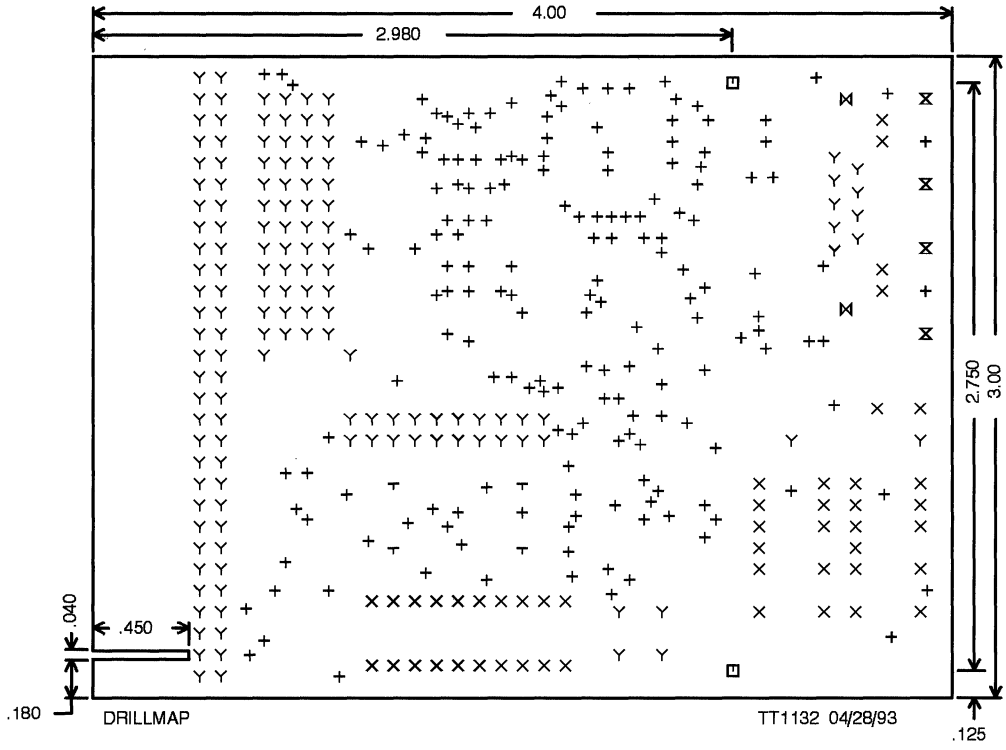
Appendix E. CY9266−C/T Artwork — Bottom Layer Copper



Appendix E. CY9266−C/T Artwork — Bottom Silkscreen



Appendix E. CY9266−C/T Artwork — Drill Chart



TOOL	SIZE	SYM	QTY	PLT
T01	.020	+	179	Y
T02	.032	×	48	Y
T03	.040	Y	143	Y
T04	.052	⊤	4	Y
T05	.080	⧖	4	Y
T06	.125	⋈	2	Y
T06	.156	◳	2	Y

Appendix F. CY9266 Configuration Guide

Function	JP1 Jumper Settings	Switch SW1 Settings (0=on, 1 = off)									
		1	2	3	4	5	6	7	8	9	10
Xmtr BIST Enable *		0									
Xmtr BIST External *		1									
Xmtr Encode Mode *			0								
Xmtr Bypass Mode *			1								
Xmtr ENA Active *				0							
Xmtr ENA External *				1							
Xmtr ENN Active *					0						
Xmtr ENN External *					1						
Rcvr BIST Enable *						0					
Rcvr BIST External *						1					
Rcvr Decode Mode *							0				
Rcvr Bypass Mode *							1				
Rcvr Port A Selected *	CX–CY							1			
Rcvr Port B Selected *								0			
Xmtr Enabled (FOTO Off) *	EX–EY								0		
Xmtr External *									1		
Active High Carrier Detect *										0	
Active Low Carrier Detect *										1	
Active High Byte Sync *							1				1
							0				0
Active Low Byte Sync *							1				0
							0				1
Rcvr Port Select DIP	CX–CY										
Rcvr Port Select External	BX–BY										
FOTO Select DIP	EX–EY										
FOTO Select External	FX–FY										
Xmtr Clock Local Oscillator	GY–HY										
Xmtr Clock XMITCLOCK	GX–GY										
Rcvr Clock Local Oscillator	IX–IY										
Rcvr Clock XMITCLOCK	HX–IX										
Rcvr Clock EXTREFCLK	IX–JX										
OLC–266 Mode	BX-BY, FX-FY, GX-GY, HX-IX	1	1	1	0	1	1			1	1
BIST Mode w/Cable (Standalone)	CX-CY, EX-EY, GY-HY, IX-IY	0		0	1	0		1	0		
BIST Mode wo/Cable (Standalone)	CX-CY, EX-EY, GY-HY, IX-IY	0		0	1	0		0	0		

* – These SW1 controlled signals have a 5.1-kΩ pull-up resistor on the CY9266 card, and may be controlled externally when the SW1 switch is in the off position. With no attached external driver these signals go to a logic-1 state when the SW1 switch position is off.

Timing Products – 7

Timing Products Section Contents and Abstracts

There are many different (and often confusing) terms associated with clock-based devices. This application note attempts to clarify these terms, and hence serves as a comprehensive reference on clock terminology. This application note can be divided into two sections. The first section describes and distinguishes between various clock sources available today. The second section defines and distinguishes between various parameters used to describe clocks. This section also provides methods of measuring some of these parameters.

A PLL-based frequency synthesizer uses a reference input to generate output clocks. The reference can be provided by a quartz crystal or an external clock source. The accuracy and stability of the output clocks in a PLL-based frequency synthesizer are directly proportional to those of the reference. Thus, it is important to provide a stable, accurate, and appropriate reference input. This application note describes the recommended reference inputs for Cypress's PLL-based frequency synthesizers, and concludes with an error budget analysis.

Jitter is extremely important in systems using PLL-based clock drivers. The effects of jitter range from not having any effect on system operation to rendering the system completely non-functional. This application note provides the reader with a clear understanding of jitter in high-speed systems. It introduces the reader to various kinds of jitter in high-speed systems, their causes and their effects, and methods of reducing jitter. This application note will concentrate on jitter in PLL-based frequency synthesizers.

The Cypress Timing Technology products family features ECL-compatible outputs in products such as the ICD2062. These outputs allow clocking at frequencies above 160 MHz, with all the inherent advantages of differential ECL signal transmission. This application note covers the principal advantages of using ECL outputs and makes recommendations concerning layout and wiring methods for parts such as the ICD2062.

The CY2291 and CY2292 are three-PLL frequency synthesizers that utilize EPROM technology. Many different programmable output frequencies and power saving features are contained in one small package. These features result in flexibility and cost savings, as well as short sample and production lead times.

This document begins with an explanation of the CY2291 features. The internal architecture and common applications are then presented. At that point, some recommendations about layout and filtering techniques are made. Finally, the Configuration Request Form is discussed in detail. Although this application note specifically references the CY2291, the information presented also applies to the CY2292.

The CY2254 is a two-PLL clock generator for the Intel Triton™ chipset-based motherboard and other Pentium™ motherboards. It features four high-drive outputs at the CPU clock frequency (50, 60, or 66.66 MHz, selected by two pins), six high-drive synchronous PCI clock outputs at half the frequency of the CPU clocks, two high-drive Reference outputs at 14.318 MHz, a 12-MHz Keyboard clock output, and a 24-MHz Floppy clock output. This application note discusses the internal architecture of the CY2254, and provides recommendations for using it in a system.

The following application note provides a detailed description of the CY7B991 and CY7B992 Programmable Skew Clock Buffers (PSCB). The application note begins with a brief description of clock distribution definitions and solutions. The note follows with RoboClock system design considerations including board decoupling and PCB transmission line analysis, effects, and terminations including actual waveforms and V-I characteristics. A detailed description comes next that explains the device architecture, device configuration, device operation, functional implementations, and a detailed analysis of the AC specifications. The application note ends with a brief AC characterization of the output rise time, fall time, and duty cycle variation.

This application note uses several real world examples of clocking solutions using the RoboClock family of clock buffers. Examples include using RoboClock as a zero propagation delay buffer, using RoboClock as a clock multiplier, gating the output of RoboClock, and using RoboClock as a dynamic phase controlled clock source.

This application note illustrates how the clocks to two Intel 80960CA processors can be synchronized to each other, as well as to an external oscillator, using the "RoboClock" CY7B991. The technique is then extended to n processors using n−1 RoboClocks. One RoboClock is shown driving many processors, which is expedient if either the processors do not have internal Phase-Locked Loops, or if the designer chooses not to use them.

This application note presents a unique application of RoboClock, whose complex and precise waveform generation capability is utilized to implement PWM to enhance color images and increase the resolution of laser printers. The first section of this application note provides a brief description of Roboclock and presents three methods that users could employ to configure it. Then, a brief background on image and resolution enhancement is presented. Finally, the required waveform to implement the image enhancement, and the configuration of Roboclock is presented.

This application note discusses the Test mode capabilities of the CY7B991 and CY7B992 (RoboClock) devices. It begins with an introduction to these devices and then discusses how to use the Test mode features. These features stop the PLL of the device to allow operation in single-step mode while maintaining selected clock output configuration.

Clock Terminology

There are many different (and often confusing) terms associated with clock-based devices. This application note attempts to clarify these terms, and hence serves as a comprehensive reference on clock terminology. This application note can be divided into two sections. The first section describes and distinguishes between various clock sources available today. The second section defines and distinguishes between various parameters used to describe clocks. This section also provides methods of measuring some of these parameters.

Clock Devices

There are a variety of clock devices available today. Some of them are described below.

Crystals

A *Crystal* is a basic piezoelectric quartz crystal. On its own, it cannot generate electrical clocks. It has to be connected to a clock oscillator to get a clock waveform. There are two kinds of crystals; *Series Resonant,* which can be modeled as a high Q series L-C circuit, and *Parallel Resonant,* which can be modeled as a high Q parallel L-C circuit. The series resonant crystal has minimum impedance at the resonating frequency, while the parallel resonant crystal has maximum impedance at the resonating frequency. Cypress-ICD devices expect parallel resonant crystals for the reference device.

Crystal Oscillators

A *Crystal Oscillator* is an oscillator with the crystal as the feedback element. There are other kinds of oscillators with active or passive feedback components, but the crystal oscillator provides the most accurate output frequency.

Crystal oscillators come in a variety of packages, though the 4-pin package (Metal Can Oscillator) in the 300-mil 14-pin DIP footprint is very popular. Surface mount and Half DIP packages are also available. Finally, crystal oscillators are the preferred clock source in most high-speed digital systems requiring clocks.

Compensated Oscillators

The output frequency of a crystal oscillator varies with temperature and voltage. Applications that require a highly stable clock usually use compensated oscillators. *Compensated Oscillators* try to adjust the variation in frequency due to temperature and voltage. *Temperature Compensating Oscillators (TXCO)* contain circuitry that compensates for temperature changes, and hence combat frequency variations. *Oven Controlled Oscillators* encase their crystals in a temperature-controlled oven, and so maintain a precise operating temperature at the crystal. *Double Oven Oscillators* contain two ovens, with the crystal encased in the inner oven, and the temperature control circuitry and the inner oven encased in the outer oven. Such oscillators provide even better temperature stability than Oven Controlled Oscillators. Obviously, as the frequency stability improves, the cost of the oscillator increases.

Voltage Controlled Oscillator

The output of *Voltage Controlled Oscillators (VXCO)* is controlled by a voltage control input pin. Variation between control voltage and frequency is usually nonlinear.

Frequency Synthesizers

Frequency Synthesizers use one or more *Phase-Locked Loops (PLL)* to generate one to many different frequencies on their outputs, from one or more reference sources. The reference frequency is usually generated by a crystal attached to the synthesizer. The design goal of frequency synthesizers is to replace multiple oscillators in a system, and hence reduce board space and cost. *Figure 1* shows a block diagram of a Phase Locked Loop (PLL).

A PLL has two inputs, a reference input and a feedback input. A PLL corrects frequency in two ways. The first, frequency correction, corrects large differences in frequency between the reference input and the feedback input. Frequency correction is akin to "rough" tuning and occurs when F_{vco} is less than $0.5F_{ref}$ or greater than $2F_{ref}$. Phase correction is the "fine" tuning and occurs when $0.5F_{vco} < F_{ref} < 2F_{vco}$.

The Phase/Frequency Detector detects differences in phase and frequency between the reference and feedback inputs and generates compensating "Up" and "Down" signals depending on whether the feedback frequency is lagging or leading the reference frequency respectively. These control signals are then passed through a charge pump and a loop filter to generate a control voltage, which controls a Voltage-Controlled Oscillator (VCO). The frequency of this oscillator is dependent on the V_{ctrl} input. At steady state, the VCO frequency is:

$F_{vco} = F_{ref} * P/Q$

The output frequency of the PLL can be expressed as

$F_{out} = (F_{ref} * P)/(Q * N)$

The *Sample Rate* of a Frequency Synthesizer determines how often the inputs are sampled in order to perform phase and frequency correction. It is expressed as F_{ref}/Q.

The *Acquisition/Lock Time* of a PLL-based Frequency Synthesizer is the amount of time taken by the Frequency Synthesizer to attain the target frequency after power-up, or after a programmed output frequency change.

The *Resolution* of a PLL-based Frequency Synthesizer is based on the number of bits in the P and Q counter. The Resolution will determine in what size increments the frequency can change.

The *Deadband* of a PLL-based Frequency Synthesizer is the largest phase difference between the ref-

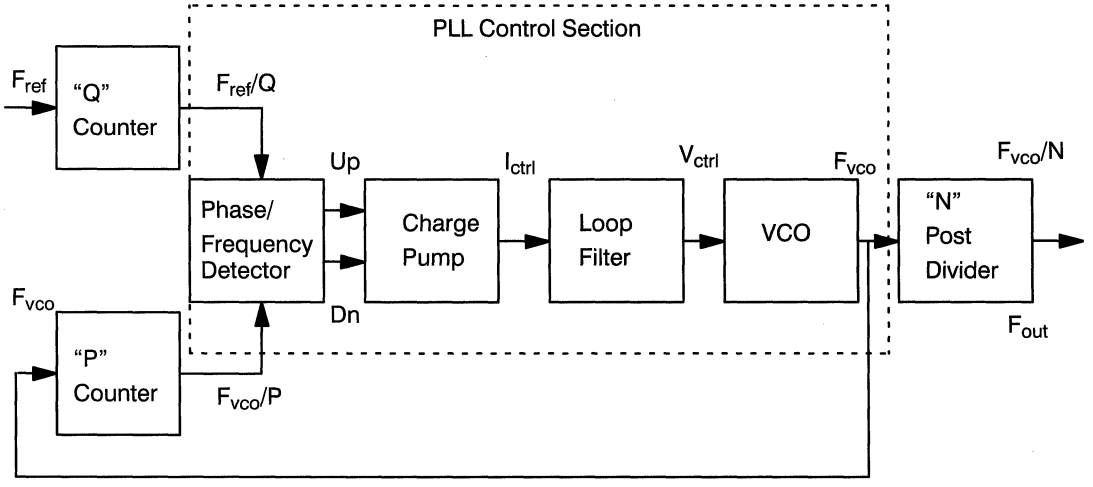


Figure 1. Block Diagram of a Phase Locked Loop

erence and the feedback inputs, which will not be corrected by the PLL.

Multiple PLLs are needed within a single frequency synthesizer to generate multiple unrelated frequencies.

Frequency synthesizers are gaining in popularity as system complexity increases and systems utilize multiple clocks. The term "Clock Generator" is interchangeably used with "Frequency Synthesizer."

Clock Buffers

A *Clock Buffer* is a device in which the output waveform directly follows the input waveform. The input waveform propagates through the device and is redriven by the output buffers. Hence, such devices have a propagation delay associated with them. In addition, due to the differences between the propagation delay through the device on each input-output path, skew will exist on the outputs. An example of a clock buffer is the 74F244, which is available from several manufacturers.

Clock Parameters

This section contains definitions and explanations of various parameters used to describe clocks.

Clock Jitter

Jitter can be defined as the deviations in a clock's output transitions from their ideal positions. The deviation can either be leading or lagging the ideal position. Hence, jitter is expressed in ±ns. Jitter can be classified into three categories: cycle-cycle jitter, period jitter, and long-term jitter.

Cycle-cycle jitter is the difference in a clock's period from one cycle to the next. This kind of jitter is the most difficult to measure and usually requires a Timing Interval Analyzer. *Figure 2* shows a graphical representation of cycle-cycle jitter. J_1 and J_2 are the jitter values measured. The maximum of such values measured over multiple cycles is the maximum cycle-cycle jitter.

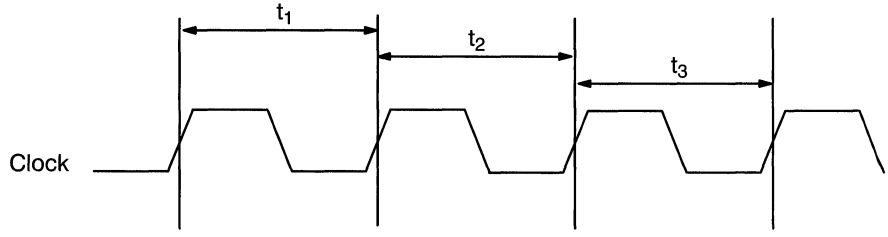
Period jitter, also called short-term jitter, is a change in a clock's output transition from its ideal position over consecutive clock edges. *Figure 3* shows short-term jitter. Note that in the case of short-term jitter, the variation of the rising edge of clock from the ideal position is measured and expressed in units of time or frequency.

Long-term jitter is a change in a clock's output transition from its ideal position, over "many" cycles. The term "many" depends on the application and the frequency. For PC motherboard and graphics applications, this term "many" usually refers to 10–20 microseconds. For other applications, it may be different. *Figure 4* shows a graphical representation of long-term jitter.

Causes of Jitter

There are four primary causes of jitter as indicated below:

- Power supply noise
- The internal PLL of the synthesizer



Jitter $J_1 = t_2 - t_1$
Jitter $J_2 = t_3 - t_2$

Figure 2. Cycle-Cycle Jitter

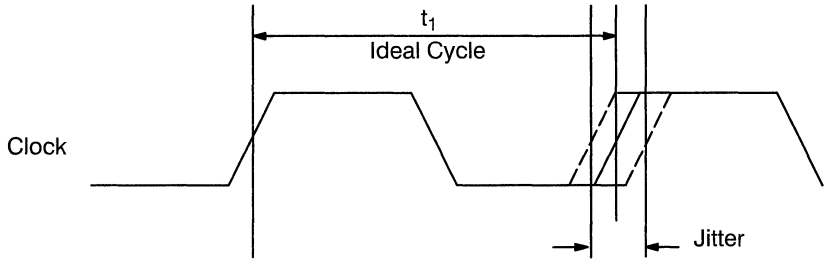


Figure 3. Period Jitter

- Random thermal noise from crystal, or any other resonating device.
- Random mechanical noise from vibrations of the crystal

For a more detailed discussion on jitter, please refer to the application note entitled "Jitter in PLL-Based Systems."

What Systems Does Clock Jitter Affect?

Clock jitter affects almost all high-speed synchronous systems. Common applications affected by jitter are PC motherboards, graphics cards, and communications equipment.

Skew

Skew is the variation in arrival time of two signals specified to arrive at the same time. Skew is composed of two parts, the output skew of the driving device, and board design skew, caused by layout variation of board traces. *Figure 5* explains skew.

Clock Driver Skew (Intrinsic Skew) is the amount of skew caused by the clock driver itself. There are two kinds of clock driver devices; buffer devices and PLL-based devices. Skew occurs on the output of the buffer devices because of the differences in propagation delay of the input signal through the device. A majority of this difference is attributed to differences in output loading. Skew in PLL-based devices can be very small, since a PLL-based device

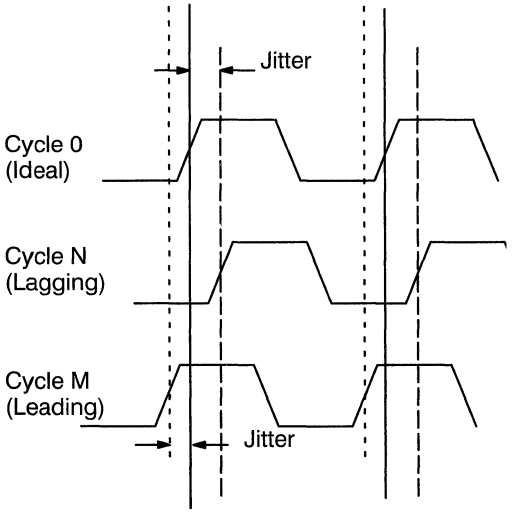


Figure 4. Long-Term Jitter

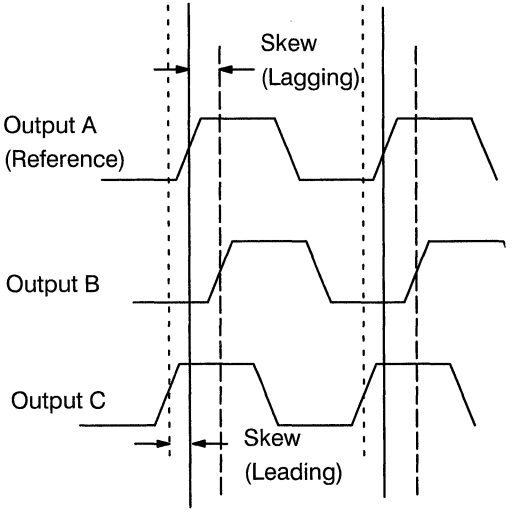


Figure 5. Graphical Representation of Skew

can be adjusted to compensate for differences in output loading.

Board Design Skew (Extrinsic skew) is the amount of skew caused by board layout issues such as:

- Trace Length: The amount of time for a signal to propagate down a trace is dependent on the material of the PCB, length of the trace, width of the trace and capacitive loading. Different trace lengths cause different signal propagation times, and hence cause skew.
- Threshold Voltage Variation: The threshold voltage of the receiving device can cause skew. For example, if a receiving device has a threshold voltage of 1.2V and another device has a threshold voltage of 1.7V, and the rise time of the input signal is 1V/ns, then the two devices will switch 500 ps apart, which is skew.
- Capacitive Loading: The differences in capacitive loading on traces will cause differences in the clock rise times at the load. This affects the time at which the clock edge crosses the input threshold and results in skew.
- Transmission Line Termination: With the extremely fast edge rates in today's clock drivers, traces longer than 4 inches are considered transmission lines. Without proper termination, these lines will exhibit transmission line effects like voltage reflections, which will cause skew.

Why Is Skew Important?

In high-speed systems, clock skew forms an important component of timing margin. A skew of 1 ns is a significant portion of a 15-ns cycle time. If the timing budget does not allow for skew, it is highly likely that the system will perform marginally.

Measuring Skew

The simplest method of measuring skew between two outputs of a device is to display both waveforms in a dual-channel oscilloscope and measure the difference between the rising edges. This is the skew.

Clock buffer datasheets usually specify two parameters, *"output-to-output skew"* and *"part-to-part skew."* The latter parameter includes the former. If neither parameter is specified, then the maximum output skew is the difference between the maximum and minimum propagation delay times through the device.

Stability

Stability is a parameter usually associated with oscillators. Stability is defined as the variation in operating frequency from the nominal frequency and is expressed in ppm (parts per million). The nominal frequency is the frequency shown on the device package.

All variations in frequency are lumped together in the stability specification. Variations in manufacturing processes, aging, temperature, and voltage cause variations in stability. The worst effects are due to temperature variation.

Why Is Stability Important?

Using the stability parameter, a system designer can find the maximum variation in frequency, and hence can design systems based on worst-case specifications. Designing systems without considering stability can cause failure over time.

Aging

Aging is defined as the variation in frequency over time. It is usually expressed in ppm/year, and may be incorporated in the Stability spec, if it is not drawn out separately. It is a parameter usually associated with crystal oscillators. New crystals age faster than old crystals. Typical aging rates are of the order of 5 ppm/yr.

Why Is Aging Important?

Aging may cause marginal operation of a design over an extended period of time, if it is not accounted for in the design.

Voltage Sensitivity

Voltage Sensitivity is the variations in frequency due to variations in operating voltage. It is expressed in ppm/volts. On crystal oscillators, it is usually incorporated in the stability spec. On PLL-based devices, it is usually incorporated in the jitter spec.

Accuracy/Precision

Accuracy/Precision is a measure of how close the part operates to the specified (nominal) frequency.

For example, if a part is specified with a 25.000-MHz output, and the long-term (user-defined) average of its output frequency is 25.001 MHz, the part has 40 ppm accuracy. Accuracy can be expressed as:

Accuracy=(L.T. Avg. Freq. – Nominal Freq.)/Nominal Freq.

Error

On a PLL-based device, it may not always be possible to get the specified frequency on the outputs. The limitation is due to the size of the internal "P" and "Q" counters in the PLL (see later sections for detailed information). If, for example, the specified frequency is 25.000 MHz, and the PLL can output 24.998 MHz, the error is −80 ppm. *Error* can be expressed as:

Error = (Nominal Freq. – Target Freq.)/Target Freq.

Note the difference between error and accuracy. Error specifies the difference between the frequency you want, and the frequency you get. Accuracy specifies the difference between the frequency you get, and the long term average of this frequency.

Slew

The rate of change of voltage or frequency is called *Slew*. Slew is usually measured on the rising and falling edges of digital signals. However, rise times and fall times are more commonly specified, instead of slew, in vendor's catalogs.

Recently, with the advent of low-power devices, slew is being used to define a rate of change of frequency.

Wander/Drift

Wander and *Drift* are the same, and are usually used to express frequency variations due to temperature and voltage. Usually, wander and drift are incorporated in the stability specification.

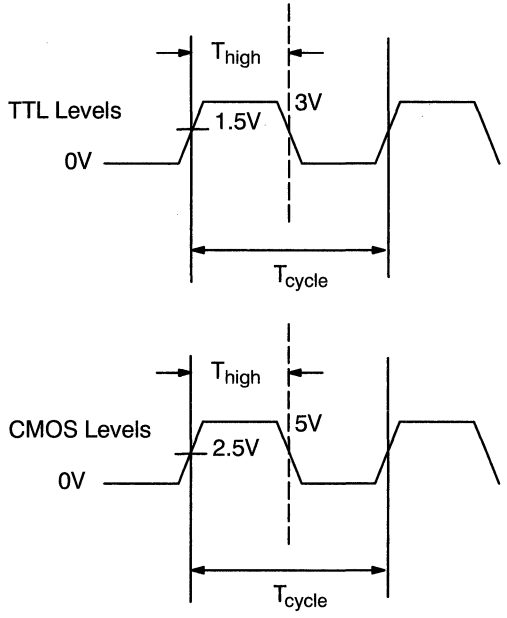


Figure 6. CMOS/TTL Duty Cycle Measurement

Duty Cycle

Duty Cycle is the ratio of the output high time to the total cycle time. It is expressed as a percentage. 50% is the ideal duty cycle, though most clock manufacturers specify duty cycles from 40% – 60%. Duty cycle is important in systems that use both the rising and falling clock edges.

Duty cycles can be expressed for both TTL and CMOS devices. For TTL devices, since the voltage swing is from 0V−3V, the high time is measured at the 1.5V level. For CMOS devices, since the voltage swing is from $0-V_{dd}$ Volts, the high time is measured at $V_{dd}/2$. Hence, if a device claims to meet both CMOS and TTL duty cycle measurements, it refers to the voltage at which the high time is measured, not the output voltage swing. *Figure 6* shows the difference between CMOS and TTL duty cycle measurement levels.

Conclusion

This application note presented clear and detailed descriptions of various clock devices available today, along with parameters used to describe clocks. It also provided methods of measuring some of these parameters.

References

1. Johnson, Howard, and Graham, Martin, *High-Speed Digital Design: A Handbook of Black Magic.* PTR Prentice-Hall. New Jersey, 1993.

Crystal Oscillator Topics

Introduction

A PLL-based frequency synthesizer uses a reference input to generate output clocks. The reference can be provided by a quartz crystal or an external clock source. The accuracy and stability of the output clocks in a PLL-based frequency synthesizer are directly proportional to those of the reference. Thus, it is important to provide a stable, accurate, and appropriate reference input. This application note describes the recommended reference inputs for Cypress's PLL-based frequency synthesizers, and concludes with an error budget analysis.

Please note that this application note does not apply to the ICD6233 (one-time programmable clock oscillator) or the CY7B991/2 and CY7B9910/20 (RoboClock and RoboClock Jr.). For applications assistance on CY7B991/2 and CY7B9910/20, see the application note "Everything You Need to Know About CY7B991/2 (RoboClock) But Were Afraid to Ask."

Cypress's PLL-Based Frequency Synthesizers

Figure 1 shows the block diagram of a typical PLL-based frequency synthesizer. Note that the reference input to all PLLs comes from an on-chip crystal oscillator, which is the architecture of all Cypress clock generators.

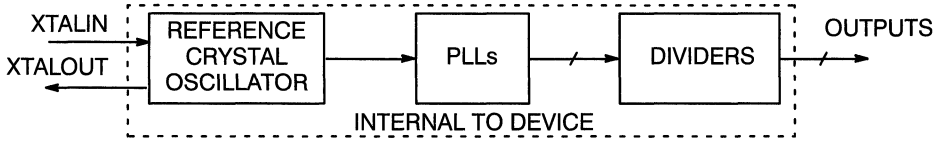


Figure 1. Typical PLL-based Frequency Synthesizer

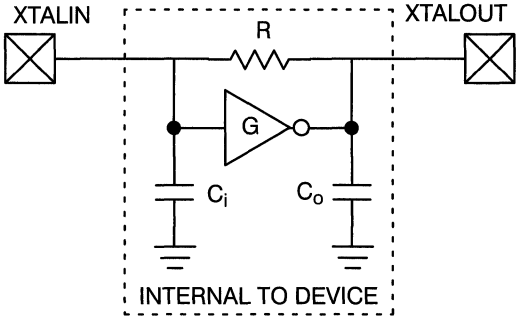


Figure 2. On-Chip Crystal Oscillator Circuitry

Figure 2 shows the circuitry of the on-chip crystal oscillator (a.k.a. Pierce oscillator), which is formed by components R, G, C_i and C_o, where G is a linear inverter. For this circuit to produce an electrical clock, a quartz crystal needs to be connected between the XTALIN and XTALOUT pins.

Crystals Recommended for Cypress Clock Generators

Figure 3 shows the required connection of a crystal to an on-chip oscillator of a PLL-based frequency synthesizer. For best results, a parallel resonant crystal should be used. The load capacitance of this crystal must match the load capacitance of the oscillator circuitry (C_{load}), as seen by the crystal. As shown in *Figure 3*, under normal AC conditions, C_o will be in series with C_{i2}. Thus,

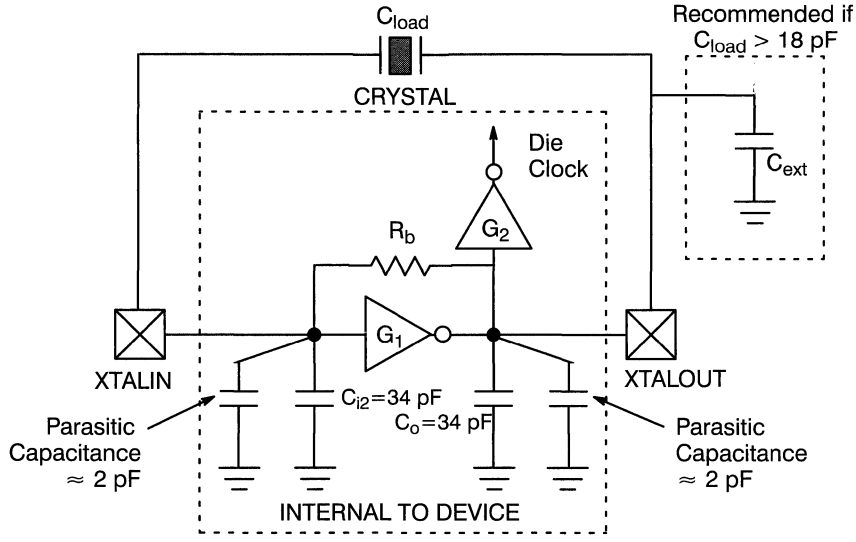


Figure 3. Using a Crystal as Reference

$$C_{load} = \frac{C_o \cdot C_{i2}}{C_o + C_{i2}} \quad \text{Eq. 1}$$

C_{load} = 17 pF. However, if parasitics are accounted for,

$$C_{load} = \frac{C_{oeq} \cdot C_{i2eq}}{C_{oeq} + C_{i2eq}} \quad \text{Eq. 2}$$

where $C_{oeq} = C_o + 2\,pF$, $C_{i2eq} = C_{i2} + 2\,pF$

which results in C_{load} = 18 pF.

Hence, parallel-resonant crystals with C_{load} = 17 to 18 pF should be used for best results with Cypress clock generators. If the C_{load} of the crystal does not equal 17 or 18 pF, the output frequency will be somewhat different from the target. Also, since capacitors C_{i2} and C_o are on-chip, no additional external components are required for operation, provided a crystal with matched C_{load} is used.

A Patch for Crystals with an Unmatched C_{load}

As shown in *Figure 3*, Cypress recommends the addition of an external capacitor, C_{ext}, on or close to the XTALOUT pin to compensate for a C_{load} > 18 pF. C_o and C_{ext} are in parallel, which, under AC conditions, are in series with C_{i2}. Solving the following equation for C_{ext}, which accounts for parasitics,

$$C_{load} = \frac{C_{i2eq} \cdot (C_{oeq} + C_{ext})}{C_{i2eq} + (C_{oeq} + C_{ext})} \quad \text{Eq. 3}$$

gives the value of the external capacitor required. For a crystal with C_{load}=20 pF, C_{ext} = 9 pF would be required.

Note that for C_{load} < 17 pF, solving *Equation 4* (does not account for parasitics) for C_{ext} results in a negative capacitance value.

$$C_{load} = \frac{C_{i2} \cdot (C_o + C_{ext})}{C_{i2} + (C_o + C_{ext})} \quad \text{Eq. 4}$$

Thus, there is no patch available, and the user needs to instead use a crystal with C_{load} = 17 to 18 pF. Using a capacitor in series with the XTALIN or XTALOUT pin will redue the C_{load} seen by the crystal, but will cause start-up problems. This is because the crystal needs to have a DC voltage across it to start oscillations. And if a capacitor is used in series with the XTALIN and XTALOUT pins, this capacitor will block any DC voltage normally applied to the crystal on start-up.

Using a Series Resonant Crystal

In general, using a series resonant crystal with a parallel resonant circuit will introduce an error on the output frequencies of the device. For Cypress's on-chip oscillator, using a series resonant crystal will typically add a 500 ppm (.05%) error on the output frequencies. For some applications, such as time keeping, choosing the right crystal type is crucial. For example, a 50 ppm error in the reference frequency produces a real time clocking error of 2 minutes per month. Thus, the user must ensure that proper crystals are used with Cypress clock generators.

Special Case: 32.768 kHz Crystal

Several of Cypress's clock devices offer internal parallel resonant oscillation circuitry that can produce a 32.768-kHz signal, which is commonly used as a real time clock. Since the internal circuitry does not have a biasing resistor on-chip, a 10-MΩ resistor must be placed in parallel to the 32.768-kHz crystal, as shown in *Figure 4*. Performing the calculations based on *Equation 1* and *Equation 2* results in a crystal requirement of C_{load} = 12 to 13 pF. If the crystal has C_{load} > 13 pF, then a C_{ext}, as calculated from *Equation 3*, is needed. If the C_{load} of the crystal is less than 12 or 13 pF, a capacitor cannot be placed in series with the 32XIN or 32XOUT pin, as explained before.

Using an External Signal Source

Frequently, a frequency synthesizer is driven by an external signal source rather than a crystal. In this case, the external clock should be driven in on the XTALIN pin, and the XTALOUT pin must be left floating. Cypress also recommends using a small coupling capacitor in series with the signal, as shown in *Figure 5*. Such a capacitor provides the benefits of reduced loading of the signal source and restoration of duty cycle, as explained below.

Reduced Loading

As shown in *Figure 5*, the two internal capacitors are each 34 pF. Without the coupling capacitor C_{i1}, the frequency source is effectively driving C_{eff}=34 pF (not accounting for parasitics), where C_{eff} is the effective load capacitance seen by the driver. C_{eff} is reduced by the addition of C_{i1} in series with C_{i2}. Now,

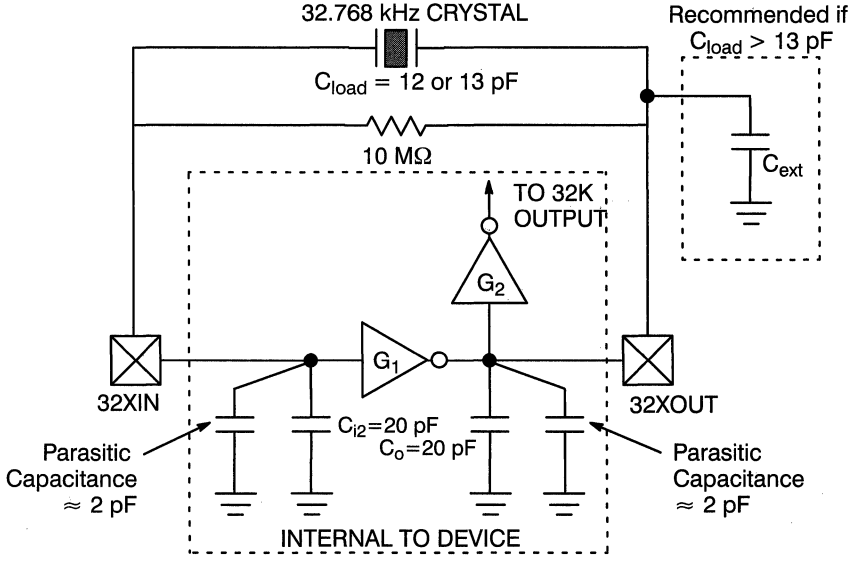


Figure 4. Using a 32.768 kHz Crystal

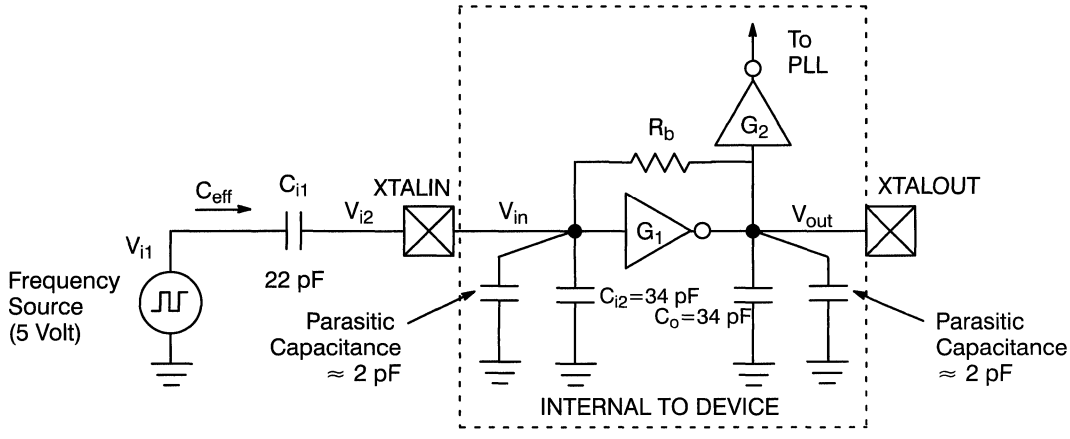


Figure 5. Using an External Driver as Reference

$$C_{eff} = \frac{C_{i1} \cdot C_{i2}}{C_{i1} + C_{i2}} \quad \text{Eq. 5}$$

For example, C_{i1}=22 pF and C_{i2}=34 pF results in C_{eff}=13.4 pF. In this case, C_{eff} is reduced by 62%, which results in reduced loading of the frequency source, reduced power supply noise, and thus improved signal transition times.

While the load is reduced, so is the amplitude of the signal at XTALIN according to the following equation:

$$V_{i2} = V_{i1} \frac{C_{i1}}{C_{i1} + C_{i2}} \quad \text{Eq. 6}$$

Using the same numbers, as in the example above, and setting the input voltage V_{i1}=5V_{pp} results in V_{i2} = 2V_{pp}. However, the reduction in amplitude is not a problem since the linear inverter, G1, helps bias and re-amplify the signal. Specifically, the DC level of V_{in} equals the DC level of V_{out}, and thus the DC level is biased to V_{DD}/2 (CMOS threshold level). Furthermore, the amplifier circuit, consisting of G1 and feedback resistor R_b, results in an AC gain of the signal.

Restoration of Duty Cycle

Typically a waveform at XTALOUT, with a duty cycle of 35-65%, can have the duty cycle restored close to 50%. This restoration can be seen on the output of G2, in *Figure 5*, which is typically the XBUF pin on most devices.

Both the matched characteristics of G1 and G2, and the R-C components work to restore the duty cycle, the mechanism being an AC gain and their effect on DC biasing, as previously mentioned. However, duty cycle regulation is reduced by G1 saturating near V_{DD} or ground. To keep G1 in the linear region, C_{i1} should not be too large. A smaller C_{i1} reduces signal amplitude, thus improving linearity.

Coupling Capacitor Value

For V_{i1}=5V_{pp} applied to a Cypress device, a capacitor value of C_{i1}=22 to 24 pF, placed as close to the XTALIN pin as possible, is recommended. Using C_{i1}=22 to 24 pF provides 2V_{pp} around an average DC level of V_{DD}/2 at XTALIN, as well as reduced loading and restored duty cycle.

Cypress clock generators require V_{i2}=2V_{pp}. Thus for 5V input signal (V_{i1}=5V_{pp}), V_{i2}=2V_{pp}, and C_{i2}=34 pF, solving *Equation 5* results in $C_{i1} \approx$ 22 pF. Accounting for parasitics by substituting C_{i2eq}=36 pF for C_{i2}=34 pF, the result is C_{i1}=24 pF.

For a 3.3V input signal (V_{i1}=3.3V_{pp}), V_{i2}=2V_{pp}, and C_{i2}=34 pF, solving *Equation 5* results in $C_{i1} \approx$52 pF. Accounting for parasitics results in $C_{i1} \approx$55 pF.

General Error Budget Analysis

As in any good design, an error budget should be calculated. Several sources of error must be taken into account.

- Reference source frequency tolerance (ppm); specified by manufacturer of reference
- Reference source temperature tolerance (ppm); specified by manufacturer of reference
- Crystal Oscillator process variation (ppm); specified by clock chip manufacturer
- Crystal Oscillator supply tolerance (ppm); specified by clock chip manufacturer
- Crystal Oscillator temperature tolerance (ppm); specified by clock chip manufacturer

Two methods of budgeting can be done.

- Addition of the relevant sources of error
- The well respected Monte Carlo Analysis, which states that if a number of uncorrelated variables are changing randomly, it is not reasonable to add up the individual worst-case figures to calculate an aggregate worst-case value.

The following example uses typical error values for crystals and Cypress clock devices. The first method of budgeting results in a total error of ±94 ppm.

Example: Addition of Relevant Sources of Error

Source of Error	Error in ppm
Reference Source, Crystal	
Frequency tolerance	±50 ppm
Temperature tolerance	±30 ppm
Crystal Oscillator in Cypress Clock Generator	
Process Variation	±05 ppm
Supply Tolerance	±03 ppm
Temperature Tolerance	±06 ppm
Total	±94 ppm

Using the same values, the Monte Carlo Analysis results in a much lower total error, as shown below. If there are n variables $X_1, X_2, \ldots, X_n$, all varying randomly and independently, then the overall variation is:

$$X_{total} = \sqrt{X_1^2 + X_2^2 + \ldots + X_n^2} \qquad \text{Eq. 7}$$

This results in a total error of ±59 ppm.

In general, if we compare the first method with the second, the first will always yield a higher result, as long as $X_1, X_2, \ldots, X_n$ are either all positive or negative numbers. Stated mathematically,

$$X_1 + X_2 + \ldots + X_n > \sqrt{X_1^2 + X_2^2 + \ldots + X_n^2} \qquad \text{Eq. 8}$$

for all $X_1, X_2, \ldots, X_n > 0$ or all $X_1, X_2, \ldots, X_n < 0$.

Summary

In summary, Cypress recommends the following for our clock generators. For designs that use a crystal for the input reference, the crystal should be parallel resonant, and have C_{load} = 17 to 18 pF. If C_{load} > 18 pF, then use an external capacitor, as shown in *Figure 3*, with C_{ext} calculated from *Equation 3*. If C_{load} < 17 pF, then instead use a crystal with C_{load} = 17 to 18 pF.

For designs using the 32.768-kHz circuitry, a parallel resonant crystal with C_{load} = 12 to 13 pF must be used. A 10-MΩ biasing resistor must be placed in parallel with the crystal.

5V designs using an external clock source must AC couple the clock input with a 22- to 24-pF capacitor in series with the clock source. 3.3V designs should use a 52- to 55-pF coupling capacitor instead.

For layout recommendations on Cypress clock devices, please read the application note: "Jitter in PLL-Based Systems: Causes, Effects, and Solutions," and, if available, the application note corresponding to the specific device.

Jitter in PLL-Based Systems: Causes, Effects, and Solutions

Jitter is extremely important in systems using PLL-based clock drivers. The effects of jitter range from not having any effect on system operation to rendering the system completely non-functional. This application note provides the reader with a clear understanding of jitter in high-speed systems. It introduces the reader to various kinds of jitter in high-speed systems, their causes and their effects, and methods of reducing jitter. This application note will concentrate on jitter in PLL-based frequency synthesizers.

What is a PLL-Based Frequency Synthesizer ?

Frequency Synthesizers use one or more *Phase-Locked Loops (PLL)* to generate one to many different frequencies on their outputs, from one or more reference sources. The reference frequency is usually generated by a crystal attached to the synthesizer. It is rarely generated from an external oscillator. The design goal of frequency synthesizers is to replace multiple oscillators in a system, and hence reduce board space and cost. *Figure 1* shows a block diagram of a Phase-Locked Loop (PLL).

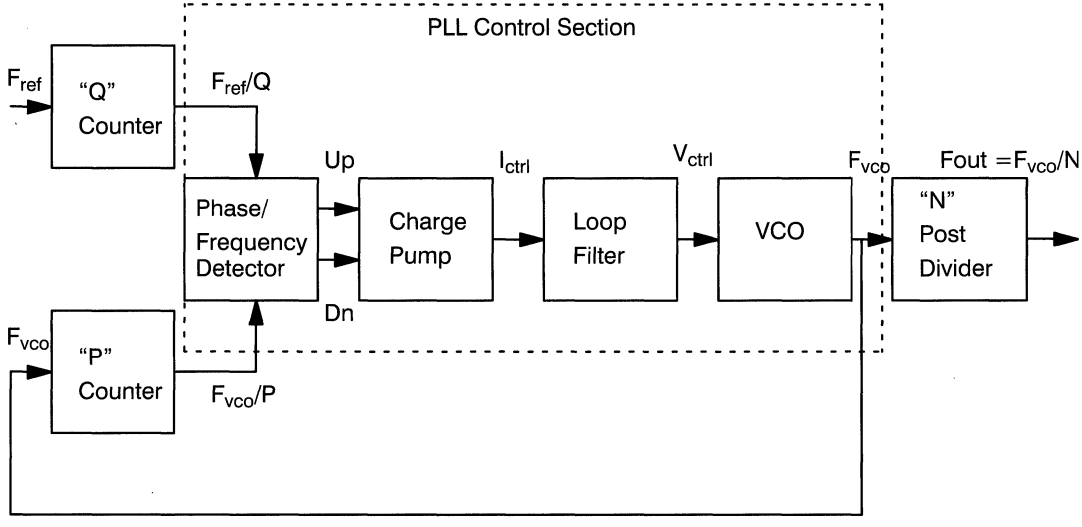


Figure 1. Block Diagram of a Phase-Locked Loop

A PLL has two inputs: a reference input, and a feedback input. A PLL corrects frequency in two ways. The first, frequency correction, corrects large differences in frequency between the reference input and the feedback input. Frequency correction is activated when the input frequency is changing significantly, or when the device is powered up. Frequency correction is the "rough" tuning of the PLL. "Fine" tuning occurs when phase correction is activated.

The Phase/Frequency Detector detects differences in phase and frequency between the reference and feedback inputs and generates compensating "Up" and "Down" signals. The pulsewidth of the "Up" signal is greater than the "Down" signal, if the feedback input frequency is less than the reference frequency, and vice versa. These control signals are then passed through a charge pump and a loop filter, to generate a control voltage, which feeds into a Voltage-Controlled Oscillator (VCO). The frequency of this oscillator is dependent on the V_{ctrl} input. At steady state, the VCO frequency is:

$F_{vco} = F_{ref} * P/Q$

The output frequency of the PLL can be expressed as

$F_{out} = (F_{ref} * P)/(Q * N)$

where

— F_{vco} = VCO Frequency

— F_{ref} = Reference Frequency

— P = Multiplier, lies in feedback path

— Q = Divider, lies in reference path

— N = Post Divider

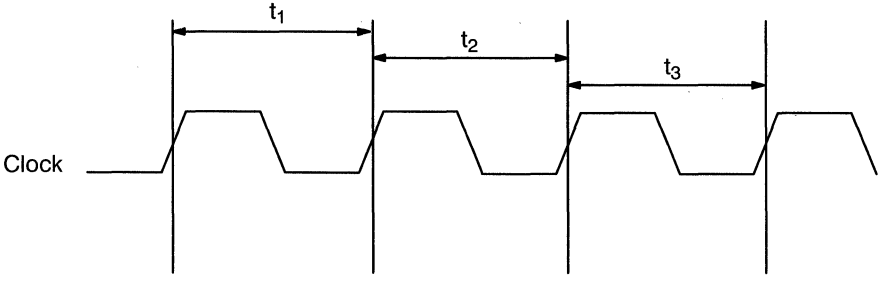
Clock Jitter

Jitter can be defined as the deviations in a clock's output transitions from their ideal positions. The deviation can either be leading or lagging the ideal position. Hence, jitter is sometimes specified in ±ps. Jitter is also specified in other units, like a percentage of frequency, or absolute value, in ns. Jitter measurements can be classified into three categories: cycle-cycle jitter, period jitter, and long-term jitter. Additionally, all jitter measurements are made at a specified voltage.

Cycle-Cycle Jitter

Cycle-cycle jitter is the change in a clock's output transition from its corresponding position in the previous cycle. This kind of jitter is the most difficult to measure and usually requires a Timing Interval Analyzer. *Figure 2* shows a graphical representation of cycle-cycle jitter. J_1 and J_2 are the jitter values measured. The maximum of such values measured over multiple cycles is the maximum cycle-cycle jitter.

Until recently, cycle-cycle jitter was not particularly meaningful in most cases. However, like the incorporation of PLLs in CPUs (e.g., the 486 and the Pentium™ processors), cycle-to-cycle jitter has taken on new significance. Consider the case shown in *Figure 3* where the output of one PLL_1 is the reference of



Jitter $J_1 = t_2 - t_1$

Jitter $J_2 = t_3 - t_2$

Figure 2. Cycle-Cycle Jitter

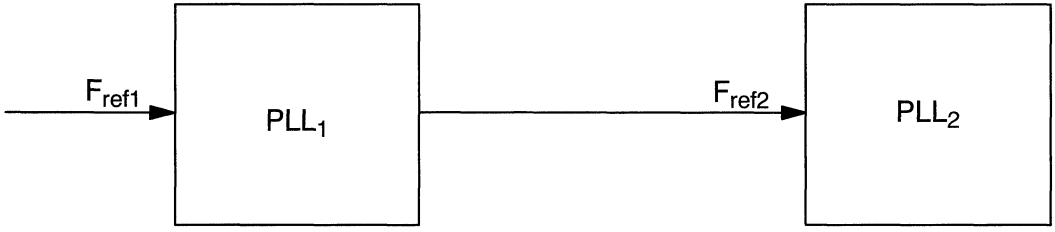


Figure 3. Application for Cycle-Cycle Jitter Measurement

PLL_2. In this case, if PLL_2 cannot lock to the reference frequency, the cycle-cycle jitter of the output of PLL_1 will have exceeded the maximum jitter allowable for PLL_2 to lock. If PLL_1 is the clock generator for PLL_2 embedded in the CPU, the output jitter of PLL_1 must be sufficiently low to successfully time the inputs to PLL_2.

Period Jitter

Period jitter measures the maximum change in a clock's output transition from its ideal position. *Figure 4* shows period jitter.

Period jitter measurements are used to calculate timing margins in systems. Consider, for example, a microprocessor-based system in which the processor requires 2 ns of data set-up time. Assume that the clock driving the microprocessor has a *maximum* of 2.5 ns period jitter. In this case, the rising edge of clock *can* occur before data is valid on the data bus. Hence, the processor will be presented with incorrect data, and the system will not operate. This example is illustrated in *Figure 5*. The system designer needs to take period jitter into account while designing the system.

Long-Term Jitter

Long-term jitter measures the maximum change in a clock's output transition from its ideal position, over many cycles. The term "many" depends on the application and the frequency. For PC motherboard and graphics applications, this term "many" usually refers to 10–20 microseconds. For other applications, it may be different. *Figure 6* shows a graphical representation of long-term jitter.

A classic example of a system affected by long-term jitter is a graphics card driving a CRT. Assume that a pixel of data is meant for the pixel at co-ordinates (10,24) on the CRT. Because of a jittery clock, this data may drive a pixel at location (11,28) on the CRT. Over an extended period of time, the data meant for pixel (10,24) may be driving a pixel far away from its ideal (10,24) location. Since this effect of a jittery clock is usually consistent over all pixels, the overall effect of a jittery clock is to cause an image to shift from its ideal display position on screen. This effect is sometimes called "running" of the screen.

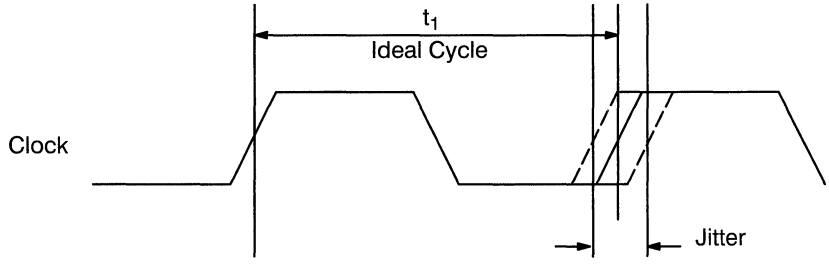


Figure 4. Period Jitter

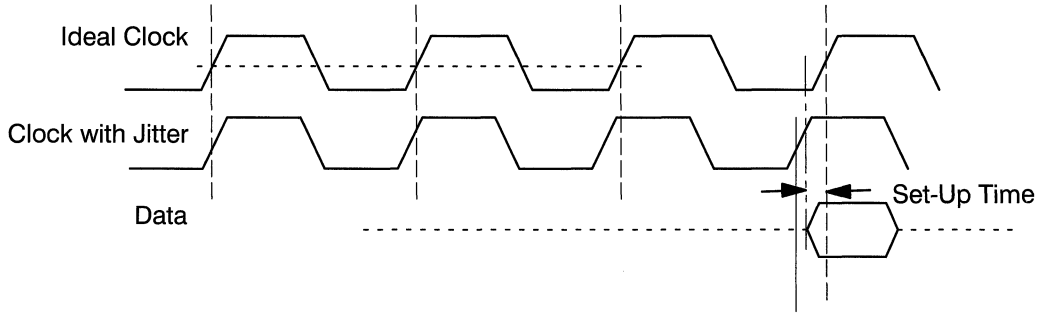


Figure 5. Application for Period Jitter Measurement

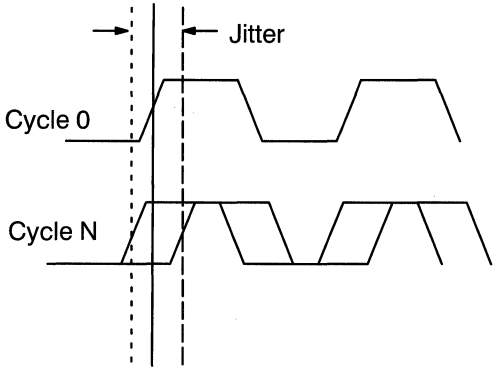


Figure 6. Long-Term Jitter

Causes of Jitter

There are four primary causes of jitter as indicated below in decreasing order of importance.

- Power supply noise on a PLL's supply inputs, which appears on the output as jitter. This is the largest, though not always constant, contributor to jitter. Power supply noise manifests itself through various ways, some of which are:
  - — *Ground Bounce:* When there is a surge of current through the output drivers, the inductance of the leads to the supply planes (V_{cc} and GND) have a voltage drop across it (value = L.di/dt). This raises or lowers the effective ground potential of the device. Hence, if the output frequency is dependent on the effective supply voltage, this frequency will change because of ground bounce. Second, the threshold voltage of transistors within the oscillator changes, which causes a change in frequency. This has a twofold effect. First, the output frequency changes. Second, if the oscillator feeds a PLL, this PLL tries to correct the change in frequency. Both of these effects appear on the outputs as jitter.
  - — *V_{dd} Noise: Figure 7* shows an inverter in the internal counter of the PLL. The threshold voltage of the input is half the V_{dd} potential. Assume for example, that the V_{dd} signal has a 100-mV p-p noise ripple associated with it. This noise will cause a shift in the threshold voltage at the input of the inverter. The change in the triggering level of this inverter will cause jitter. If this noise signal has a rise time of 1 V/ns, then 100 ps of peak-peak jitter will appear on the outputs of the inverter, due to the 100-mV p-p ripple voltage.

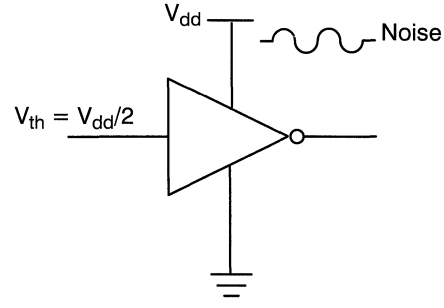


Figure 7. Effect of V_{dd} Noise on Jitter

- The PLL in a frequency synthesizer has a dead-band associated with it, during which the phase and frequency detector does not detect small changes in the input phase. Since these changes are not detected, they do not get corrected and appear on the outputs in the form of jitter.
- Random thermal noise from the crystal reference, or any other resonating device.
- Random mechanical noise from vibrations of the crystal reference.

Measuring Jitter

Since we have defined three kinds of jitter, we will propose three methods of measuring them.

Cycle-Cycle Jitter

Measuring cycle-cycle jitter is extremely difficult. A Timing Interval Analyzer (TIA) is required to perform this measurement. In this case, the output of the jittery clock is connected to a TIA, and the measurement to be specified is the difference of time periods of consecutive clock cycles. The maximum of this difference over multiple cycles is the cycle-cycle jitter.

Period Jitter

A simple method of measuring period jitter requires a storage oscilloscope. Set the trigger for the rising edge of clock. Then scroll the display to the next rising edge of the clock and turn on the persistence. If the scope is set up correctly, the width of the blurring on the displayed transition will indicate the amount of period jitter in the clock. An example of period jitter measurement is shown in *Figure 2*. The peak in the horizontal histogram indicates the fundamental frequency, while the spreading around this frequency shows the jitter.

Long-Term Jitter

Long-term jitter is probably the easiest to measure. It uses a measuring technique called *differential phase measurement*. The jittery clock is connected to an oscilloscope with a delayed time-base feature. The scope is set to trigger on the rising edge of clock. Then, using the delayed time-base feature, the same clock waveform is displayed on the screen.

To make sure that the scope calibration and characteristics can perform the jitter measurement, measure the output of a stable clock source, like a crystal oscillator. If the waveform has no blurs or bands, the scope can correctly measure long-term jitter.

Methods of Reducing Jitter

As discussed before, two major causes of jitter are power supply noise and ground bounce. Reducing the power supply noise and eliminating ground bounce will reduce most of the jitter in a system.

Reducing Power Supply Noise

Power supply noise can be reduced by bypassing and filtering the power supply appropriately.

Bypassing, by using a large tantalum capacitor (10−1000 µF) *attached to the board power supply*, will prevent a fall in voltage caused by current surges, as well as reduce power supply ripple. Attach this capacitor as close as possible to where the V_{dd} and GND signals enter the PCB.

This large capacitor will, however, be ineffective at very high frequencies. Hence, a small capacitor, 0.1 µF, will be required to *filter* high-frequency noise. Cypress recommends attaching a 0.1-µF ceramic capacitor *on every V_{dd} pin* of the frequency synthesizer. These capacitors must be attached as close to the pins as is physically possible. Surface mount capacitors are preferred because of their low lead inductance.

If the part has separate analog and digital power supply pins, use a 22Ω resistor in series with a 22-µF capacitor to ground to filter low-frequency noise. Using a smaller capacitor in parallel with the 22-µF will ensure better attenuation.

Finally, using a regulated power supply (such as from a 3-pin regulator, or a Zener diode), with the above bypassing and filtering techniques, will ensure better power supply rejection.

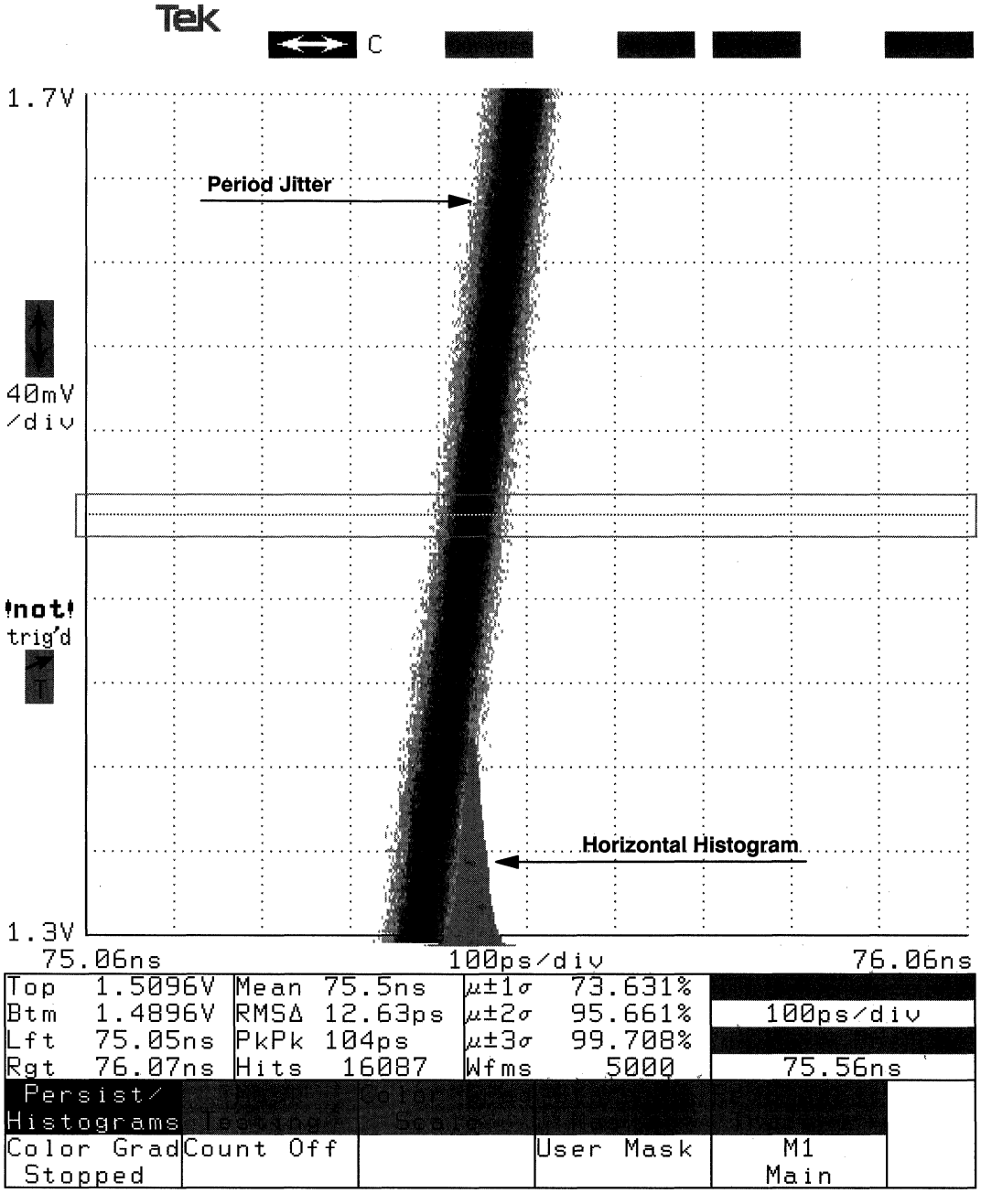


Figure 8. An Example of Period Jitter Measurement

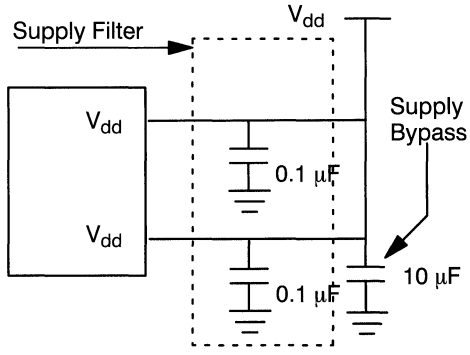


Figure 9. Power Supply Noise Filter Circuit

Figure 9 shows a circuit which can be used to reduce power supply noise for clock generators with multiple digital power supplies, such as the CY2254. In case the clock generator has separate analog and digital power supplies, such as the ICD2028, use the circuit shown in *Figure 10*.

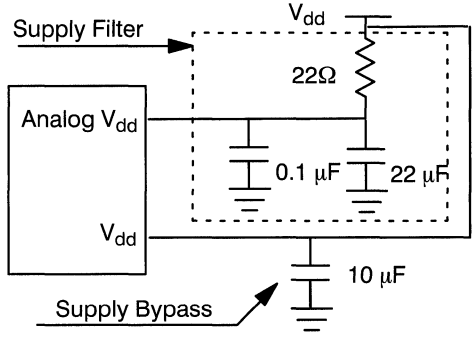


Figure 10. Power Supply Noise Filter Circuit

In addition to using power supply bypass and filtering techniques, avoid routing any high-frequency signals below the clock generator. This will minimize noise-coupling effects, and will result in reduced jitter on the outputs of the clock generator.

Eliminating Ground Bounce

Ground bounce can be eliminated in three ways. The first is to reduce the number of loads on the output of the device. A second method of reducing ground bounce is to provide large ground planes on your PCB. Finally, if you have two or more ground pins, connect them *individually* to the ground plane, instead of shorting them together. The third way is to install a series resistance on the output pins. This will limit the output current and reduce ground bounce.

Conclusion

This application note has discussed the various jitter measurements which can be made on a system. It also discussed the causes and effects of jitter, and presented techniques for reducing jitter in PLL-based systems. Using this information the reader should be able to design more reliable high-speed systems.

References

1. High Speed Digital Design, A Handbook of Black Magic, *Howard Johnson & Martin Graham,* 1993 Prentice-Hall, Inc.

Pentium is a trademark of Intel Corporation.

ECL Outputs

Introduction

The Cypress Timing Technology products family features ECL-compatible outputs in products such as the ICD2062. These outputs allow clocking at frequencies above 160 MHz, with all the inherent advantages of differential ECL signal transmission.

This application note covers the principal advantages of using ECL outputs and makes recommendations concerning layout and wiring methods for parts such as the ICD2062.

Power Supplies (PECL vs. ECL)

The ECL V_{DD} and V_{EE} pins have traditionally been powered from a −5.2V supply, V_{DD} being grounded and V_{EE} set at −5.2V—the intent is to achieve the lowest V_{DD} noise by grounding the V_{DD} pins. In more recent designs, however, ECL is often used with +5.0V instead of −5.2V. (V_{DD} set to + 5.0V and V_{EE} tied to ground.) Since V_{DD} noise is not a major concern, this permits the use of a standard logic supply. This application note will focus on +5.0V ECL designs (sometimes called PECL).

ECL Advantages

As clock speeds rise beyond 100 MHz, the advantages of using ECL become more obvious. Most of these advantages involve the use of differential signal transmission.

Differential signals are less susceptible to ground noise problems, as all noise becomes common-mode. Single-ended CMOS is much more susceptible, since ground bounce and other noise affect logic thresholds, degrading noise immunity. ECL signals remain unaffected, since noise rides on both signals as an average level. Logic levels are also less critical, since the threshold is the differential cross point, which can tolerate significant signal attenuation. Differential circuits also tend to generate less noise in the power supply.

ECL is designed with termination resistors that allow high-frequency signals to propagate with minimal overshoot and reflection.

These advantages are most pronounced in a bipolar implementation, but many of the same benefits can be realized in CMOS designs.

Pad Structure

Referring to *Figure 1*, transistors Q1 and Q2 form differential ECL output drivers. Unlike single-ended outputs (see *Figure 2*), N-type transistors are not required, since termination resistors are always present and serve as pull-downs.

The ECL output drive logic guarantees that when Q1 switches ON, Q2 switches OFF (and vice versa). A complementary logic state is always maintained, assuring a constant current supply draw in either output state.

Logic Levels

The V_{OH}/V_{OL} logic levels are approximately 4.1V to 3.2V. This gives a differential signal of 0.9 V. This is more than adequate, since bipolar ECL V_{OH}/V_{OL} are typically 4.1V to 3.3V for a differential swing of 0.8V. If the termination resistors are reduced from the 220Ω/330Ω suggested value, the V_{OH} level can be reduced somewhat.

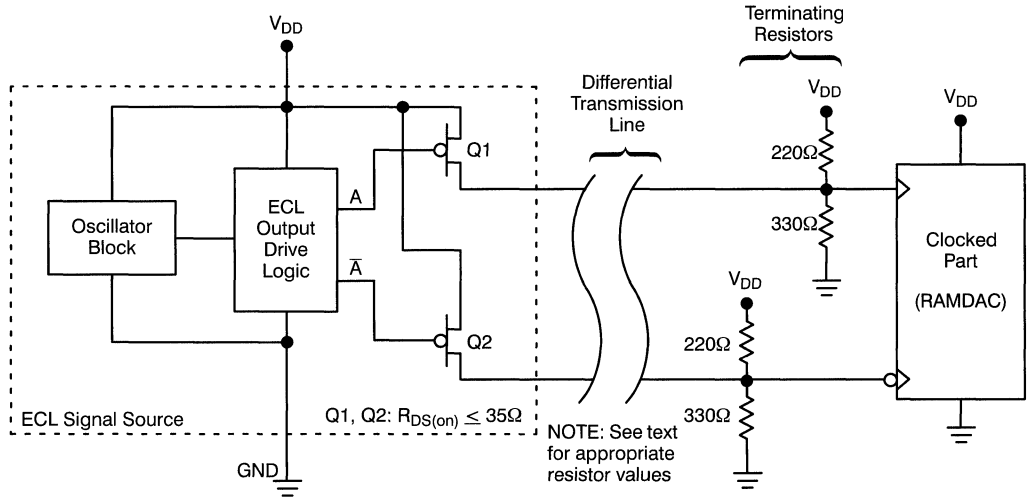


Figure 1. Differential ECL Output Driver

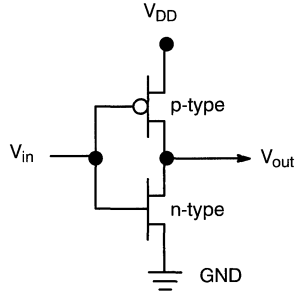


Figure 2. Single-Ended Output Driver

Output Routing and Board Layout Issues

ECL signals maintain their integrity over long routing distances. A 100-MHz single-ended trace should be limited to a few inches, but ECL can travel several feet at that frequency. This is due to ECL's termination resistors at the receiving devices.

For good signal integrity, ECL traces are laid out in pairs. By controlling the board layout and specifying appropriate electrical properties, a constant characteristic impedance of 100Ω to 150Ω can be achieved. By employing transmission live techniques, high-quality signals are ensured.

To reduce V_{DD} noise the clock generator must be properly bypassed at the supply pins, as should the parts to be clocked. Terminating resistors should also be bypassed separately if they are not located near a bypass capacitor.

Terminating Resistor Values

The 220Ω/330Ω values represent a trade-off between low-power draw and best high-frequency performance. For frequencies above 80 MHz, they may pull down too weakly, resulting in inadequate signal swing. These values can go as low as 68Ω/100Ω at 160 MHz (always maintaining an approximate 2:3 ratio). Low values work satisfactorily at both low and high frequencies, the only drawback being a higher current drain.

Summary

A general overview of ECL logic has been presented, with emphasis on the interface of ECL logic to +5V powered CMOS clock generators. Should additional support be required, contact Cypress Applications for assistance.

Understanding the CY2291 and CY2292

Abstract

The CY2291 and CY2292 are three-PLL frequency synthesizers that utilize EPROM technology. Many different programmable output frequencies and power saving features are contained in one small package. These features result in flexibility and cost savings, as well as short sample and production lead times.

This document begins with an explanation of the CY2291 features. The internal architecture and common applications are then presented. At that point, some recommendations about layout and filtering techniques are made. Finally, the Configuration Request Form is discussed in detail.

Although this application note specifically references the CY2291, the information presented also applies to the CY2292. The only differences are that the CY2292 comes in a 16-pin SOIC package (32XIN, 32XOUT, 32K, and V_{BATT} are absent) and the FLOPPYCLK output has been replaced with a GND pin.

CY2291 Features

The CY2291 has eight output clocks (four are configurable), smooth slewing on outputs originating from the CPU PLL, power-saving features, low skew between related outputs, and user-selectable reference support. Each of these functions is discussed in more detail below. *Figure 1* shows the logic block diagram of the CY2291.

Multiple Outputs

The CY2291 has eight output pins, enabling it to support almost all PC motherboard clock requirements. These outputs consist of four user configurable clocks, a CPUCLK, a FLOPPYCLK, a XBUF and 32-kHz clock output. Each of these outputs is explained in more detail in the CY2291 Internal Architecture section of this application note. When any output is in a three-state condition, the signal is pulled LOW because the CY2291 has weak pull-downs on all outputs (except 32K). This is to ensure compatibility with Pentium™-based systems.

Variable Reference Frequency

The default reference frequency for the CY2291 is 14.318 MHz. However, the part can accept any reference frequency between 10 MHz and 25 MHz, preferably from an accurate, stable, parallel-resonant crystal. In addition, this reference crystal does not require any external resistors or capacitors.

Alternatively, the CY2291 can use an external reference clock of frequency between 1 MHz and 30 MHz. In this case, the external reference clock is driven in over the XTALIN pin and the XTALOUT pin is left floating. The duty cycle of this input clock should be between 40% and 60% measured at $V_{DD}/2$. For more information on AC-coupling the external reference clock, please refer to the application note "Crystal Oscillator Topics."

Smooth Slewing

The CY2291 provides smooth slewing on outputs originating from the CPU PLL. The term "smooth slewing" refers to frequency slewing (the rate of change of frequency with respect to time). Specifically, the frequency of such an output changes smoothly and monotonically from 4 MHz to 80 MHz for 3.3V operation and up to 100 MHz at 5V.

Smooth slewing is required for processors such as the 486, which accept only a limited amount of frequency change per clock cycle.

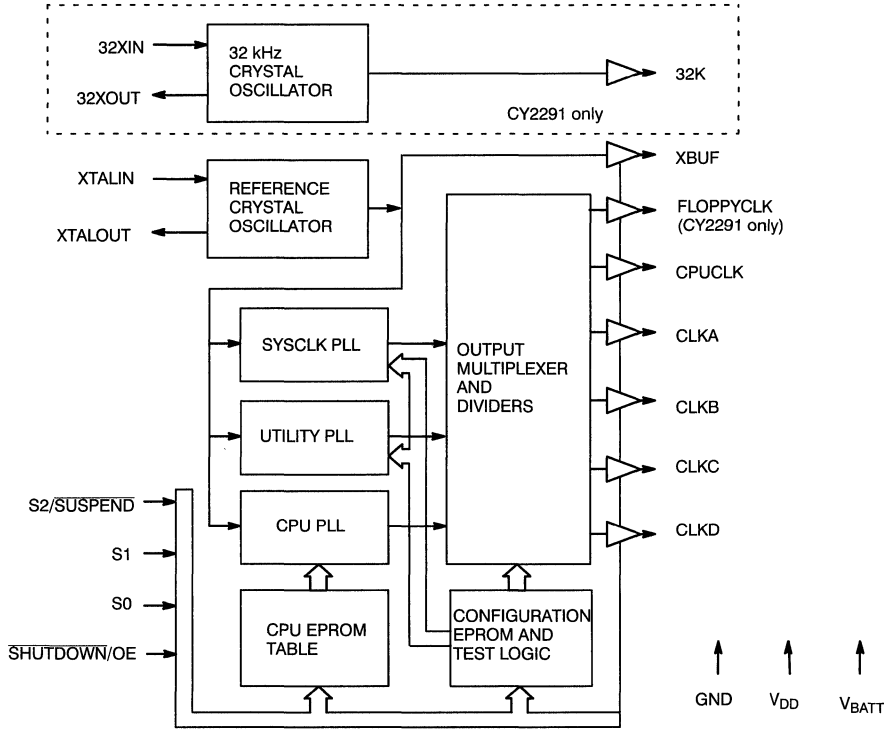


Figure 1. Block Diagram of CY2291/2

Power-Saving Modes

The CY2291 features a variety of power-saving modes, which are especially useful in Green PC and laptop applications.

Suspend Mode

The suspend option allows the user to activate and deactivate selected resources at will. The suspend feature must be requested and the resources-to-be-suspended selected when ordering the part.

Each of the three PLLs and each of the outputs, except for the 32.768-kHz output, can be suspended independently. Suspending a PLL shuts down all associated logic including counters and downstream post- dividers, and places related outputs in a three-state condition. Suspending an output simply forces a three-state condition on the output. Moreover, transitioning from the suspend to active state requires the PLLs to re-lock (50 ms maximum, 5 ms typical).

Suspend mode is controlled by the S2/$\overline{\text{SUSPEND}}$ pin (active LOW). In this power-saving mode, the CPU PLL, unless also suspended, will output a frequency corresponding to a selection with S2 = 0. If the suspend option is disabled (i.e., not implemented during configuration) the S2/$\overline{\text{SUSPEND}}$ pin is used solely as select input for the CPUCLK output.

Shutdown Mode

The shutdown option allows the user to activate and deactivate the entire CY2291 chip at will, using the $\overline{\text{SHUTDOWN}}$/OE pin (active LOW). During shutdown, the current draw of the CY2291 is reduced to less than 65 μA (50 μA if 32-kHz oscillator is not used). The shutdown option must be specified when requesting the part.

In shutdown mode, all outputs (32K output not affected) are three-stated. All PLLs, associated logic,

ROMs, counters, Reference Oscillator, and any other active components are shut down.

Transitioning from the shutdown to active state requires the PLLs to re-lock (50 ms maximum, 25 ms typical). In addition, because the SHUTDOWN/OE pin has no pull-up resistors, the user must drive this pin to a voltage level for proper operation.

The "Off" Option

The "off" option allows permanent shutdown of selected resources, independent of the suspend and shutdown options. Selecting "off" for a PLL permanently shuts down the PLL and all associated logic, and three-states associated outputs. Selecting "off" for an output simply three-states the output.

Unlike the suspend and shutdown modes:

- The "off" mode does not power-down the entire part.
- The "off" mode is pin-controllable only for the CPU PLL.

Low Skew

The CY2291 has low skew (500 ps maximum) between related signals on CLKA–CLKD, and CPUCLK outputs. Referring to Table 1, related signals are defined as those which are on the same row (except the "Ref" row). Therefore, SPLL/3 and SPLL/6 are related, but SPLL/3 and SPLL/24 are not.

In addition, the outputs must have identical capacitive loads to meet the skew specifications.

CY2291 Internal Architecture

In addition to a dedicated 32-kHz output, the CY2291 uses three internal PLLs, EPROM technology, and a reference crystal oscillator to synthesize up to four unrelated frequencies. These frequencies are then divided, using post-dividers, allowing the device to provide up to a total of eight different outputs.

The internal architecture of the CY2291 is explained in more detail below.

Phase Locked Loop

In general, frequency synthesizers use one or more PLLs to generate one to many different frequencies. The CY2291 can generate up to four unrelated frequencies: CPLL, SPLL, UPLL, and one buffered reference frequency, where CPLL is the frequency generated by the CPU PLL, SPLL by the SYSCLK PLL, and the UPLL by the UTILITY PLL. For more information on PLLs, see the application note "Jitter in PLL-Based Systems."

EPROM Technology

Using factory-programmable EPROM technology provides two advantages to the customer:

- Instead of relying on the manufacturer's available ROM options, the customer can order a custom set of frequencies on the CPUCLK output.
- Factory-programmable EPROM technology enables fast turnaround times on the product. The customer no longer needs to wait six weeks for a custom mask set to be created, or for the part to be fabricated. Typical turnaround times are less than one week.

The CY2291 is controlled by two factory programmable EPROMs. The CPU EPROM, which is a ROM table, controls the operation of the CPU PLL. Input pins S[2:0] allow the user to select the desired output frequency of CPUCLK. The Configuration EPROM contains information to configure the SYSCLK and UTILITY PLLs, as well as output frequencies and suspend selected resources.

Outputs

The three internal PLLs allow the CY2291 to offer numerous frequencies on its eight outputs.

32.768 kHz

The on-chip 32-kHz circuitry is electrically isolated from the rest of the device. Activating any of the power-saving modes will not affect the buffered output (32K). To generate the 32 kHz output, connect:

- A 32-kHz reference crystal between pins 1 and 20
- A 10 MΩ resistor in parallel to the 32 kHz crystal as shown in *Figure 2*
- The VBATT pin to the battery operated supply

If the 32-kHz output is not required, either leave pins 1, 2, 19, and 20 floating, or consider using the CY2292.

XBUF

This output is a buffered copy of the reference oscillator.

FLOPPYCLK

The SYSCLK PLL is usually configured to an output of SPLL=96 MHz. The user has a choice of dividing SPLL by two, three, or four. In this particular example, the FLOPPYCLK output can be either 24, 32, or 48 MHz.

CPUCLK

This output, generated by the CPU PLL, is user-selectable. The user selects one frequency from the CPU EPROM, which is factory programmed with the configuration desired by the user. The usage of pins S[2:0] to select a CPUCLK frequency differs in the following three situations.

- *If the CY2291 is factory-programmed without the Suspend feature*, then the S2/$\overline{\text{SUSPEND}}$ pin is dedicated solely to selecting one of the eight CPUCLK frequencies.
- *If the CY2291 is factory-programmed with the Suspend feature, but not on the CPU PLL*, then the S2/$\overline{\text{SUSPEND}}$ pin is used to control the suspend feature *and* the CPU PLL frequency selection. However, in suspend-mode, the CPUCLK output will be a frequency that corresponds S2 = LOW. Specifically while in suspend-mode, the CPUCLK can only output one of four frequencies.
- *If the CY2291 is factory-programmed with the Suspend feature on the CPU PLL*, then the S2/$\overline{\text{SUSPEND}}$ pin is reserved solely for suspend-mode (active LOW). Thus, when S[2:0] = 0XX, the CPU PLL will be suspended. When the CY2291 is not in suspend-mode, the user sets S[1:0] (with S2 = HIGH) to select one of four CPUCLK frequencies.

Configurable Outputs

The outputs CLKA–CLKD can each be set to one of 32 selections. This palette of choices is generated by the output multiplexers and dividers. Internal signals that originate from the PLLs and the reference crystal oscillator, are further divided, resulting in 32 output possibilities as shown in *Table 1*.

Table 1. CLKA–CLKD Palette of Choices.

Ref	Ref/2	Ref/4	Ref/8
CPLL	CPLL/2	CPLL/4	CPLL/8
UPLL	UPLL/2	UPLL/4	UPLL/8
SPLL	SPLL/2	SPLL/4	SPLL/8
SPLL/3	SPLL/6	SPLL/12	OFF
SPLL/5	SPLL/10	SPLL/20	SPLL/40
SPLL/12	SPLL/24	SPLL/48	SPLL/96
SPLL/13	SPLL/26	SPLL/52	SPLL/104

Within the 32 output choices, the SPLL/12 option occurs twice, thus allowing up to 31 unique output selections. In addition, for CLKD, the Ref/8 frequency is replaced with a Ref/3 option.

Note that if any one of the configurable clocks (CLKA–CLKD) is obtained from the CPU PLL, that clock will exhibit the same characteristics as the CPUCLK.

Layout and Filtering Techniques

In order to ensure optimal operation of the CY2291, use the following layout and filtering techniques. *Figure 2* shows the recommended external connections.

Series Terminations

If the output of the CY2291 drives multiple loads or long traces, use a terminating resistor in series with the output, attached as close to the output pin as is possible. *Figure 2* shows a 22Ω resistor in series with the output. The value of this resistor, summed with the output impedance of the CY2291, should equal the characteristic impedance of the trace (transmission line). Typical values of the series resistor range from 10Ω to 75Ω.

A resistor in series with the output dampens the voltage reflections that occur with output imped-

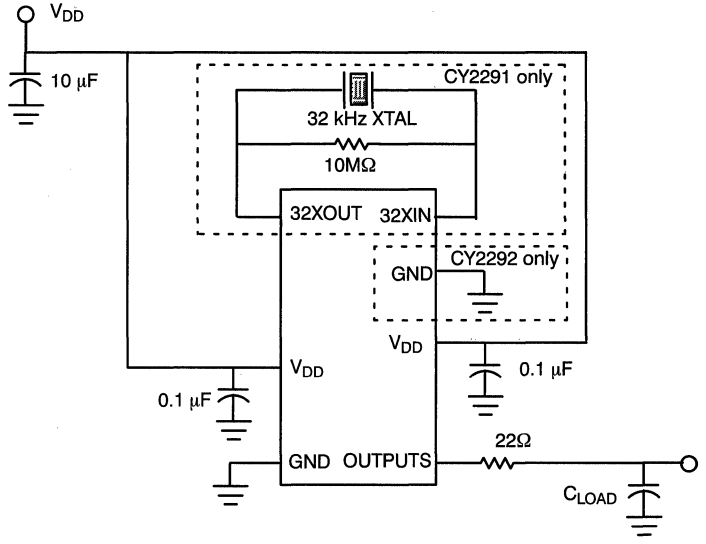


Figure 2. External Connections of the CY2291/2

ance mismatches. It has the ultimate effect of reducing jitter on the output of the CY2291.

Layout Guidelines

The following guidelines apply for laying out the CY2291 on a board:

- Provide a large ground plane under the device. This will have the effect of reducing ground bounce in the system, thus reducing jitter.
- Connect each GND pin to the ground plane individually. Connecting them together, and then to the plane will defeat the purpose of providing multiple ground pins.
- Avoid routing any high-frequency or clock signals below the device. Placing the device in a relatively quiet area of the board will reduce noise coupling into the PLL, and will ensure lower jitter on the outputs.

All the above recommendations, along with a stable power supply source, will result in significantly reduced jitter on the clock outputs.

Typical Applications

The CY2291 is an extremely versatile device. It can be used in PC, printer, and other embedded applications.

Personal Computers

The CY2291 can provide the multiple frequencies, smooth slewing, and power-saving features to help computer manufacturers meet the Green PC requirement.

Desktop and Notebook PCs Using 486 Processors from Intel, AMD, or Cyrix

The CY2291 is an excellent choice for 486 motherboards. The CPUCLK output is designed to slew smoothly, meeting the 486 requirements.

Desktop PCs based on the Pentium Processor

The CY2291 can provide the multiple clock frequencies required by a highly integrated Pentium motherboard. For example, a Pentium PC motherboard may require:

- 40 MHz for SCSI
- 24 MHz for Floppy
- 12 MHz for Keyboard

- 14.318 MHz for Interrupt Controller
- Pin-strappable CPU clock frequencies of 50, 60, 66.66, and 75 MHz.
- CPU/2 for PCI clock frequencies with low skew to CPU.

The CY2291 can provide all the above frequencies.

Notebook PCs based on the Pentium Processor

The programmable power-saving features of the CY2291 (suspend and shutdown options), are extremely useful in notebooks, which need to conserve power. When configured correctly, this device can utilize up to three power reducing options: "off", suspend, and shutdown.

Printers (Networked and Desktop)

Desktop and Network printers each require multiple frequencies to drive the serial port, parallel port, Ethernet port, CPU and ASICs. Such printers may require:

- 1.8432 MHz for serial port
- 20 MHz for Ethernet port (network printers only)
- 25 MHz for Centronics Parallel port
- 33 MHz for CPU

All frequencies are provided by a CY2291. In addition, the smooth slewing on the outputs, as well as the power-saving features of the CY2291, allow manufacturers to design printers that can power-down when idle.

Upgrade for the ICD2028

The CY2291 is pin-compatible with the ICD2028 and offers higher performance with respect to jitter and skew. Users upgrading to the CY2291 should note that the device has no internal pull-up or pull-down resistors on any inputs. Depending on how these inputs are driven, the CY2291 requires external pull-down resistors on the select lines (S[2:0]) and a pull-up resistor on the OE pin.

Other Uses

Any application that requires clocks to be generated from a single device can use the CY2291. Applications include game systems, scanners, copiers, and mass storage devices.

Guidelines for the Configuration Request Form

Before placing orders for the CY2291, a configuration request form (shown at the end of the application note) needs to be completed. From this, Cypress can correctly program the EPROMs. When filling out the request form, please follow the directions and note the guidelines below.

Guidelines

Operating Voltage

Select either 3.3V or 5V operation.

Input Reference Frequency

Specify a frequency between 10 MHz and 25 MHz if using an external crystal. When using an external reference clock, specify a value between 1 MHz and 30 MHz.

PLL Frequencies

Fill in the desired frequencies in the *Requested* column. The requested CPLL, UPLL, and SPLL frequencies should be greater than 8 MHz, and not more than 100 MHz for 5V operation (80 MHz for 3.3V operation).

If the suspend option is desired on the CPU PLL, then only request CPLL frequencies corresponding to S2 = 1.

Output Configuration

Select one frequency for each output, using the Output Options Table on the form. In addition, fill in the corresponding frequency value, as a double-check. Please follow the constraints specified in parenthesis by each output. Note that for the CLKD output, Ref/8 is replaced with a Ref/3 frequency.

Shutdown Option

If the shutdown option is chosen, this will enable the user to power down the entire CY2291 with pin 18. If this option is not desired, then pin 18 will be used as an output enable (OE).

Suspend Option

Two ways to specify suspending an output are:

- Suspend the associated PLL.
- Suspend the output directly.

Recall that suspending a PLL powers down all associated logic and outputs, as well as the PLL, when in this mode. Turning a PLL "off" permanently shuts down the associated logic and outputs, as well as the PLL.

Other Guidelines

If no outputs are associated with a PLL, then select "off" for the PLL. If an output is not needed, select "off" for the output. Implementing these suggestions will reduce power consumption.

A word of caution: PLLs running at the same or integer-multiple frequencies of each other will cause harmonics to appear at associated outputs. To avoid this, do not select PLL frequencies to be equal or integer-multiples of each other.

Pentium is a trademark of Intel Corporation.

Summary

In summary, this application note explains the features, internal architecture, potential applications, and the configuration request form of the CY2291.

This space-saving part offers flexibility and power-down features, cost savings, and fast turnaround times. The three PLLs support variable frequencies at the outputs. The low jitter and smooth slewing of the CY2291 provides the accuracy needed by today's high speed applications. The power saving modes allow designs to meet benchmarks, such as the Green PC requirement. Most importantly, the CY2291 utilizes EPROM technology, which allows for customized frequencies and short sample and production lead times, helping customers to meet their design schedules.

CY2291/2 CUSTOM CONFIGURATION REQUEST FORM

Company _____ Engineer _____ FAE/Sales _____

Phone # _____ Fax # _____ Date _____

CIRCLE ONE **CY2291** **CY2292**

The CY2291 and CY2292 are the industry's most flexible frequency synthesizers, offering a high degree of configurability due to their unique internal factory-programmable EPROM array. Of the CY2291/2's outputs, six (five on the CY2292) may be defined within the scope of the PLL frequencies and divider criteria described in the following. The process may require several iterations to achieve the desired frequencies. Shaded areas are for Cypress use only. Contact your local Cypress representative for assistance.

1. **OPERATING VOLTAGE** (Circle one) **3.3V** **5.0V**

2. **INPUT REFERENCE FREQUENCY** (Circle one) **Crystal** **External Clock** **14.31818 MHz** (Default)
   If a different reference is required, specify the frequency in the box to the right (must be between 10 MHz and 25 MHz for crystal, 1 MHz and 30 MHz for external clock): _____

3. **CPU-PLL FREQUENCIES** ("Off" is a valid selection for any address and will automatically be entered for blanks.)

Select	Requested	Actual
000		
001		
010		
011		
100		
101		
110		
111		

Range: 8–100 MHz at 5V; 8–80 MHz at 3.3V

If the Suspend Option is specified in #7 below, the Select MSB (S2) serves a dual function as both the MSB CPU address and as the Suspend select pin. The CPU frequencies specified for addresses 000–011 will be active unless the CPU-PLL is shut down during the suspend mode (CPU-PLL is circled in #7). Also, any outputs derived from a non-suspended CPU-PLL (assigned in #5 as options 5–8) that are not circled in #7 will remain active during the suspend mode.

4. **UTILITY-PLL AND SYSTEM-PLL FREQUENCIES** ("Off" is a valid frequency selection for either PLL.)
   To minimize harmonic effects, avoid setting any PLL to an equal or multiple frequency of another PLL.

	Requested	Actual
U-PLL		

Range: 8–100 MHz at 5V; 8–80 MHz at 3.3V

	Requested	Actual
S-PLL		

Range: 8–100 MHz at 5V; 8–80 MHz at 3.3V
Default = 96 MHz at 5V; 48 MHz at 3.3V

5. **OUTPUT CONFIGURATION** ("Off" is a valid selection for any output and will automatically be entered for blanks.)
   Assign by number from the Output Options Table below and fill in the Frequency column as a double-check.

Output Options Table

1. Ref	6. CPLL/2	11. UPLL/4	16. SPLL/4	21. SPLL/12	26. SPLL/40
2. Ref/2	7. CPLL/4	12. UPLL/8	17. SPLL/5	22. SPLL/13	27. SPLL/48
3. Ref/4	8. CPLL/8	13. SPLL	18. SPLL/6	23. SPLL/20	28. SPLL/52
4. Ref/8	9. UPLL	14. SPLL/2	19. SPLL/8	24. SPLL/24	29. SPLL/96
5. CPLL	10. UPLL/2	15. SPLL/3	20. SPLL/10	25. SPLL/26	30. SPLL/104

	Option	Frequency
32K (Fixed 32 kHz)	–	32.768 kHz
FLOPPYCLK (Options 14–16, Off)		
XBUF (Option 1 only)	1	
CPUCLK (Options 5–7, Off)		

32K and FLOPPYCLK are not available on the CY2292.

	Option	Frequency
CLKA (Options 1–30, Off)		
CLKB (Options 1–30, Off)		
CLKC (Options 1–30, Off)		
CLKD (Options 1–30, Off)		

For CLKD only: option #4 (Ref/8) is replaced with Ref/3.

6. **SHUTDOWN OPTION** (Circle Yes or No) **Yes** **No**

7. **SUSPEND OPTION** (Circle Yes or No) **Yes** **No**

   IF SUSPEND = "Yes": Circle each resource to be shut down when the Suspend mode is active (S2=0). Note that suspending a PLL automatically suspends its outputs.

   CPU-PLL XBUF CLKA
   UTIL-PLL CPUCLK CLKB
   SYS-PLL FLOPPYCLK CLKC
   CLKD

FOR CYPRESS USE ONLY (Shaded areas above and below)

Customer Configuration	Marking
Date	Quantity

Understanding the CY2254

Introduction

The CY2254 is a two-PLL clock generator for the Intel Triton™ chipset-based motherboard and other Pentium™ motherboards. It features four high-drive outputs at the CPU clock frequency (50, 60, or 66.66 MHz, selected by two pins), six high-drive synchronous PCI clock outputs at half the frequency of the CPU clocks, two high-drive Reference outputs at 14.318 MHz, a 12-MHz Keyboard clock output, and a 24-MHz Floppy clock output. This application note discusses the internal architecture of the CY2254, and provides recommendations for using it in a system.

CY2254 Features

The logic block diagram of the CY2254 is shown in *Figure 1*. The device accepts input from a 14.318-MHz parallel-resonant crystal. This signal is then fed to the two internal PLLs, which generate the required frequencies on the outputs. All clock outputs are controlled by an active-HIGH Output Enable pin, which three-states the outputs when

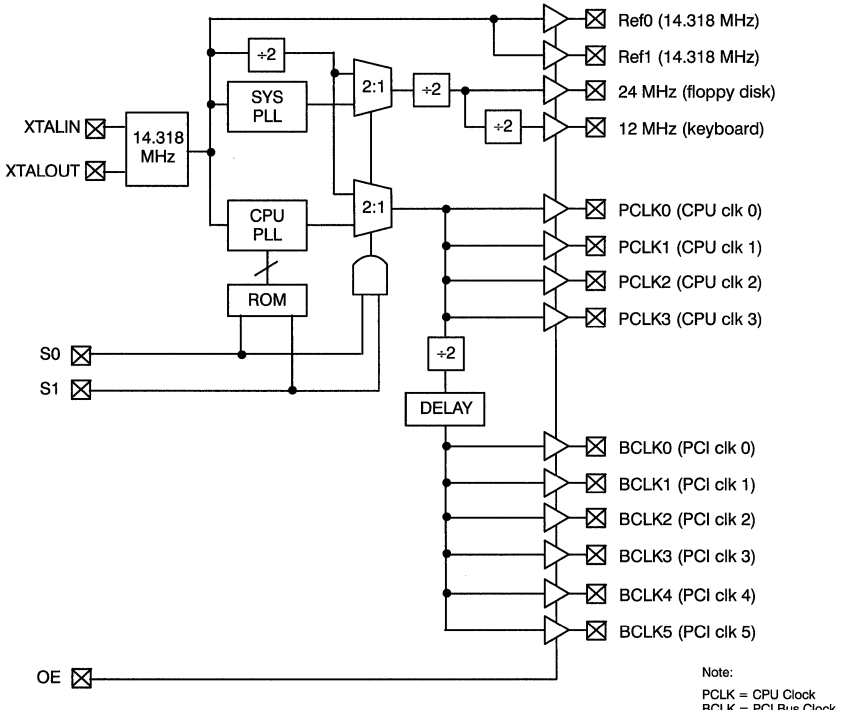


Figure 1. CY2254 Logic Block Diagram

deasserted. *Table 1* shows the CY2254 function table.

CPU Clock Outputs

The CY2254 CPU PLL generates four outputs at the CPU clock frequencies of 50, 60 or 66.66 MHz, by applying appropriate levels on the select inputs, S0 and S1. All CPU clock outputs meet the Pentium's maximum cycle-cycle jitter specification of 200 ps. Finally, all CPU clock outputs are skew-controlled, with a maximum skew of 250 ps between them.

PCI Clock Outputs

The CPU PLL output, after being internally divided by two, drives six outputs at one-half the CPU clock frequency. In addition, these PCI clock outputs lag the CPU clock outputs by 1 to 5 ns. Finally, all PCI clocks outputs are skew-controlled, with a maximum skew of 500 ps between them.

Both CPU and PCI clock outputs feature:

- Matched impedances on the rising and falling edges of output drivers resulting in equal rise and fall times
- Low output impedance: 25Ω (typical) and 40Ω (maximum), measured at 1.5V
- Max. Load on CPU clock = 20 pF
- Max. Load on PCI clock = 30 pF

Reference Clock Outputs

The CY2254 buffers two clock outputs at the reference frequency of 14.318 MHz. The REF0 output has a larger drive capability of 30 pF, as compared to the REF1 drive capability of 15 pF.

Keyboard and Floppy Clocks

The SYS PLL generates the Keyboard and Floppy clocks at 12 and 24 MHz respectively. Both these outputs are capable of driving 20-pF loads, with low jitter.

Test Mode Support

The CY2254 supports the Triton Test mode when both S1 and S0 are set to a logic HIGH. In this mode, reference frequency (TCLK), applied to the XTALIN input, is buffered onto the REF0 and REF1 outputs. The CY2254 also generates TCLK/2, TCLK/4, TCLK/4, TCLK/8 frequencies on the CPU, PCI, Floppy, and Keyboard clock outputs respectively.

Power Supply

The CY2254 requires a clean and accurate 3.3-Volt (±5%) power supply for proper operation.

Reference Frequency

Cypress recommends the use of a parallel-resonant 14.318-MHz crystal to generate the most accurate clock outputs. A series-resonant crystal will result in clock outputs of a slightly higher frequency (appx. 5%).

System Applications

The CY2254 was primarily designed to meet the clock requirements of the Intel Triton chipset. However, since it is a Pentium-compatible device, it can be used in any motherboard requiring high-drive CPU and PCI clock outputs.

Table 1. CY2254 Function Table

OE	S0	S1	XTALIN Input	PCLK	BCLK	Ref. Clock Output	24 MHz	12 MHz
0	X	X	14.318 MHz	High-Z	High-Z	High-Z	High-Z	High-Z
1	0	0	14.318 MHz	50 MHz	PCLK/2	14.318 MHz	24 MHz	12 MHz
1	0	1	14.318 MHz	60 MHz	PCLK/2	14.318 MHz	24 MHz	12 MHz
1	1	0	14.318 MHz	66 MHz	PCLK/2	14.318 MHz	24 MHz	12 MHz
1	1	1	TCLK	TCLK/2	TCLK/4	TCLK	TCLK/4	TCLK/8

The CY2254 will provide accurate, low-jitter clocks on its output. To ensure the quality of the clock outputs, a noise-free power supply is necessary. Additionally, the user should follow the high-speed design techniques summarized in the following sections to ensure reliable operation of the CY2254 and the board. For more details on these techniques, please refer to the Application Notes, "System Design Considerations" and "Protection, Decoupling, and Filtering of Cypress CMOS Circuits," both of which are available in the latest edition of the Cypress Applications Handbook. Please contact your local Cypress representative for a copy.

Supply Bypass and Filtering

To ensure low jitter on the outputs of the CY2254, the designer must provide a clean source of power. A large tantalum capacitor (10–1000 µF) *attached to the board power supply*, will prevent a fall in voltage caused by current surges, as well as reduce power supply ripple. Attach this capacitor as close as possible to where the V_{dd} and GND signals enter the PCB.

This large capacitor will, however, be ineffective at very high frequencies. Hence, a small capacitor, 0.1 µF, will be required to filter high-frequency noise. Cypress recommends attaching a 0.1-µF ceramic capacitor *on every V_{dd} pin* of the CY2254. These capacitors must be attached as close to the pins as is physically possible. Surface mount capacitors are preferred because they have lower lead inductance.

Figure 2 shows the external capacitor connections.

Series Terminations

If the output of the CY2254 drives multiple loads or long traces, use a terminating resistor in series with the output, attached as close to the output pin as is possible. *Figure 2* shows a 22Ω resistor in series with the output. The value of this resistor, summed with the output impedance of the CY2254, should equal the characteristic impedance of the trace (transmission line). Typical values of the series resistor range from 10Ω to 75Ω.

A resistor in series with the output dampens the voltage reflections which occur with output impedance mismatches. It has the ultimate effect of reducing jitter on the output of the CY2254. Once again, surface mount resistors are preferred because of their lower lead inductance.

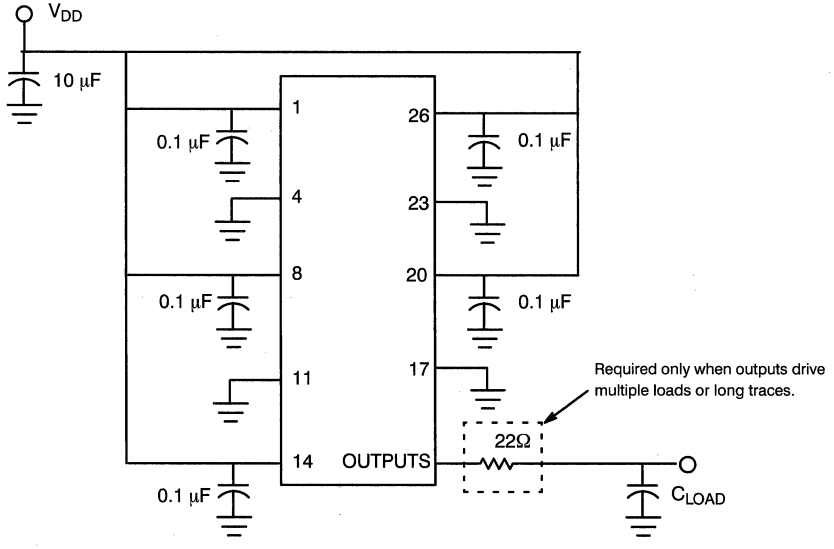


Figure 2. External Connections of the CY2254

Layout Guidelines

The following guidelines apply for laying out the CY2254 on a board:

- Provide a large ground plane under the CY2254. This will have the effect of reducing ground bounce in the system, thus reducing jitter.
- Connect each V_{SS} pin of the CY2254 to the ground plane individually. Connecting them together, and then to the plane will defeat the purpose of providing multiple ground pins.
- Avoid routing any high-frequency or clock signals below the CY2254, and place it in a relatively quiet area of the board. This will eliminate the coupling of any noise into the PLL, and will ensure lower jitter on the outputs.

All the above recommendations, along with a stable power supply source, will result in significantly reduced jitter on the clock outputs of the CY2254.

Conclusion

This application note introduced the reader to the CY2254 and presented some guidelines on using the device in systems. A summary of power supply filtering, termination, and layout guidelines was presented. With this information, the reader should be better able to design with the CY2254.

Triton and Pentium are trademarks of Intel Corporation.

Everything You Need to Know About CY7B991/CY7B992 (RoboClock) But Were Afraid to Ask

Introduction

The following application note provides a detailed description of the CY7B991 and CY7B992 Programmable Skew Clock Buffers (PSCB). It also provides an overview of clock distribution and transmission line analysis. This application note is divided into the following sections:

- General Description
- Clock Distribution
- System Design Considerations
- Detailed Device Description
- AC Specifications
- AC Characterization
- DC Specifications

General Description

Figure 1 is a general block diagram of Cypress's Programmable Skew Clock Buffer (internally called RoboClock). RoboClock employs a phase-locked-loop architecture to provide output clocks that are aligned both in phase and in frequency with a reference input clock. Each of the four output pairs is controlled by two dedicated three-level function select inputs that allow the outputs to be phase adjusted by as much as ±18 ns, divided, multiplied, or inverted. In all, over 26,000 different output combinations are possible.

A three-level frequency select (FS) input selects one of three PLL operating ranges that allow the outputs of the PSCB to operate from 3.75 to 80 MHz. All of these device configurations are possible while still maintaining an output-to-output skew and a propagation delay no greater than 500 ps.

The following section discusses the effects of skew on system performance, showing why RoboClock is ideally suited for solving clock distribution problems.

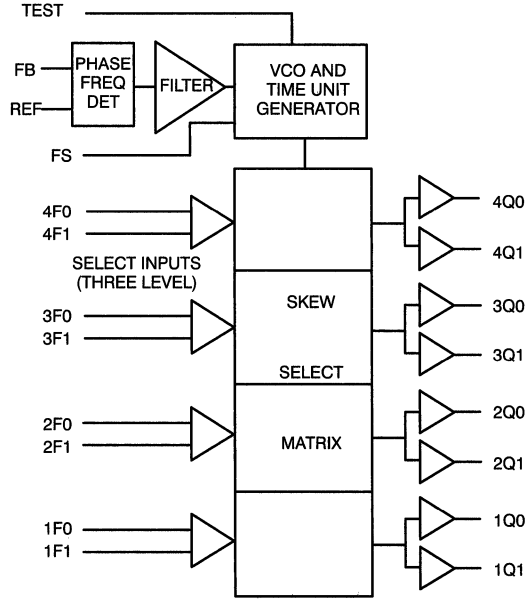


Figure 1. Logic Block Diagram

Clock Distribution

Skew is the variation in arrival time of two signals specified to occur at the same time. Skew is composed of the output skew of the driving device and variation in the board delays caused by the layout variation of the output traces.

Skew affects synchronous systems primarily in the form of clock skew. Since the clock signal drives many components of the system, and since all of these components should receive their clock signal at precisely the same time in order to be synchronized, any variation in the arrival of the clock signal at its destination will directly impact system performance. Skew directly affects system margins by eroding the predictability of the arrival of a clock edge. Because elements in a synchronized system require clock signals arrive at the same time, clock skew reduces the cycle time within which information can be passed from one device to the next.

As system speeds increase, clock skew becomes an increasingly large portion of the total cycle time. When cycle times were 50 ns, clock skew was rarely a design priority. It could be as much as 20% of the cycle time. As cycle times approach 15 ns and less, however, clock skew requires an ever-increasing amount of design resource. Typically, these high-speed systems can have only 10% of their timing budget dedicated to clock skew, so obviously, it must be reduced. The following sections will look at the two types of clock skew and how each affects system performance. The first type of skew is caused by the clock driver itself. It is referred to as intrinsic skew. The second type of clock skew is caused by the PCB layout and design and is referred to as extrinsic skew.

Clock Driver Skew (Intrinsic Skew)

Intrinisic clock skew is the amount of skew caused by the clock driver or buffer by itself. Intrinsic skew is not caused by board layout or any other design issues except for the specification stated on the clock driver data sheet. There are two main types of clock driver architectures: a buffer-type device and a feedback-type device.

Buffer Devices

In a buffer-style clock driver the input waveform propagates through the device and is "redriven" by the output buffers. This output signal directly follows the input signal. The output skew of these devices is caused by the differences in propagation delay between the input signal through the device and the precision of the matching and tuning of the internal circuit elements.

A member of this type of clock driver is the 74F244, which is available from several manufacturers. This device contains eight inputs that individually drive their respective outputs, and it is made into a clock distribution device by tying its inputs together to minimize the contribution of input skew to the device skew. The output skew of this device, if it is not listed on the datasheet, can be calculated by subtracting the minimum propagation delay from the maximum propagation delay. This calculated skew can be up to 3 ns.

This 3 ns clock driver skew does not even take into account the affects the board layout and design. In a 20- or 25-MHz system this is an acceptable amount of skew, but for systems running at 33 MHz and beyond, another method is needed.

To eliminate the device propagation delay variation that contributes to skew, manufacturers have designed devices that are specifically tuned to drive clock traces with low skew outputs. These manufacturers have specified the maximum variation in propagation delay through the device. In these devices, three types of skew parameters are usually listed. Output skew specifies the maximum amount of propagation delay variance between output pins. Duty cycle variation indicates a measure of the propagation delay difference between a LOW-to-HIGH output edge transition and a HIGH-to-LOW output edge transition. And part-to-part skew indicates the difference in output skew from device to device. Output skew of these devices has, in many cases, been reduced from the 3 ns mentioned above to 500 ps.

These devices still face the problems of device propagation delay. The propagation delay through these devices is about 5 ns. This delay will cause skew in systems where both the reference clock to the buffer and the outputs of the buffer need to be aligned.

These devices also have the drawback that the output waveform is directly based on the input waveform. If the input waveform is a non-50% duty-cycle clock, the output waveform will also have a less-than-ideal duty cycle. Expensive crystal oscillators are needed when using this type of buffer in systems requiring near 50/50 outputs.

These devices also lack the ability to phase or frequency adjust their outputs. Phase adjustment allows the clock driver to compensate for trace propagation delay mismatches and set-up and hold time differences, and frequency adjustment allows the distribution of high- and low-frequency clocks from the same common reference. Expensive additional components and time-consuming board routing techniques must be used to compensate for the functional shortcomings of these buffer-style clock driver devices.

PLL Clock Driver Devices

The second type of clock distribution device uses a feedback input that is a function of one of the outputs. This type of device is usually based upon one or more phase-locked loops (PLL) that are used to align the phase and frequency of the feedback input and the reference input. In this way the propagation delay through the device can be virtually eliminated. Cypress's Programmable Skew Clock Buffer family is based on this architecture, and will be explained in greater detail in the following sections.

In addition to very low device propagation delay, this type of architecture enables output signals to be phase shifted to compensate for board-level trace-length mismatches, and outputs can be selectively divided, multiplied, or inverted while still maintaining very low output skew.

Board Design Skew (Extrinsic Skew)

Just as the clock driver had to be evaluated to contribute minimal clock skew, the board layout and design must also be evaluated. Issues that affect board-level clock skew include trace length, capacitive loading, transmission line termination, and threshold voltages at the loads.

The time that it takes for a signal to propagate down the trace is dependent on factors such as the material that the PCB is constructed from, the length of the signal trace, the width of the trace, and capacitive loading. Variations in these factors from trace to trace will cause signals to arrive at their destination at different times.

In addition to this, threshold voltage variation on the receiving devices can play a significant role in the time of the recieved clock signals at various loads. If one load device has a threshold of 1.2V and another load device has a threshold of 1.7V and the rising edge rate is 1V/ns, there will be 500 ps of skew caused by the point at which the load device switches based on the input signal.

The most obvious way to reduce board design clock skew is to make the physical length of all clock traces the same. The propagation delay of an electrical signal down a trace is about 2 ns per foot. If one clock trace is just 3″ longer than the next, this will cause 500 ps of clock skew; which is as much skew as the clock driver itself contributes. But this is not enough. Impedance variation causes signal velocity variation, so physically matched lines may not be electrically identical.

Capacitive loading also contributes to clock skew. The differences in capacitive loading will cause differences in clock edge rate at the load. The variation between the edge rate of a lightly loaded trace and a heavily loaded trace will directly affect the time at which the clock edge crosses the input threshold and therefore the affect clock skew of these two devices.

Transmission line termination also plays a significant role in board-induced clock skew. Remember that a transmission line is any trace that has a propagation delay longer than one-half the driving device edge rate. With the extremely fast edge rates of today's clock drivers approaching 500 ps, traces with lengths of only 2″ must be considered transmission lines. Without proper termination, the clock signals

present on these traces will exhibit transmission-line effects such as voltage reflections that will, in the best case, cause a variability in clock-edge position and, at worst, might cause multiple clocking of the load.

Many of the issues mentioned in this section will be discussed in more detail in the System Design Considerations section.

System Design Considerations

Board Decoupling

Figure 2 shows the pinout for the PSCB family of devices. These parts are offered in both 32-pin PLCC and LCC packages. Each device contains 6 power pins (Vcc) and 5 ground pins. Each output pair (e.g., 1Q0 and 1Q1) have a dedicated power and ground pin immediately adjacent to them. For example, the 1Q0 and 1Q1 output pair have pins 25 and 22 as their dedicated power and ground supplies, respectively. These dedicated power pins are only used for the output driver pair. This provides the RoboClock outputs with very high drive while maintaining very high crosstalk immunity from adjacent outputs.

The other two pairs of power and ground pins are used to supply the internal PLL and associated circuitry. These power pins are completely separated from the power pins supplying the output buffers. This minimizes the output switching noise effects on the PLL and, therefore, minimizing output jitter.

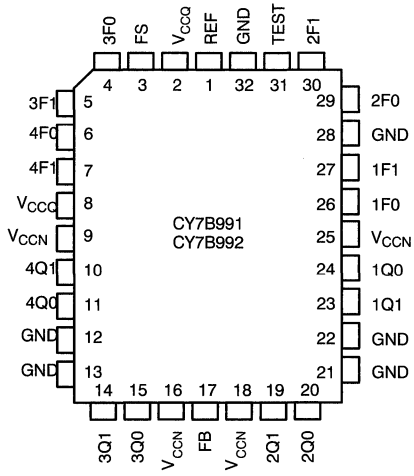


Figure 2. RoboClock PLCC/LCC Pinout

The CY7B99x family requires common high-speed power-supply decoupling and bypassing. Power-supply bypassing requires adding capacitance between the power and ground supply of a device in order to supply instantaneous current for its rapidly changing signals. This capacitor prevents the device from becoming current starved by providing the "instantaneous" transient current and thus preventing the local power-supply-voltage dip during current demands.

If all capacitors were ideal, this would be a trivial task. Capacitors, however, are not ideal. They are made up, in a first-order approximation, of an effective series inductor (ESL), effective series resistor (ESR), and a capacitor, as shown in *Figure 3*. Their response, therefore, is not constant over the frequency spectrum. At some frequencies this circuit (the bypass capacitor) will look most like a capacitor, at some frequencies an inductor, and at a particular frequency it will behave as nothing more than a resistor. This point is called series resonance and occurs when

$$\omega = \frac{1}{\sqrt{LC}} \qquad \text{Eq. 1}$$

In general, for a given capacitor construction, the series resonant frequency increases as the capacitance decreases. It would seem simple to choose a capacitor that has a series resonance at a higher frequency than the frequency of the current it needed to supply: To select a capacitor with a resonant point beyond 80 MHz, for example, if that were the operating frequency of the clock outputs. The selection process is not that simple.

Figure 4 shows the spectrum analysis of an 80 MHz clock signal with 1 ns edge rates. Notice that the fre-

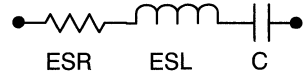


Figure 3. Equivalent Capacitor Diagram

quency components necessary to build this near ideal square wave have significant energy all the way out to 2 GHz. This means that in order to supply the current demands of these frequency components, a capacitor would have to have a series resonant frequency beyond 2 GHz. Generally available capacitors, however, have series resonant frequencies of about 400 MHz or less. These devices will be sufficient for supplying the majority of the instantaneous current.

At least two or three capacitors representing different capacitance ranges should be used for circuit bypassing. The first type of capacitor should be a 100- to 500-pF capacitor made of an NPO dielectric. This capacitor should be rated for operation at frequencies equal to or greater than 350 MHz. The second type of capacitor should be a 0.1-μF capacitor made from an X7R or a similar dielectric. This capacitor will supply the majority of the low frequency current requirements. If space permits, a third capacitor can be chosen that has a capacitance between that of the .1-μF and the 100- to 500-pF capacitors. This will provide a broad range of noise filtering and current supply.

The series resonant frequency can be decreased in two ways; lowering the capacitance or lowering the inductance. The major contributor to the inductance of a capacitor is the leads themselves. Surface-mount capacitors have a much smaller inductance than leaded capacitors and therefore have a higher resonant point. The benefits of a surface-mount capacitor can be completely nullified, however, if it is not properly integrated into the PCB environment.

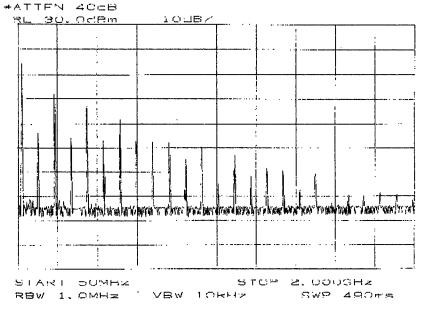


Figure 4. Frequency Components of an 80-MHz Clock Signal

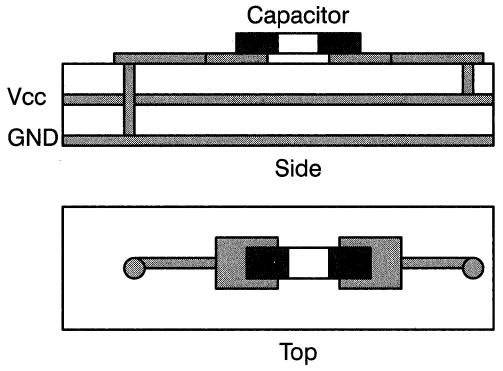


Figure 5. Typical Capacitor Layout

Figure 5 shows a typical method of integrating a surface-mount capacitor into a PCB environment. A surface-mount capacitor is used because of its low lead inductance, but no attention to reducing power connection trace inductance is made. In this case a leaded capacitor would provide less total inductance. The surface-mount capacitor, in this case, may not provide any high-frequency bypassing.

A better method of laying out surface-mount capacitors is shown in *Figure 6.* Here the multiple short leads are made to the power planes. The trace

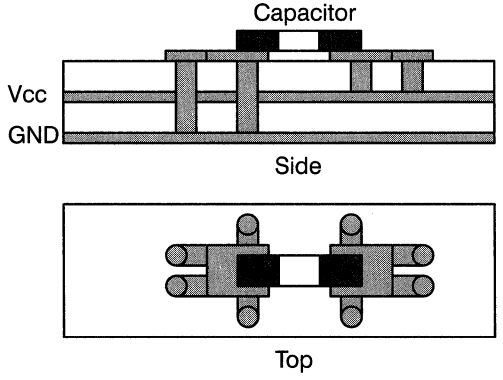


Figure 6. Better Capacitor Layout

widths and via hold sizes are also increased. These methods reduce the inductance of the power connections of the capacitor. Only when proper attention is paid to the selection and layout of the capacitor will the true benefits of circuit bypassing be realized.

Figure 7 shows a sample layout of RoboClock on a multilayer printed circuit board. This figure assumes that an internal V_{CC} and GND plane exist. The internal board V_{CC} and GND planes are connected to the device V_{CC} and GND planes through multiple via holes shown as black dots in the figure. Multiple via holes and connection of the chip power pins to local power planes reduces the amount of inductance that these pins have to their respective power connections.

Two sets of capacitors are used. They are placed on the same side of the board as the device. Each set consists of a 0.1-µF and a 100-pF high-frequency capacitors. Mutliple via holes are also included for the bypass capacitor connections to reduce the inductance that the V_{CC} pins see in series with their capacitor. The FB pin is connected to the 2Q1 pin in this diagram as an example of how easy the FB pin can be connected to either the 3Q0 and 2Q1 pins in order to reduce trace length and minimize potential problems associated with voltage reflections on transmission lines.

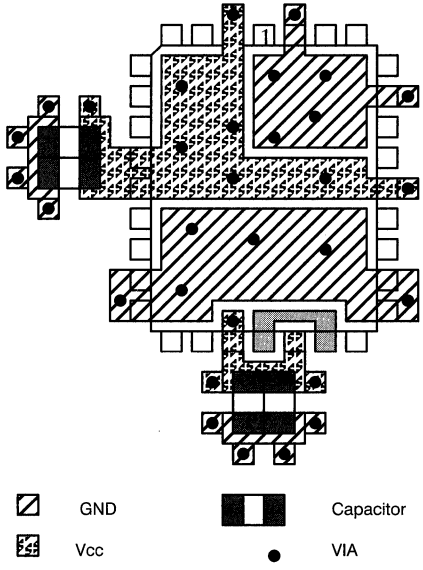


Figure 7. Sample RoboClock Layout

This layout is not the only way that these devices can be laid out, but this figure shows examples of good high-performance layout techniques. It is assumed that the board in which RoboClock will be placed will contain at least one dedicated V_{CC} plane and one dedicated ground plane and that the device will be surface mounted directly to the PCB without the use of a socket. The reason for the last constraint is that the additional lead inductance of the socket directly impacts the output skew of the device.

A more detailed discussion of series resonant frequency and other capacitor characteristics can be found in the materials supplied by capacitor manufacturers such as American Technical Ceramics [(516) 547-5700] and AVX [(803) 448-9411].

Transmission Lines

Transmission line theory states that a signal sent down a transmission line that has a constant characteristic impedance will propagate undistorted along the line. At the end, a voltage reflection will occur if the load impedance is not equal to the characteristic impedance of the transmission line. These voltage reflections are always present in electrical interconnections between devices and have traditionally been ignored. With the lengthening of Printed Circuit Board (PCB) traces and the decreasing of the edge rates of the driving element of these electrical signals, however, these effects become more pronounced. Transmission line effects cause many undesirable results in high-speed systems, such as delays and ringing. These effects will be discussed in greater detail.

In general, the effects of voltage reflections should be considered when laying out clock lines and any other PCB trace, if the propagation delay of the trace is greater than twice the faster of the rise time or fall time of the driving signal (*Equation 2*).

$$MIN[t_r, t_f] < 2\,t_{pd} \qquad \text{Eq. 2}$$

In other words, if the rise time (or fall time) of the source is less than the two-way propagation delay, then the rising signal will not hide the effects of the signal propagating down a transmission line. In this case, the switching wave will have enough time to propagate down the transmission line, reflect off of the load, and be seen at the source. These voltage reflections can cause decreased signal integrity, that will manifest itself, in the case of clock traces, as increased rise and fall times, non-ideal duty cycle performance, and possibly even unwanted clock pulses due to voltage reflections that cross the threshold of the load device.

The first step, then, in determining if a trace should be considered a transmission line, is to evaluate the propagation delay and characteristic impedance. The propagation delay is needed in order to determine when a trace must be considered a transmission line and the characteristic impedance is needed in order to determine how to reduce voltage reflections on these traces as will be shown in the section entitled Transmission Line Termination. The following discussion will focus on calculating the propagation delay and characteristic impedance on various PCB traces. This analysis, however, can be easily extended to include other types of transmission media such as coax, twisted pair, and wire-wrapped environments.

The analysis of transmission line effects on PCB traces begins with a simplified circuit analysis of the trace itself (*Figure 8).* This figure models the trace as a distributed intrinsic resistance (R_O), inductance (L_O), and capacitance (C_O). For the purposes of this discussion, a lossless transmission line will be assumed, that implies that the intrinsic series resistance will be equal to 0. The effect of this resistance on the characteristic impedance is extremely small, and only on very long traces will the effects of this component result in a noticeable drop in the voltage realized at the load.

The characteristic impedance, therefore, can be expressed as:

$$Z_0 = \sqrt{\frac{L_0}{C_0}}\ \Omega \qquad \text{Eq. 3}$$

And the propagation delay can be expressed as:

$$t_{pd} = \sqrt{(L_0 \times C_0)} = Z_0 C_0\ ns/length \qquad \text{Eq. 4}$$

In many cases it may be hard to measure the intrinsic inductance and capacitance of the trace in order to determine the magnitude or even the existence of transmission line effects. In this case, equations are needed in order to determine these values.

The following two sections will give equations for the characteristic impedance and propagation delay of two typical types of PCB trace construction; microstrip and strip line. Many sources exist that give an analysis of the equations listed below. Some sources give an even a more detailed analysis, but the minor differences are overshadowed by errors caused by factors not related to the analysis such as

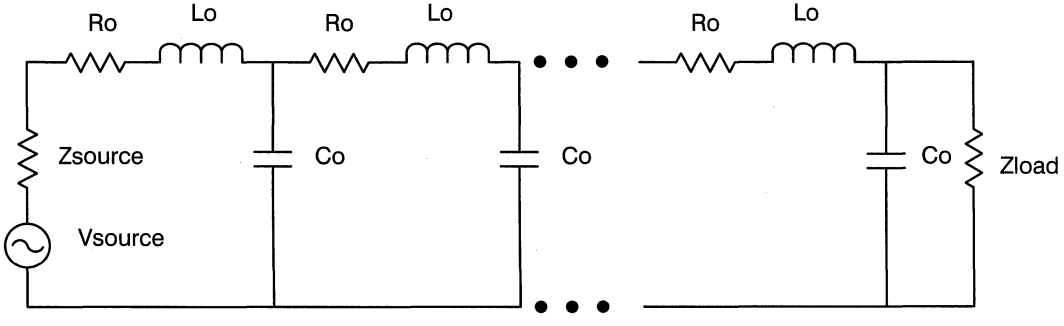


Figure 8. Simplified PCB Trace Model

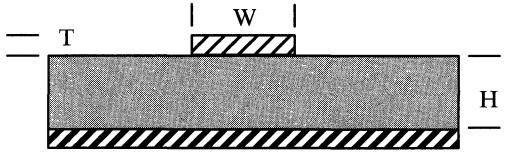


Figure 9. Microstrip

component variation and manufacturing uncertainties.

Microstrip

A microstrip trace is a signal separated from the ground plane by a dielectric, as shown in *Figure 9*. This type of trace is most commonly found as the top or bottom traces on a multilayer printed circuit board. The formula for calculating the characteristic impedance is given as:

$$Z_0 = \frac{87}{\sqrt{\epsilon_r + 1.41}} \ln\left(\frac{5.98H}{0.8W + T}\right) \quad \text{Eq. 5}$$

and the propagation delay can be expressed as

$$t_{pd} = 1.017\sqrt{0.475\epsilon_r + 0.67} \quad \text{Eq. 6}$$

where

ε_r is the dielectric constant of the material used for the PCB construction
H is the distance the trace lies away from the ground plane (board thickness)
W is the trace width (wire width)
T is the trace thickness (wire thickness)

This formula will not yield the exact impedance of the trace, but is meant as a guideline for estimating the trace impedance. Differences between the calculated and real trace impedance will be caused by slight errors in the equation itself and in process and layout variability in parameters such as dielectric constant, ground plane continuity, capacitive loading, trace width and thickness, and board thickness.

All of these variables, except for possibly the dielectric constant and trace thickness, are under the control of the designer during board layout. Dielectric constants of material used in the construction of fiberglass PCBs have a value between 4.0 and 5.5. Board manufacturers should be able to provide this parameter upon request. *Figure 10* is a graph showing how trace impedance varies with trace width and dielectric thickness. The graph assumes a dielectric constant of 4.5 and a trace thickness of 1.4 mils.

For example, if the designer wishes to create a microstrip trace with a 50Ω impedance, the designer would first have to know the thickness of the dielectric. If the board is a four-layer board (two signal layers and two routing layers) and if the trace will be on the component side with the power plane directly beneath it, then the dielectric thickness is roughly

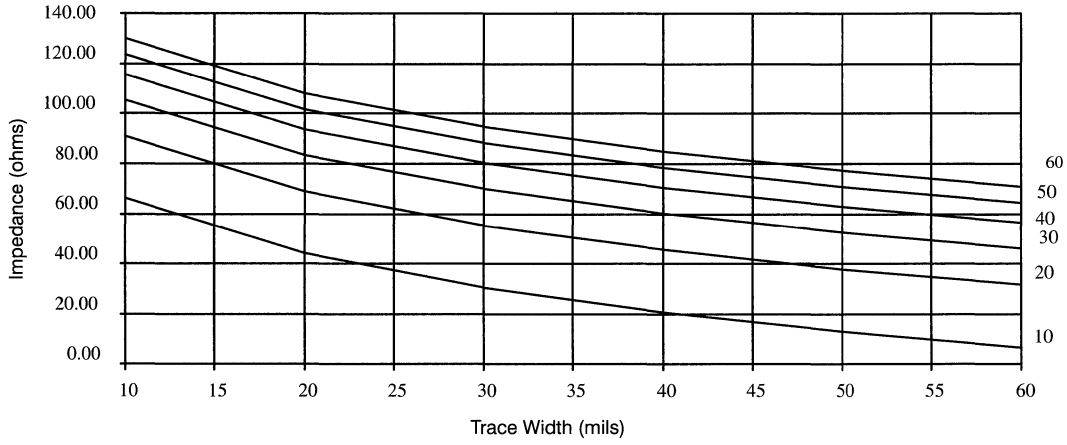


Figure 10. Impedance vs. Trace Width over Dielectric Thickness (Microstrip)

the board thickness divided by three. For a 62.5 mil board this would translate to a dielectric thickness of about 20 mils.

The designer would also have to know the thickness of the trace (approximately 1.4mils for standard 1 oz copper traces), and the dielectric constant (assume 4.4). All that is left now is the thickness of the trace, which can be directly controlled as part of the layout process. In this example, if the trace width is 36 mils, the characteristic impedance of the trace would be

$$Z_0 = \frac{87}{\sqrt{4.4 + .41}} \ln\left(\frac{5.98x20}{0.8 * 36 + 1.4}\right) = 49.68\Omega \quad \text{Eq. 7}$$

The propagation of a signal along this trace is independent of everything but the dielectric constant and is calculated in this case as:

$$t_{pd} = 1.017\sqrt{0.475 * 4.4 + 0.67} = 1.69\ ns/foot \quad \text{Eq. 8}$$

Both the equation for characteristic impedance and propagation delay will be useful for estimating the magnitude of voltage reflections and for determining the correct method of eliminating these reflections, which will be discussed in the Transmission Line Termination section.

Strip Line

Strip Line is analogous to a buried trace in a multilayered PCB between two power planes as shown in *Figure 11* .

The characteristic impedance for this type of trace can be expressed as

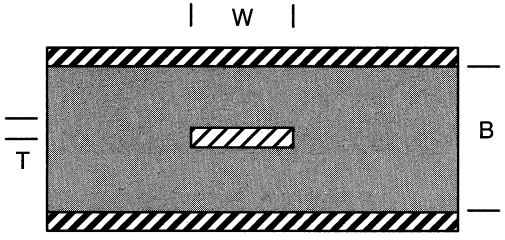


Figure 11. Strip Line

$$Z_0 = \frac{60}{\sqrt{\epsilon_r}} \ln\left[\frac{4B}{0.67\pi W(0.8 + \frac{T}{W})}\right] \quad \text{Eq. 9}$$

For cases when

$$\frac{W}{B - T} < 0.35 \quad \text{and} \quad \frac{T}{B} < 0.25$$

With a propagation delay of

$$t_{pd} = 1.017\sqrt{\epsilon_r} \quad \text{Eq. 10}$$

Figure 12 shows how the impedance of a strip line trace varies with the trace width and dielectric thickness. The graph shows that to create a strip line trace with a 50Ω impedance in a multilayer board with 10 mils of epoxy between layers requires approximately a 7-mil trace. This assumes a trace thickness of 1.4 mils and a dielectric constant of 4.5.

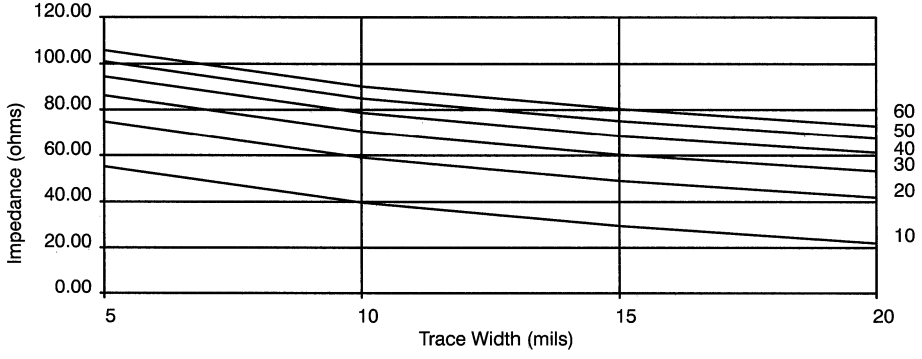


Figure 12. Impedance vs. Trace Width over Dielectric Thickness (Strip Line)

Transmission Line Effects

The previous section discussed how to calculate the characteristic impedance of a trace and the propagation rates of that signal along a trace. This section will briefly discuss the effects of transmission lines on signal integrity.

The cause of transmission line effects is impedance mismatches. These mismatches occur because of the impedance differences between the transmission line and the source and load. They are also caused by impedance discontinuities and unequal loading of the transmission line along its length. Transmission line stubs and vias are examples of impedance discontinuities otherwise known as impedance bumps.

Any time the impedance along a transmission line changes, a voltage reflection will occur. *Figure 13* shows a simplified diagram of the transmission line environment that will be used to illustrate the concept of voltage reflections.

At time 0, the voltage source provides a current source to the load equal to

$$I_S = \frac{V_S}{Z_S + Z_0'} \qquad \text{Eq. 11}$$

Since no voltage is dropped across the transmission line (assume a lossless transmission line as before), the total supplied current must be utilized by the load. If the load resistance is not equal to the characteristic impedance of the trace, a voltage reflection will occur. The reflection coefficient at the load is expressed as

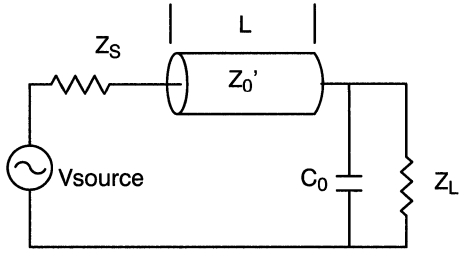


Figure 13. Simplified Transmission Line

$$\rho_L = \frac{\textit{reflected voltage}}{\textit{incident voltage}} = \frac{V_{S1}}{V_S} = \frac{Z_L - Z_0'}{Z_L + Z_0'} \qquad \text{Eq. 12}$$

and the reflection coefficient at the source is specified as

$$\rho_S = \frac{\textit{reflected voltage}}{\textit{incident voltage}} = \frac{V_{S2}}{V_{S1}} = \frac{Z_S - Z_0'}{Z_S + Z_0'} \qquad \text{Eq. 13}$$

If the load has a lower impedance than the characteristic impedance of the transmission line, then a negative voltage reflection will be sent back to the load, indicating that the load over used the available current. If the load has a higher impedance than the characteristic impedance of the transmission line, then a positive voltage will be sent back to the source, indicating that the load under used the amount of available current.

The following example shows the magnitude of these voltage reflections in a typical unterminated transmission line being driven by a CY7B991. For this example, assume the parameters have values listed in in *Table 1.*

Table 1. Sample Parameters

Param	Description	Value
ε_r	Dielectric Constant	4.4
H	Distance from ground	20 mils
T	Trace Thickness	1.4 mils
W	Trace Width	12 mils
Z_{PUS}	Source Pull-Up	25Ω
Z_{PDS}	Source Pull-Down	10Ω
L	Trace Length	6"
R_L	Load Impedance	12 KΩ
t_R	Output Rise Time	1.5 ns
t_F	Output Fall Time	1.5 ns

Solving the appropriate equations for Z_0, t_{PD}, and Z_0 yields

$$Z_0 = \frac{87}{\sqrt{4.4 + 1.41}} \ln\left(\frac{5.98 * 20}{0.8 * 12 + 1.4}\right) = 86.26\Omega \qquad \text{Eq. 14}$$

$$t_{pd} = 1.017\sqrt{0.475 * 4.4 + 0.67} = 1.66\ ns/foot \qquad \text{Eq. 15}$$

From this, the load and source reflection coefficients can be calculated:

$$\rho_{SLH} = \frac{25 - 86.26}{25 + 86.26} = -0.55 \quad \text{Eq. 16}$$

$$\rho_{SHL} = \frac{10 - 86.26}{10 + 86.26} = -0.79 \quad \text{Eq. 17}$$

$$\rho_{L} = \frac{120,000 - 86.26}{120,000 + 86.26} = 1.00 \quad \text{Eq. 18}$$

Two source reflection coefficients need to be calculated for the LOW-to-HIGH output transition and the HIGH-to-LOW output transition. The reason for this is that TTL-style output drivers have different resistances depending if they are driving HIGH or LOW. In the calculation of the source coefficients, the actual output impedances for the output drivers and the input impedance of the REF input of the PSCB were used. They are listed in *Table 1.*

This analysis assumes a transmission line exists. To confirm this, the propagation delay is used in *Equation 2:*

$$MIN[t_R, t_F] < 2 * t_{pd}'$$
$$1.5 < 2 * 1.69 \quad \text{Eq. 19}$$

A complete DC analysis of voltage reflections on this type of transmission line is not conducted here. Refer to Reference 4. at the end of this note for a sample analysis. It is important to note from this that the voltage reflection at the load is positive and the reflection at the source is negative. The opposite signs of these two reflection-coefficients and the magnitude of these two constants indicate that ringing along the transmission line could potentially cause unwanted clocks pulses to be seen at the load.

Figure 14 shows the effects of an unterminated transmission line. This plot is taken from a Tektronix DSA 602A Digitizing Signal Analyzer. This plot indicates that the the voltage scale is 1V/div and the time scale is 5 ns/div. The ground signal is indicated by the ground symbol found on the left-hand side of the figure, three voltage divisions from the bottom. The scope trigger, indicated by an arrow, is the 1.5V level of the source waveform.

This analyzer is capable of taking measurements on the waveforms. The bottom quarter of the plot displays the various measurements taken on the source waveform. The rise and fall measurements are taken between the 0.8V and 2.0V level, which is consistent with the measurement points of the CY7B991. The pulse width measurement is taken at the 1.5V level and is used to indicate the duration of time that the waveform spends above 1.5V. This is used to calculate the output duty cycle variation. The last measurement shown on this plot is the frequency. In this case, a 25-MHz source waveform is used as the trigger.

The source waveform shown in *Figure 14* is the 4Q0 output of a typical CY7B991-5. The load waveform is the end of 18″ of coaxial cable having a characteristic impedance of 50Ω. In this particular example the load is completely unloaded except for the scope probe and associated connector (approximately 5 pF). As can be seen, both the source and the load are riddled with transmission line effects. This is caused by the positive voltage reflection from the load due to the infinite load impedance and the negative voltage reflection from the source due to the source impedance being less than the characteristic impedance of the coaxial cable. Notice that an additional clock pulse occurs (below the word "load" in the figure) due to voltage reflections.

A complete analysis of the voltage reflections causing this waveform would be extremely boring and

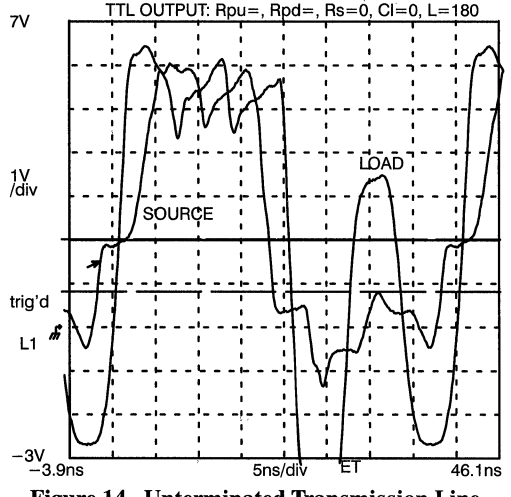


Figure 14. Unterminated Transmission Line

yield little additional information aside from the fact that this is an unacceptable clock waveform. The next section will give some methods for eliminating voltage reflections by terminating the transmission line.

Transmission Line Termination

The goal of transmission line termination is to make the source and/or load impedance match the characteristic impedance of the transmission line, insuring an optimal delivery of signal to the load. Two types of termination will be discussed: parallel termination and series termination.

Parallel Termination

Parallel termination, also known as Thevenin termination, attempts to match the load impedance with that of the transmission line. It is accomplished by placing a pull-up and pull-down resistor pair at the end of the transmission line nearest the destination as shown in *Figure 15*. The Thevenin equivalent resistance must be equal to Z_0 and the Thevenin voltage must be somewhere near the middle of the normal TTL voltage swing. Since typical CMOS inputs have steady-state impedances in the 100kΩ to 1MΩ the Thevenin resistance can be simplified to R_{PU} and R_{PD} in parallel. *Table 2* gives the recommended parallel termination resistor values for both the CY7B991 (TTL) and the CY7B992 (CMOS) devices for given transmission line impedances (Z_0).

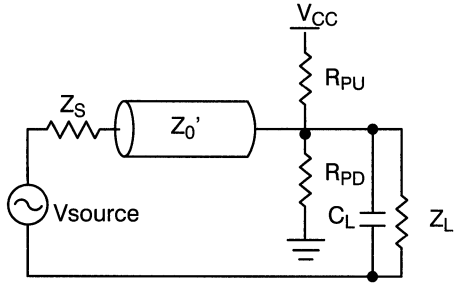


Figure 15. Parallel Termination

Table 2. PSCB Parallel Termination Recommendations

Z_0	TTL (R_{PU}/R_{PD})	CMOS (R_{PU}/R_{PD})
50Ω	130/91	100/100
65Ω	158/111	130/130
75Ω	182/128	150/150
100Ω	243/170	200/200

Figure 16 shows a 100Ω parallel termination of a transmission line with a 50Ω impedance. This figure is a plot taken from the same analyzer mentioned above. As before, the source waveform is taken at the RoboClock output pin and the load waveform is taken at the end of 18″ of coaxial cable with a 50Ω impedance. This figure shows the positive voltage reflection from the load because the load impedance is greater than the trace impedance. This reflection manifests itself as the source waveform stepping up to the 3.7V level as shown in *Figure 16*. The magnitude of the intitial source waveform in transmission line environment can be calculated by determining the the current in the transmission line before switching. The reflected voltage can be calculated by multiplying the incident voltage by the calculated reflection coefficient.

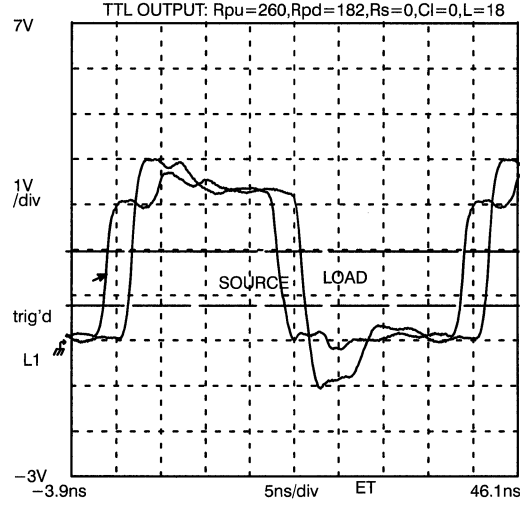


Figure 16. Unloaded 100Ω Parallel Termination

This waveform looks much better than in the previous unterminated case, but there is still a 1V undershoot at the falling edge of the waveform that may cause problems in some devices due to substrate current generation.

Figure 17 shows a 15Ω parallel termination. This transmission line is under terminated due to the fact that the load impedance is much less than the characteristic impedance of the trace. This manifests itself as a negative voltage reflection from the load. While the rising edge of this trace looks fairly good, the falling edge will cause problems. Notice that the negative voltage reflection ringing during the falling edge causes a positive-going transition in the threshold region of the load. This could cause spurious clocking.

Figure 18 shows a unloaded terminated transmission line. This transmission line, unlike in the other three cases, is terminated in its characteristic impedance. Both the source and the load waveforms are unaffected by voltage reflections. The only evidence of transmission line effects is the ringing during the HIGH and LOW time of the waveform, caused by slight impedance mismatches between the source, load, and transmission line, and the impedance "bumps" caused by the impedance differences at the various probe and coax connectors along this transmission line.

This ringing is also caused by the imperfect nature of terminating a transmission line with a complex load and source impedance. Each of the capacitances hanging on this transmission line have different impedances based on the input frequency. And, since the input waveform is a wideband signal containing many frequency components, a simple termination as given here will not eliminate all voltage reflections. The important thing to note is that this termination gives good results on the given transmission line.

Figure 19 shows a 50Ω parallel termination with a 22-pf load. Notice the voltage spikes on the source waveform. This is caused by the impedance of the load capacitor at a particular frequency being incorrectly terminated and causing a voltage reflection.

Figure 20 shows a 50Ω parallel termination with a 50-pF load. Again, the source waveform exhibits voltage glitches caused by brief negative voltage reflections, but in this case the voltage reflections come dangerously close to the input threshold region of a device that was placed on the transmission line somewhere between the source and the load shown in the figure. For this reason, it is recommended that all transmission lines drive either single loads or loads that are lumped at the end of the transmission line. If not, voltage spikes, even in

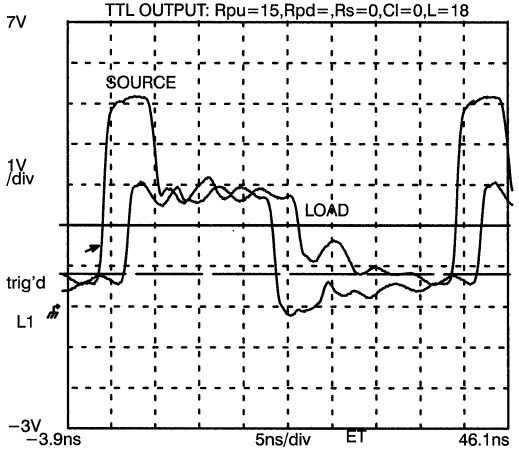


Figure 17. Unloaded 15Ω Parallel Termination

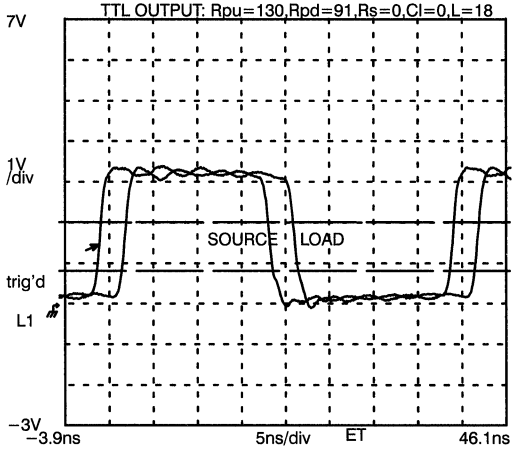


Figure 18. 50Ω Parallel Termination, Unloaded

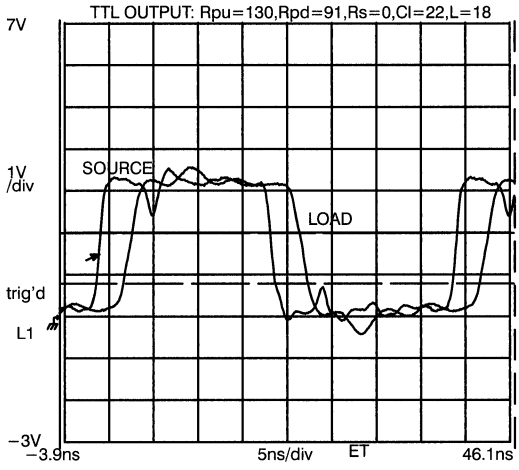


Figure 19. 50Ω Parallel Termination, 22-pF Load

reasonably well terminated transmission lines, can cause unwanted clocking.

Series Termination

The purpose of series termination is to match the source impedance with the transmission line impedance as shown in *Figure 21.* This will prevent voltage reflections occurring from the load will not reflect back from the source. The value of R_S is chosen such that the series combination of the output impedance

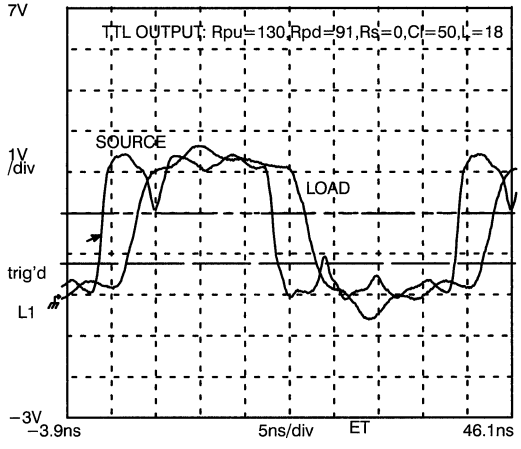


Figure 20. 50Ω Parallel Termination, 50-pF Load

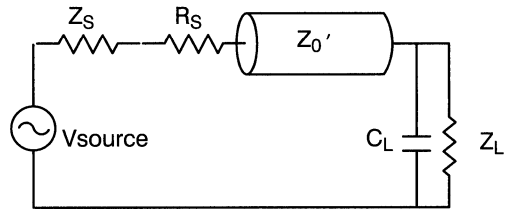


Figure 21. Series Termination

of the source devices and R_S is equal to Z_0. This series termination will absorb any voltage reflections from the load. This, however, is not a simple task.

TTL outputs have different LOW-to-HIGH and HIGH-to-LOW output impedances. The output drive has an asymmetrical output impedance. *Figure 22* shows the HIGH-to-LOW linearize Voltage vs. Current (V-I) curve for a typical CY7B991 output. The output impedance that should be used is the resistance when the output is LOW (less than 0.45V). *Figure 23* shows the LOW-to-HIGH linearized curve. The output impedance that should be used in this case is the resistance when the output is HIGH (greater than 2.4V). For these curves the output high resistance (27Ω) and the output LOW resistance (7Ω) can be determined.

Figure 24 shows an unloaded, 100Ω series terminated transmission line. This figure shows the classic "stair stepping" that occurs when a transmission line is terminated with series resistance that is too large. Several back-and-forth voltage reflections

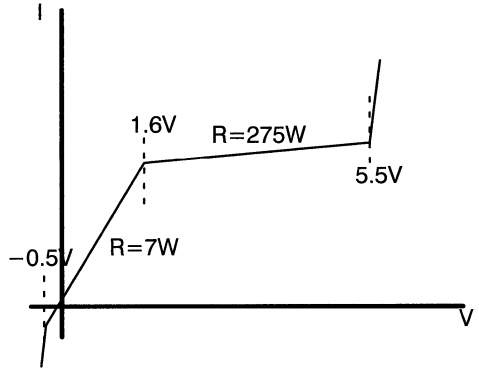


Figure 22. Output LOW Linearized V-I Curve

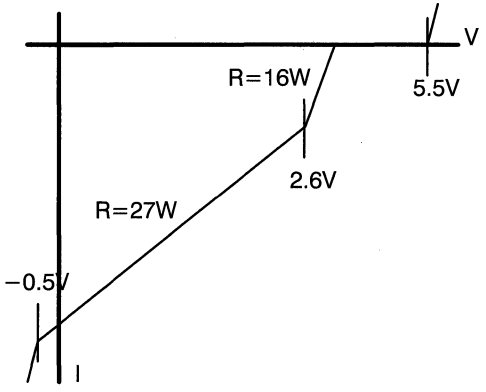


Figure 23. Output HIGH Linearized V-I Curve

are required before the load rests at its final voltage. This improper termination can cause duty-cycle distortion, increased rise and fall times, and unexpected clocking.

Figure 25 and *Figure 26* show unloaded and loaded 50Ω series terminated transmission lines, respectively. The resultant waveforms look much better in this case because the terminating resistor more closely matches the characteristic impedance.

Figure 27 shows a 27Ω terminated transmission line with a 22 pF load. The rising edge of this waveform looks much better than either of the previous two

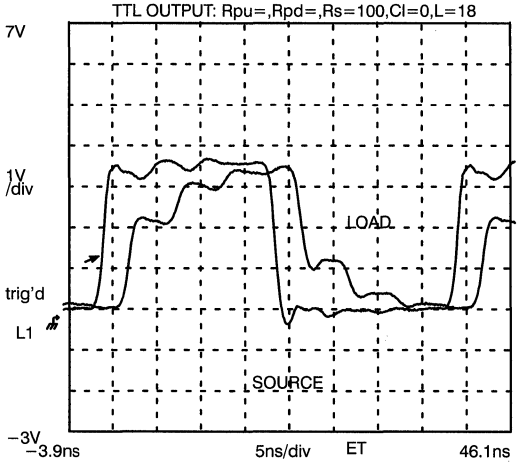


Figure 24. 100Ω Series Terminated, Unloaded

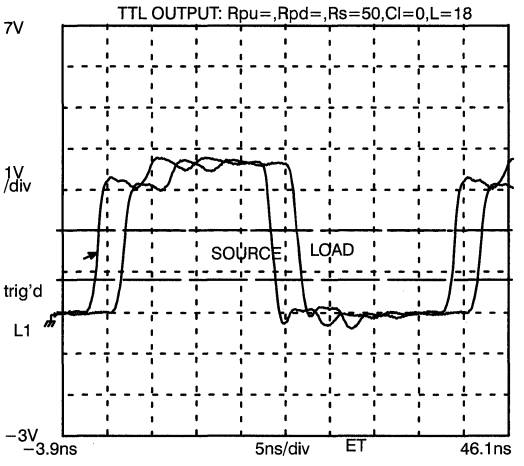


Figure 25. 50Ω Series Terminated, Unloaded

cases, but the falling edge has more undershoot than in the 50Ω termination example. The reason for this is that on the LOW-to-HIGH output transition the 27Ω terminating resistor plus the 27Ω output impedance closely match Z_0, but on the HIGH-to-LOW output transition the 27Ω resistor plus the 7Ω output impedance do not exactly match Z_0. With series termination a trade off has to be made between the LOW-to-HIGH transition and the HIGH-to-LOW transition.

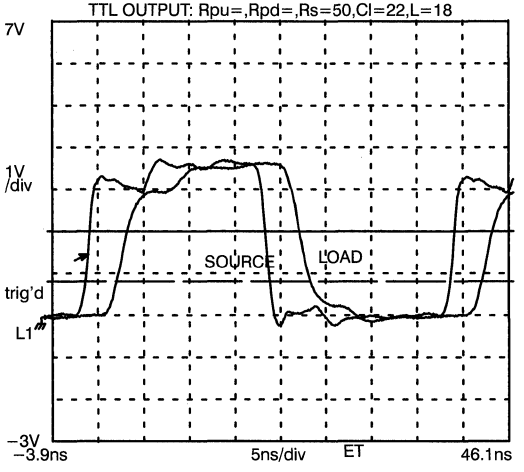


Figure 26. 50Ω Series Terminated, 22-pF Load

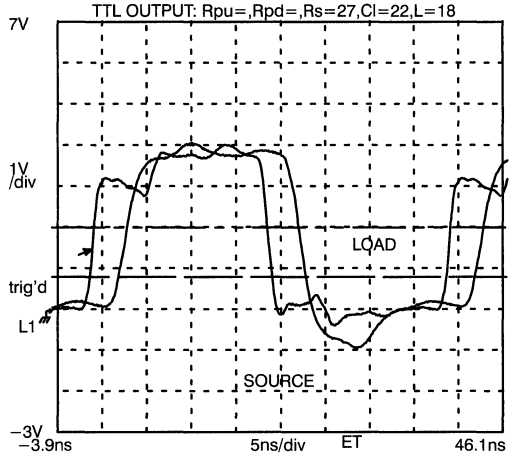


Figure 27. Series Terminated, 22-pF Load

When using series termination, no loads may exist along the transmission line. All loads have to be located at the end of the trace. The reason for this is that the series-terminating resistor acts like a voltage divider. Any loads not located at the end of the trace will see a voltage at some indeterminate level until the voltage reflection from the load builds the voltage level to its final resting value.

An advantage of series termination over parallel termination is that there is no DC power consumption. In a parallel termination, whether the output device is driving HIGH or LOW, there will always be current flow that does not drive the load but simply establishes the terminations.

Another note on transmission lines is that the rise time of the output waveform does not depend on any transmission-line loading considerations. By looking at the rise times of the source waveform in the previous example, it is clear that the method of transmission line termination and the output loading play a negligible role in the output rise time. In a properly terminated transmission line, the output rise and fall time will be a function of the characteristic impedance of the trace and the capacitive loading of the load.

The recommendation for terminating transmission lines is either a parallel termination with an R_{TH} = Z_0 and a V_{TH} = 2.06V or a series termination where $R_S = Z_0 - 20\Omega$. If more than one load has to be driven by a single device output, make sure that all loads are located very near the end of the line, or create a "star" layout by having each load have its own trace starting at the RoboClock output. Terminate each trace as if it were the only load being driven. In the case of multiple loads being driven by a single output, the series resistor should be calculated with

$$R_S = Z_0 - \frac{20}{\#\ of\ Traces} \qquad \text{Eq. 20}$$

All lines must be matched or the loads will "talk" to each other through the voltage reflections.

Trace impedances below 50Ω can be driven by tying more than one output together. For example, a 25Ω trace can be driven by tying two outputs of the same output pair together.

Never "daisy-chain" loads together. This will not only immediately add load-to-load skew, but it will also cause unpredictable transmission line effects.

Detailed Description

RoboClock is an eight-output clock driver device. It differs from traditional clock drivers and buffers in that its outputs, while having very low output skew, can also be phase adjusted, inverted, divided, and multiplied. These capabilities would be impossible to implement in a device such as a simple redrive buffer like a 74F244. A 74F244, while potentially providing very low output skew, does not have the capability to dynamically phase adjust its outputs.

Phase adjustment allows outputs to shift in time relative to a reference point. The input clock to the device is usually taken as this reference point. Phase adjustment is useful for compensating for differences in trace delay from one load to the next, and also for equalizing differences in set-up and hold time between load devices. A 74F244 would have great difficulties shifting the arrival time of its outputs both in the positive sense (output edge arrives later that the reference edge) and in the negative sense (output edge arrives before the reference edge). In order for a 74F244 to accomplish this task, it would have to predict the time at which the next

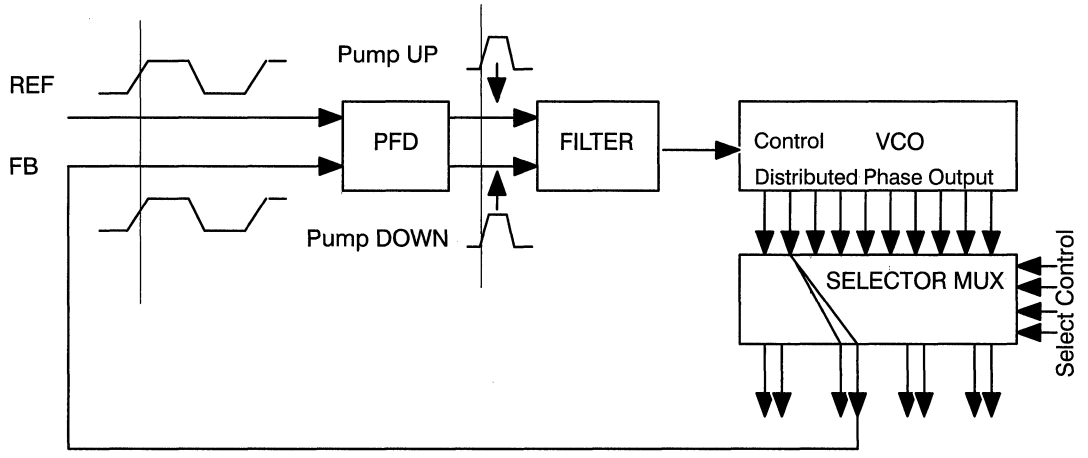


Figure 28. Simplified PLL Architecture

input edge would occur. Obviously, a new approach is needed.

PLL Operation

RoboClock includes a phase-locked loop (PLL) to achieve zero propagation delay. A completely integrated PLL allows you to align both the phase and the frequency of the reference (REF) inputs with an output. With this approach, the next occurrence of an input edge can be predicted with great accuracy while maintaining very low propagation delay through the device.

The PLL has three distinct parts; the phase/frequency detector, the filter, and the distributed-phase clock oscillator (more simply known as a voltage-controlled oscillator). In order for the PLL to align the REF input with any output, an output must be selected to be fed back to the input of the PLL. This input (FB) is then used as the alignment on which all other outputs are based. (See *Figure 1*).

Phase Detector and Filter

Figure 28 shows a simplified view of the RoboClock PLL architecture. The Phase Frequency Detector (labeled PFD) evaluates the rising edge of the REF input with respect to the FB input. If the REF input occurs before the FB input, indicating that the Voltage Controlled Oscillator (VCO) is running too slow, the PFD produces a Pump Up signal that lasts until the rising edge of the FB input. If the FB input occurs before the REF input, on the other hand, the PFD produces a Pump Down signal that is triggered on the rising edge of the FB input and lasts until the rising edge of REF. This Pump Down pulse forces the VCO to run slower. In this way, the PFD forces the VCO to run faster or slower based on the relationship of the REF and FB inputs. In the absence of a REF input, the device will function at approximately its slowest operating speed.

The Filter converts these Pump Up and Pump Down signals into a single control voltage. The magnitude of this voltage is dependent on the number of previous Pump Up and Pump Down pulses that have occurred. The range of the voltage produced by the filter is guaranteed to be able to force the VCO into any frequency within the selected frequency range.

Distributed-Phase Clock Oscillator

Figure 29 shows the Distributed-Phase Clock Oscillator ring and the Output Adjust Matrix. The RoboClock Distributed Phase Clock Oscillator (also known as a ring oscillator) has three frequency ranges of operation. These frequency ranges are se-

lected with the FS pin with range values shown in *Table 3.* At first glance, it may seem odd that a single pin (FS) has three possible selections. These three-state inputs are another feature of RoboClock. All function select inputs (FS, TEST, and xFn) have the ability to be connected to one of three states; HIGH, LOW, and MID. HIGH indicates a connection to VCC, LOW indicates a connection to Ground, and MID indicates an open connection. When a three-level input is left unconnected, internal resistors pull this input to approximately $V_{CC}/2$.

Table 3. Frequency Range Select and t_U Calculation

FS[2]	f_{NOM} (MHz) Min.	f_{NOM} (MHz) Max.	$t_U = \frac{1}{f_{NOM} \times N}$ where N =	Approximate Frequency (MHz) At Which t_U = 1.0 ns
LOW	15	30	44	22.7
MID	25	50	26	38.5
HIGH	40	80	16	62.5

The three different frequency ranges correspond to the number of stages in the oscillator. When FS is connected to ground, the oscillator contains its maximum number of stages: 22. When FS is left unconnected, the oscillator contains 13 stages. And when FS is connected to ground, the oscillator contains its minimum number of stages: 8. The operating frequency of the oscillator can be calculated with the following formula:

$$f = \frac{1}{N * t_U} \qquad \text{Eq. 21}$$

where N is the number of stages and t_U is the delay through each stage. The reason that N at the bottom of *Equation 21* is twice the number of stages in the oscillator is because, in order for the ring to oscillate, first the true and then the inverted signal must pass through each stage of the oscillator. This is accomplished through an inversion from the last stage to the first stage.

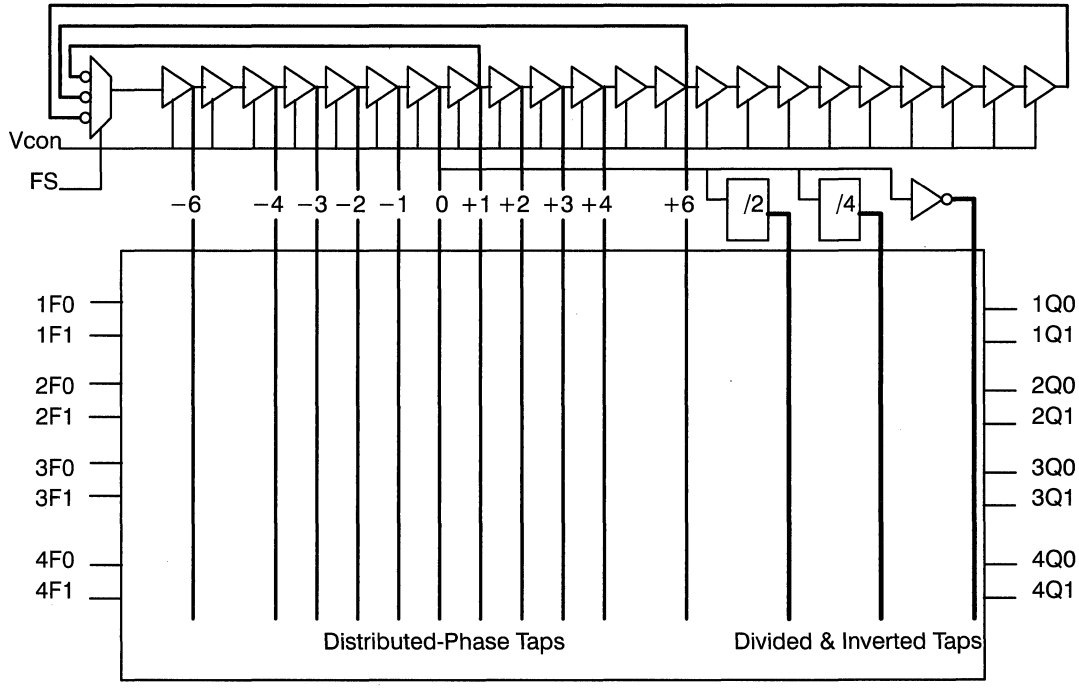


Figure 29. Distributed-Phase Clock Oscillator and Output Adjust Matrix

For example, if the delay through each stage is exactly 1 ns and the FS pin was tied to ground, then the operating frequency would be

$$f = \frac{1}{2 * 22 * 1ns} = 22.7\ MHz \qquad \text{Eq. 22}$$

The delay through each stage is controlled by the voltage on the V_{CON} line, which is simply the voltage generated by the PLL filter. From *Table 3* it is obvious that some overlap exists between the various frequency ranges. This allows a choice of stage delays within some frequency ranges, which in turn allows system designers a choice of two different increments of phase adjustment.

Within the first thirteen stages of the oscillator, 11 taps are sent to the Output Adjust Matrix. These taps represent various phase relationships to the center, or 0 time unit (t_U) position. The taps range from $-6\ t_U$ to $+6\ t_U$, as shown in *Figure 29*. This allows the outputs to shifted, either early or late, with respect to the FB input to adjust for various system requirements.

The value of t_U shown in *Figure 30* is determined by the operating frequency and the number of stages in the distributed phase oscillator. The formula for calculating t_U is shown in *Table 3* and given here:

$$t_U = \frac{1}{f_{nom} \times N} \qquad \text{Eq. 23}$$

For example, *Equation 24* calculates the stage delay (t_U) when the ring oscillator is running at 25 MHz and the FS pin is tied to ground

$$t_U = \frac{1}{25MHz \times 44} = 0.91\ ns \qquad \text{Eq. 24}$$

The value of t_U, on the other hand, if the FS pin were left unconnected, is

$$t_U = \frac{1}{25MHz \times 26} = 1.54\ ns \qquad \text{Eq. 25}$$

This shows that at the same operating frequency (f_{NOM}), two different stage delays are possible, depending on the connection of the FS pin.

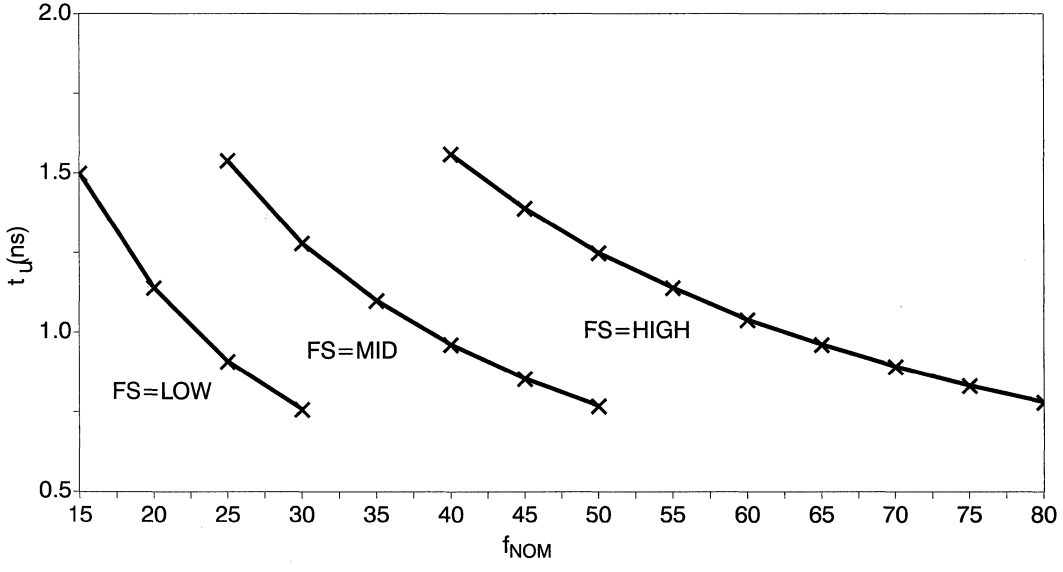


Figure 30. Time Unit (t_U) vs. Frequency

Output Adjust Matrix

The output adjust matrix allows the outputs to be configured in up to 26,000 ways (more on this later). The output options are generated by the Distributed Phase Clock Oscillator, and selected by the output function select (xFn) inputs.

In addition to the 11 taps from the distributed phase oscillator, the Output Adjust Matrix contains a divide-by-two option, a divide-by-four option, and an invert option.

The eight RoboClock outputs are configured as four output pairs. Each member of a pair of outputs operates identically to the other. The output adjustment for each output pair is controlled by its associated pair of function select inputs. For example, the 1Qn outputs are controlled by the 1Fn inputs.

The function select inputs are three-state inputs that operate in the same manner as the FS input. These inputs can be tied HIGH, tied LOW, or left unconnected (MID). The three-level input capabilities of the function select inputs allow each output to have nine different output selections with the use of only two pins.

Each pair of outputs has nine different possible output timing positions based on the appropriate connection of the function select input. The possible output combinations are shown in *Table 4.* These output adjustment configurations assume that an output with a 0 t_U configuration is used as the FB input. Output adjustment configurations with a non-0 t_U tap output selected as the FB input will be discussed in the section titled Change in Operation with FB selection.

The following example refers to *Figure 31. A*ssume that 2Q0 is used as the FB input and that 2F1 and 2F0 are both left unconnected. This will select both of the 2Qx outputs to have a 0-phase-adjusted output (0 t_U), and by connecting 2Q0 to the FB input it will also force these outputs to be phase and frequency aligned with the REF input.

Table 4. Output Adjustment Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	1F0, 2F0, 3F0, 4F0	1Q0, 1Q1, 2Q0, 2Q1	3Q0, 3Q1	4Q0, 4Q1
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

If, in this scenario, 1F1 were tied to ground and 1F0 were left unconnected, then the 1Q0 and 1Q1 output edges would precede the output used as the FB input (2Q0 in this case) by three time units. Alternately, if 1F1 were tied HIGH and 1F0 were again left unconnected, then the 1Q0 and 1Q1 output edge would follow the 2Q0 output by three time units.

If 3F1 and 3F0 were both tied HIGH, then the 3Qn outputs would both operate at one-quarter the frequency of the 2Qn outputs. And if the 4Fn function select inputs were both tied LOW, the 4Qn outputs would both operate at one-half the frequency of the 2Qn outputs.

An important point to note is the frequency and phase relationship between the 1/2 and 1/4 outputs

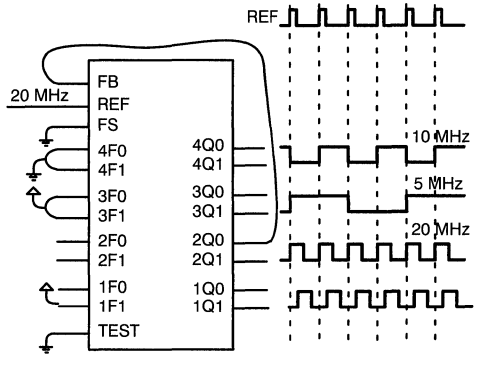


Figure 31. Frequency Divider Connections

(3Qn and 4Qn). The divide-by-two and divide-by-four outputs fall at the same time, but never rise at the same time. This feature of RoboClock makes it possible to use the rising edges of the 1/2 frequency and 1/4 frequency outputs without concern for skew mismatch. It also provides the ability to clock different parts of the system on different phases of the master clock.

The previous example showed the phase shifting and frequency division capabilities of RoboClock. Another output feature available on the 4Qx outputs is phase inversion. This output adjustment is configured by tying both 4F1 and 4F0 inputs HIGH. In this mode the 4Qx outputs will have and inverted sense with respect to the FB input.

Change in Operation with FB Selection

The previous discussion assumed that an output with a 0 t_U phase adjustment was used as the FB input. With this assumption, RoboClock has nearly 3000 different configurations. This is calculated by taking the number of possible configurations of each output pair (9) to the number of outputs pairs not being used as the FB input (3) times the number of choices of output pairs to be used as the FB input.

$$Combos = 9^3 * 4 \qquad \text{Eq. 26}$$

If the output used as the FB input is also phase or frequency adjusted, then RoboClock offers over 26,000 different configurations.

Phase Adjusted Output Used as FB Input

By feeding back an output that was selected for 0 t_U, all of the other outputs were referenced from the 0 tap position shown in *Figure 29*. This is due to the fact that the PLL aligns the REF input and the FB input in both phase and frequency.

It is not necessary to use an output with a 0 t_U configuration as the FB input. An output with any configuration can be used as the FB input. For example, if an output with a -3 t_U tap was used as the FB input, the PLL would align this output with the REF input. It would no longer exhibit a shift of -3 time units when compared with REF. The output used as the FB input is always aligned with REF.

By using an output with this configuration as the FB input, all other outputs are now referenced to this tap position within the ring oscillator. The possible tap selections for the other outputs are still the same as in the 0 tap used as FB case, but now they have a -3 tap time reference. The $+6$ tap can still be selected as an output configuration, but it will occur 9 time units after the FB output. The -6 tap can also be selected as an output configuration, but instead of occurring 6 time units before the corresponding edge of FB, it will occur 3 time units before the output used as the FB input.

Tables 5 through *7* illustrate the various possible output configurations with different FB selections. *Table 5* gives the 2Qn, 3Qn, and 4Qn output configurations when a 1Qn output is used as the FB input. It also gives the 1Qn, 3Qn, and 4Qn output configurations when a 2Qn output is used as the FB input. The reason for this is that the 1Qn and 2Qn outputs have the same possible configurations. If either is used as the FB input, the other outputs will have the same output configuration options.

Table 5 is broken into three parts corresponding to the configurations for each of the three pairs of outputs not used as the FB input. The leftmost two columns of each table indicate the various configurations of the FB input, and the right portion of the table gives the output possibilities for the given output.

For example, the first part of *Table 5* gives the possible output configurations for the 2Qn outputs assuming that a 1Qn output is used as the FB input. Alternately, this table gives the output configurations for the 1Qn outputs assuming a 2Qn output is used as the FB input. This is true because the 1Qn and 2Qn outputs have the same possible output configurations as shown in *Table 4*. For the remainder of this example, a 1Qn output is assumed to be the FB input.

The left side of the table gives the function select input settings for the FB output. L represents a connection to ground, M represents an input left open, and H represents an input connected to V_{CC}. Once a selection is made for the function select inputs of

the FB output, all of the other outputs will be referenced to that tap.

For example, if the 1F1 input is tied to ground and the 1F0 input is left unconnected, then all of the other outputs will be referenced to the -3 t_U tap used as the FB input. The available configurations on the remaining outputs will remain the same as given in *Table 4*, but the values in this table will all be shifted by +3 because this is the number of stage delays between the new reference point and 0 t_U, the reference point of *Table 4*.

All of the possible output configurations can be found in the same row in the following tables as the selection made for the FB output. If 1Fn = LM (1F1 tied to ground, and 1F0 left unconnected), then the possible selections for the 2Qn output are from -1 t_U to $+7$ t_U. This is shown in the first part of *Table 5* as a shaded row. By connecting 2Fn = HM, the 2Qn outputs will have outputs that lag the FB outputs by 6 time units or by connectng 2Fn = LL the 2Qn outputs will precede the reference by 1 time unit (-1t). Once the output configurations for the FB input are made, all other outputs will be referenced to this new reference point.

The second part of *Table 5* gives the possible output configurations for the 3Qn outputs. Assuming a 1Qn (2Qn) output is used for the FB input, the 3Qn outputs can be phase shifted with a granularity of 2 time units from -3 to +9 with respect to the FB input. Additionally, the 3Qn outputs can be divided by two and divided by four, but since the reference point for the dividing circuit for these configurations is the 0 tap (as shown in *Figure 29*), these outputs are shifted by three time units with respect to the FB input.

The third part of *Table 5* gives the possible output configurations for the 4Qn outputs, again assuming that a 1Qn output is used as the FB input. This table looks much the same as the second part with the only exception being the last column. The only difference between the 3Qn and 4Qn outputs is the ability to divide by four or invert respectively. The last column shows how RoboClock can be configured to phase shift and invert an input signal.

Table 5. 1Qx or 2Qx Output Connected to FB Input (Part 1)

1Qn(2Qn)➧FB		2Qn(1Qn) Outputs with respect to FB									
		Input Select	2F1 (1F1)	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		2F0 (1F0)	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
L	M			−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t
L	H			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	L			−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t
H	L			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	M			−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t
H	H			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t

Table 5. 1Qx or 2Qx Output Connected to FB Input (Part 2)

1Qx(2Qx)➧FB		3Qn Outputs with respect to FB										
		Input Select	3F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t f/4
L	M			+3t f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t f/4
L	H			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t f/4
M	L			+1t f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t f/4
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t f/4
M	H			−1t f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t f/4
H	L			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t f/4
H	M			−3t f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t f/4
H	H			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t f/4

Table 5. 1Qx or 2Qx Output Connected to FB Input (Part 3)

1Qn(2Qn)➧FB			4Qn Output with respect to FB								
		Input Select	4F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t INV
L	M			+3t f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t INV
L	H			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t INV
M	L			+1t f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t INV
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t INV
M	H			−1t f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t INV
H	L			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t INV
H	M			−3t f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t INV
H	H			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t INV

Tables 6 and *7* have slightly different output configurations. These tables represent the possible output configurations when a 3Qn or 4Qn output is used as the FB input.

The first part of *Table 6* gives the possible output configurations for the 1Qn (2Qn) outputs when a 3Qn output is used as the FB input. When the 3Qn outputs are configured from $-6\ t_U$ (3Fn = LM) to $+6\ t_U$ (3Fn = HM) the 1Qn outputs have a range of $+10\ t_U$ (1Fn = HH) to $-10\ t_U$ (1Fn = LL). This, again, is because if the 3Qn outputs have a −6 tap reference and the +4 tap is selected for the 1Qn outputs, then the total time delay between the FB input and the 1Qn output is +10 time units. This feature gives RoboClock a tremendous phase adjustment range.

What if a divided output is used as the FB input? The last row of *Table 6* (Part 1) shows that if 3Fn = HH (3Qx in divide-by-four mode), then all of the outputs are multiplied by four. RoboClock has become a frequency multiplier. To understand why this happens, remember that the PLL aligns the FB with the REF input in both phase and frequency. Even though the 3Qn outputs were selected to divide by four, the PLL forces them to run at the same rate as the REF input. This means that, in order for these outputs to operate at this speed, the VCO itself must operate at four times the REF frequency.

The ability to multiply an input frequency is useful in board-level designs where the distribution of a low-frequency signal is needed to reduce EMI emissions or where a faster clock is needed to increase the operating frequency of state machine logic. *Figure 32* shows how RoboClock can be configured as a frequency multiplier and a phase adjuster. By selecting 3Fn = HH, RoboClock will multiply the

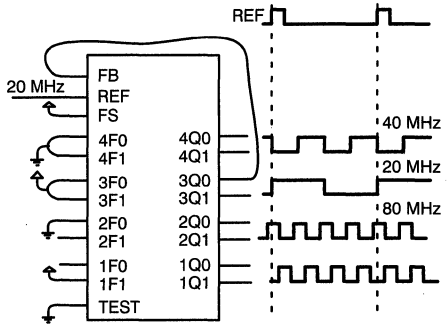


Figure 32. Frequency Multiplier with Phase Adjustment

REF frequency by four, by forcing its PLL to operate at four times the REF clock rate. 4Fn = LL selects the divide-by-two option at the 4Qn outputs. Since the PLL is operating at 80 MHz, the 4Qn outputs will operate at 40 MHz. Selecting 2Fn = ML configures the 2Qn outputs to precede the rising edge of the FB input by 1 time unit. And selecting 1Fn = HM makes the 1Qn outputs arrive 3 time units later than the FB input. Both the 1Qn and 2Qn outputs will run at 80 MHz. The xFn configurations for this example can be found in the shaded area of *Table 6*.

Table 7 appears very similar to *Table 6*. The first part gives the 1Qn and 2Qn output configurations when a 4Qn output is used as the FB input. The second part of the table gives the 3Qn output configurations when a 4Qn output is used as the FB input. The major variation is that if a 4Qn output is used as the FB input with 4Fn = HH, then all outputs will be inverted. Since the PLL phase aligns the REF and FB input, the 4Qn outputs will operate identically with the REF input. The other outputs will have a 180°phase shift from the REF input. This is useful for applications requiring more inverted clock signals than non-inverted clock signals.

Table 6. 3Qx Output Connected to FB Input (Part 1)

3Qn ⇩ FB		1Qn (2Qn) Output with respect to FB										
		Input Select	1F1, (2F1)	L	L	L	M	M	M	H	H	H
3F1	3F0		1F0, (2F0)	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		−4t f*2	−3t f*2	−2t f*2	−1t f*2	0t f*2	+1t f*2	+2t f*2	+3t f*2	+4t f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t f*4	−3t f*4	−2t f*4	−1t f*4	0t f*4	+1t f*4	+2t f*4	+3t f*4	+4t f*4

Table 6. 3Qx Output Connected to FB Input (Part 2)

3Qn ➧ FB		4Qn Output with respect to FB										
		Input Select	4F1	L	L	L	M	M	M	H	H	H
3F1	3F0		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		0t	−6t f*2	−4t f*2	−2t f*2	0t f*2	+2t f*2	+4t f*2	+6t f*2	INV f*2
L	M			+6t f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t INV
L	H			+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t INV
M	L			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t INV
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t INV
M	H			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t INV
H	L			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t INV
H	M			−6t f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t INV
H	H			0t f*2	−6t f*4	−4t f*4	−2t f*4	0t f*4	+2t f*4	+4t f*4	+6t f*4	INV f*4

Table 7. 4Qx Output Connected to FB Input (Part 1)

4Qn ➧ FB		1Qn (2Qn) Output with respect to FB										
		Input Select	1F1, (2F1)	L	L	L	M	M	M	H	H	H
4F1	4F0		1F0, (2F0)	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		−4t f*2	−3t f*2	−2t f*2	−1t f*2	0t f*2	+1t f*2	+2t f*2	+3t f*2	+4t f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t INV	−3t INV	−2t INV	−1t INV	0t INV	+1t INV	+2t INV	+3t INV	+4t INV

Table 7. 4Qx Output Connected to FB Input (Part 2)

4Qn ➧ FB		3Qn Output with respect to FB									
		Input Select	3F1	L	L	L	M	M	M	H	H	H
4F1	4F0		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		0t	−6t f*2	−4t f*2	−2t f*2	0t f*2	+2t f*2	+4t f*2	+6t f*2	0t f/2
L	M			+6t f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t f/4
L	H			+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t f/4
M	L			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t f/4
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t f/4
M	H			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t f/4
H	L			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t f/4
H	M			−6t f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t f/4
H	H			INV f/2	−6t INV	−4t INV	−2t INV	0t INV	+2t INV	+4t INV	+6t INV	INV f/4

Functional Implementations

Obviously, RoboClock has abilities to solve even the most complex problems. The challenge is to determine how to configure RoboClock to solve these problems. The following examples will give a brief overview of how to configure RoboClock to accomplish various tasks.

Low-Skew Clock Buffer

The easiest way to configure RoboClock is as a low-skew clock buffer, as shown in *Figure 33.* In this type of configuration, all xFn inputs are left open (unconnected) and the FB input can be taken from any output. The REF input waveform is shown with a very unequal duty cycle to illustrate the point that because RoboClock is a PLL-based clock buffer, the duty cycle of the outputs is 50/50 regardless of the duty cycle of the REF input. This feat would be impossible in a non-PLL based device.

If this implementation is used, either the 2Q1 or 3Q0 output is the most convenient choice to be used as the FB input because of their proximity. The shortest wire possible should be used to connect the xQn output to the FB input to avoid stub reflection effects on the actual clock line connected to the

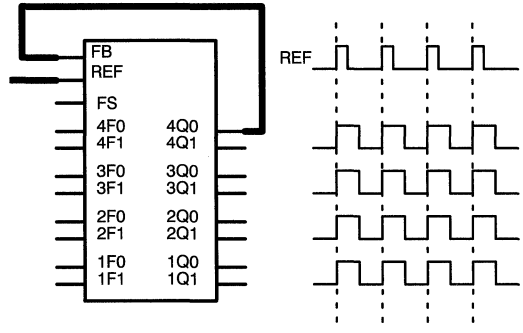


Figure 33. Low-Skew Clock Buffer

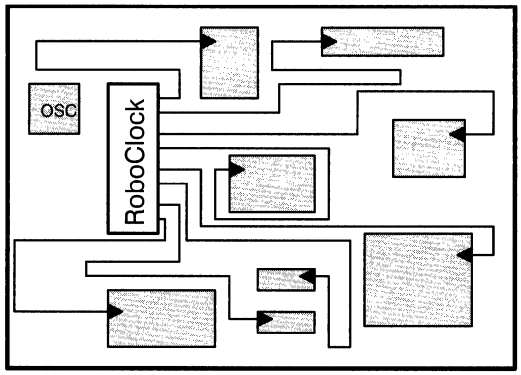


Figure 34. RoboClock in Low Skew Buffer Application

same xQn output *(Figure 2)*. This short route will reduce noise and transmission line effects from affecting the FB input.

This configuration assumes that the clock routes to the various loads they are driving are all the same length, so that each of the clocks arrives at its load at virtually the same time. *Figure 34* shows how RoboClock might be used in an application requiring only a device with low output skew.

Programmable Phase Adjustment

This type of application requires the use of RoboClock's additional features. RoboClock provides phase shifting in ranges from −12 time units to +12 time units. *Figure 35* shows RoboClock configured to phase shift its 2Qn outputs so that they lag the FB input, and to phase shift its 1Qn outputs so that they lead the FB input. Any output can be used as the FB input to implement phase shifting, but the greatest unidirectional shift (12 time units) will be achieved by selecting either the 3Qn or 4Qn outputs to be used as the FB input.

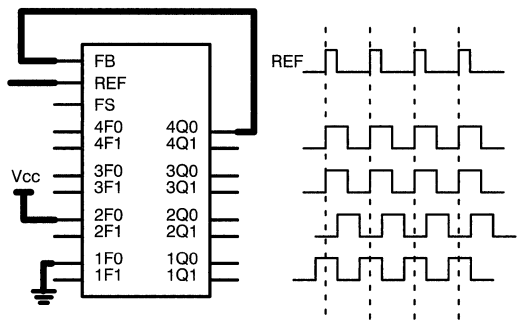


Figure 35. Programmable Phase Adjustment

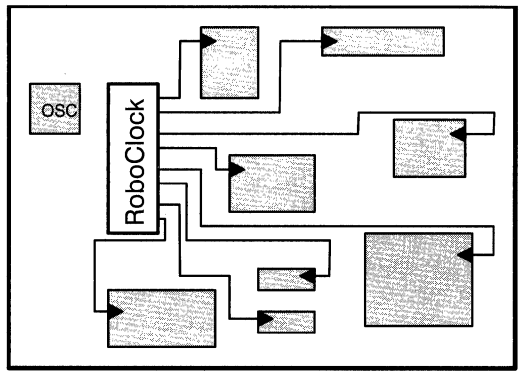


Figure 36. RoboClock in Programmable Phase Adjustment Application

Figure 36 shows RoboClock in a programmable phase adjustment application. This application differs from *Figure 34* in that the sophisticated phase adjustment abilities of RoboClock are used to compensate for trace delays and set-up and hold time mismatches.

Inverted Output Clock Driver

Figure 37 shows RoboClock configured as a inverted output clock driver. 4Fn = HH configures the 4Qn

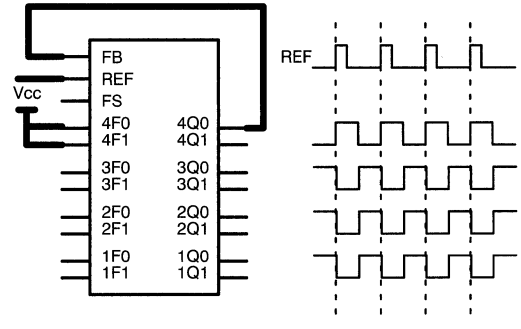


Figure 37. Inverted Output Clock Driver

outputs to operate in inverted mode. If one of these outputs is used as the FB input, then the PLL will align the rising edge of 4Q0 with REF and all of the other outputs operate 180° phase shifted from the FB input. This type of configuration is useful for system designs that require a greater number of clocks with 180° phase shift (inverted). These inverted clocks are useful for clocking logic at twice the frequency without distributing a higher frequency clock. In this configuration, RoboClock offers 6 inverted outputs and 2 non-inverted outputs. This configuration also has the advantage that all of the inverted outputs can be phase shifted and the 3Qn outputs can even be configured to divide the REF frequency by two or four while still maintaining phase inversion.

If only two inverted clocks are needed, then any output except the 4Qn outputs can be connected to the FB input. This will allow the 4Qn outputs to be selected for output inversion without affecting the other outputs. If the REF clock is not being used in other parts of the system, both of these two configuration options yield the same net effect.

Frequency Divider

RoboClock provides frequency division while still maintaining very low skew between the various output edges (more on this in the AC Specifications section). *Figure 38* shows RoboClock configured as a frequency divider. Since the PLL is operating at 20 MHz, the FS pin in this configuration is tied to ground indicating the selection of the 15− to 30-MHz operating range. By selecting 4Fn = LL, the 4Qn outputs will divide the FB frequency by two, and by selecting 3Fn = HH, the 3Qn outputs will divide the FB frequency by four.

Any of the 1Qn or 2Qn outputs could have been selected as the FB input with equivalent results. Note again that the divide-by-two and divide-by-four outputs have no coincidental rising edges. This features allows the system designer to use both outputs for multiphase clocking without concern for skew requirements between the rising edges of these two outputs.

Frequency Multiplier

RoboClock provides frequency multiplication by selecting an output configured to divide by two or divide by four as the FB input. *Figure 39* shows RoboClock multiplying up the REF input frequency. The 3Fn function select inputs are both tied to V_{CC}, configuring the 3Qn outputs to divide by four. But because 3Q0 is used as the FB input, the 3Qn outputs operate at the same frequency as the REF input. The RoboClock PLL, therefore, now operates at four times the REF frequency. This is the reason that the FS pin is tied to V_{CC} indicating the selection of the fastest operating frequency range. The selection of FS is based not on the operating frequency of the REF input, but rather on the operating frequency of the VCO. The operating frequency of the VCO is the same frequency as the 1Qn and 2Qn outputs at all times and also the 3Qn and 4Qn outputs when frequency division is not selected.

The 4Fn function select inputs are both tied to ground, which configures the 4Qn output for divide by two mode. Note that this figure looks much the same as *Figure 38* except for the selection of the FS

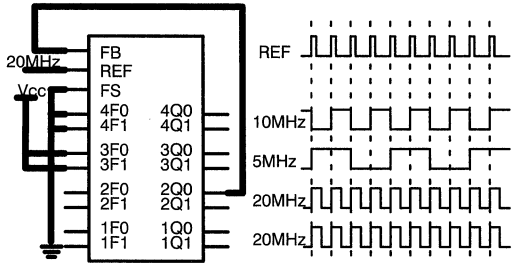


Figure 38. Frequency Divider

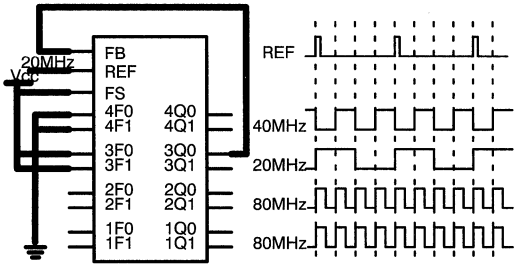


Figure 39. Frequency Multiplier

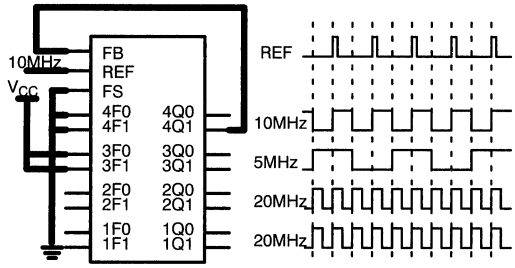


Figure 40. Frequency Divider and Multiplier

input and the selection of the output used as the FB input.

Frequency Divider and Multiplier

Figure 40 illustrates how RoboClock can be used to both multiply and divide the REF frequency. Here the VCO is running at 20MHz. The 4Qn output is used to divide the PLL frequency by two and the 3Qn outputs are used to divide the PLL frequency by four. 4Q1 is used as the FB input doubling the PLL rate.

Multi-Function Clock Driver

RoboClock is truly a multi-function clock driver. It has the ability to multiply up the REF frequency by two or four, divide down the REF frequency by two or four, perform phase inversion, create phase adjustments up to ±12 time units, while always providing very low output skew.

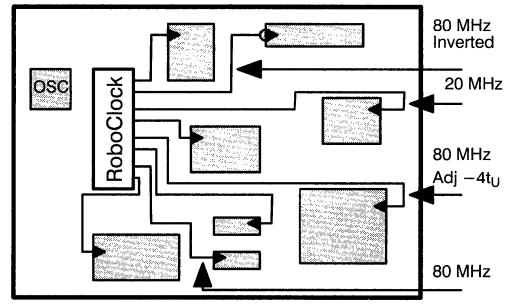


Figure 42. Multi-Function Clock Buffer Application

Figure 41 shows RoboClock configured as a multi-function clock driver. This figure shows how Roboclock can simultaneously multiply the REF frequency by four to a speed of 80 MHz and allow these high-speed outputs to be phase shifted and inverted. *Figure 42* shows how this configuration can be integrated to perform various system clocking functions.

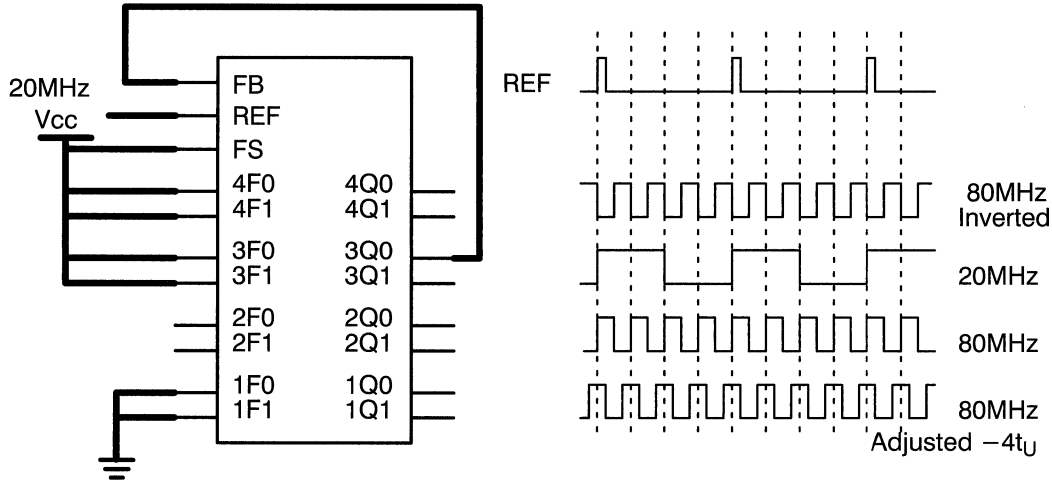


Figure 41. Multi-Function Clock Buffer

Many other examples exist of the tremendous capabilities of RoboClock. External circuitry, for example, can be placed between the RoboClock output and the FB input. This may be the case when RoboClock is used to drive external buffers, frequency dividers, or multipliers. Virtually any delay element can be placed between a RoboClock output and the FB input. A few things, however, must be remembered when doing this:

- The FB input must always be function of one of the RoboClock outputs.
- The other outputs will be referenced to the input of the external device and the delay may not be well defined due to the timing characteristics of the external devices.
- The output edge placement will be dependent on the function and skew characteristics of the external device.
- An external divider of no greater than 16 may be used in the FB path (jitter specifications may be compromised).

AC Specifications

Many AC Specifications exist for RoboClock. The following discussion will explain what these specifications mean to the designer. There are four different parts in the RoboClock family. The CY7B991 is a TTL-output (0 to 3V swing) device and the CY7B992 is a CMOS output (0 to V_{CC} swing) device. The CY7B99x–5 is the lowest skew device that Cypress offers. Its output skew has a typical value of just 250 ps with a maximum skew of 500 ps while the CY7B99x–7 has a maximum output skew of only 750 ps. The following sections will begin with the relevant datasheet specification. Only the CY7B99x–5 specifications will be explained, but all specifications apply directly to the CY7B99x–7 devices except for parameter value. Also, only the CY7B991–5 device will be explained unless the CY7B992–5 device has a different specification.

f_{NOM}: Operating Clock Frequency in MHz

Parameter	Description	CY7B99x			
		Min.	Typ.	Max.	Unit
f_{NOM}	FS = LOW	15		30	ns
	FS = MID	25		50	ns
	FS = HIGH	40		80*	ns

* The maximum operating frequency of the 7B992–7 devices is 50 MHz.

This parameter indicates the frequency range relative to given selections of the FS pin. The three-level FS input can either be tied to ground (LOW), left unconnected (MID), or tied to V_{CC} (HIGH).

The operating frequency these ranges refer to is based on the operating frequency of the VCO. If a divide-by-two output is used as the FB input, then the operating frequency of the VCO is twice that of the REF input. If a divide-by-four output is used as the FB input, then the operating frequency of the VCO is four times that of the REF input. If a non-divided output is used as the FB input then the selection of the FS pin can be made based upon the operating frequency of the REF input.

For example if a divide-by-four output is used as the FB input then the total possible REF input frequency range is 3.75 to 20 MHz (¼*15 to ¼*80 MHz). And the frequency range of the REF input when the FS pin is tied to ground, in particular, is 3.75 to 7.50 MHz.

t_{RPWH}, t_{RPWL} REF Pulse Width High and Low

Parameter	Min.	Typ.	Max.	Unit
t_{RPWH}	5.0			ns
t_{RPWL}	5.0			ns

The frequency and phase detector uses only the rising edge of the REF and FB inputs for alignment purposes. RoboClock, therefore, does not require a 50/50 duty cycle clock. The t_{RPWx} parameter is measured at the 1.5V level.

t_{PD} : Propagation Delay, REF Rise to FB Rise

Parameter	CY7B99x–5			
	Min.	Typ.	Max.	Unit
t_{PD}	– 0.5	0.0	+0.5	ns
Parameter	**CY7B99x–7**			
	Min.	**Typ.**	**Max.**	**Unit**
t_{PD}	– 0.7	0.0	+0.7	ns

Because of the PLL architecture of RoboClock there is no true propagation delay through the device as there is in a device such as a 74F244 buffer. The outputs of RoboClock are independent of the wave shape or duty cycle of the REF input. The PLL uses the REF and FB pins to generate the outputs by aligning these two inputs in both phase and frequency.

This static misalignment (t_{PD} in the datasheet) may be either positive or negative and is a function of the REF frequency. The ±500 ps specification is to accommodate the normal variation in process, voltage, temperature, and frequency. Two parts operating at the same frequency and similar voltage and temperature (approximately ±100 mV and ±10°C) will never have a t_{PD} variation more than 200 ps while the total magnitude of either might be 500 ps). This part-to-part variation appears as t_{SKEW5} in the datasheet and is discussed below. While the PLL aligns the REF and FB pins, some time difference may exist between the REF and FB pin (See *Figure 43*).

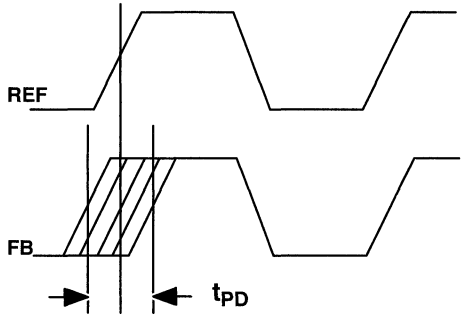


Figure 43. Propagation Delay

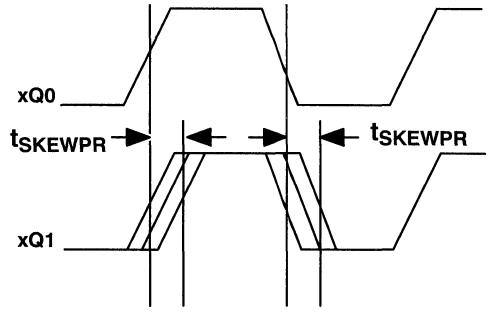


Figure 44. Zero-Output Matched Pair Skew

t_{SKEW} : Output Skew

There are six different parameters specifying output skew. Each of these parameters relates to different output configurations and to different output edges. Each of the following sections will describe the particular skew parameter with a diagram and include an example showing when this parameter would be used to calculate output skew.

t_{SKEWPR} : Zero Output Matched-Pair Skew

Parameter	Min.	Typ.	Max.	Unit
t_{SKEWPR}		0.1	0.25	ns

This parameter specifies the maximum amount of skew between two outputs of the same pair (e.g., 1Q1 and 1Q0) when all output are configured for 0 t_U. This specification has a maximum value of 250 ps. Bench characterization, however, indicates that t_{SKEWPR} is rarely greater than 100 ps. The additional margin is included for tester guard band. The reason that both outputs of the same pair can be so tightly coupled is that each pair has a dedicated power and ground pin, that they lie adjacent to each other allowing them to reinforce each other with crosstalk effects, and that they are separated from adjacent outputs by at least two non-switching pins.

Figure 44 shows that skew is measured from the first output to the last output and that t_{SKEWPR} as well as all other skew parameters pertains to both edges of the output waveform. *Figure 45* shows the 4Qn output pair loaded with the datasheet load. Although it may look like one output waveform, both outputs of the 4Qn pair are displayed. *Figure 46* shows the measurement of t_{SKEWPR} with an expanded voltage and time scale as being only 27 ps.

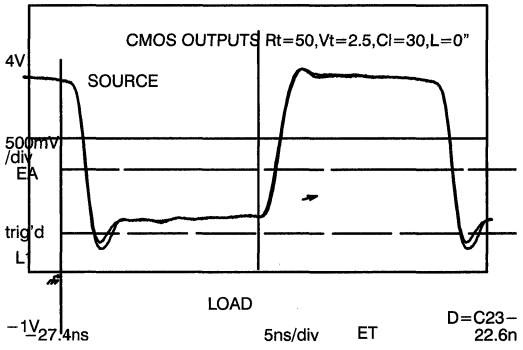


Figure 45. Lumped Load with Datasheet Load

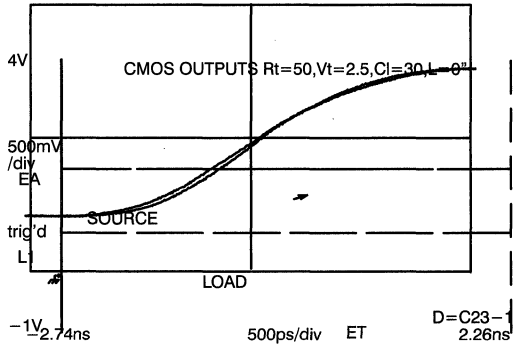


Figure 46. t_{SKEWPR} Measurement

t_{SKEW0} : Zero Output Skew

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW0}		0.25	0.5	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW0}		0.3	0.7	ns

t_{SKEW0} is the maximum skew between the first output edge and the last output edge of all outputs when all outputs are configured for 0 t_U. This specification also applies to outputs that are not adjusted when another output is divided-by-two or divided-by-four. Bench data indicates that these skew values are usually no greater than 350 ps.

For example the skew between any of the outputs in *Figure 33* is no greater than 500 ps. The maximum skew between the 1Qn and 2Qn outputs in *Figure 38* is also 500 ps, even though both the 3Qn and 4Qn outputs are divided. This assumes that a CY7B99x−5 is used to generate these signals.

t_{SKEW1} : Output Skew (Rise-Rise, Fall-Fall, Same Class Outputs)

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW1}		0.25	0.5	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW1}		0.3	0.7	ns

t_{SKEW1} is specified as the maximum amount of skew between outputs of the same output class selected for the same output adjustment without restrictions on the placement or function of other outputs. For the purposes of skew specification, there are three types of output classes:

- *Nominal*: Outputs that are selected for phase adjustment, but not inversion, division, or multiplication are of the Nominal output class. This also includes outputs that are not phase adjusted when other outputs have a non $0t_U$ configuration.
- *Divided*: This class of outputs includes the 3Qn and 4Qn outputs that are configured for divide-by-two or -four mode (3Fn=HH, or LL, and 4Fn = LL). Even when these outputs are configured this way and selected as the FB input, the are still considered to be part of the Divided output class.
- *Inverted*: This class of outputs includes the 4Qn outputs configured in the inverted mode of operation (4Fn = HH). The Inverted output class also applies to a 4Qn output configured for phase inversion and used as the FB input.

This parameter, as in the case of $t_{SKEWPR, 0, and 3}$ applies not only to rising edge to rising edge output skew, but also to falling edge to falling edge skew.

Figure 47 illustrates a an example of when to use t_{SKEW1} to calculate output skew. The maximum skew between the 1Qn and 2Qn outputs, when they are selected in this case for −4 time unit adjustment, is no greater than 700 ps. The maximum skew be-

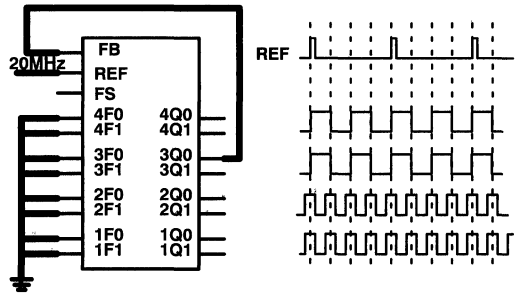


Figure 47. Multi-Function Clock Buffer

tween the 4Qn and 3Qn when both are selected for divide−by−two is also 700 ps.

Figure 48 shows a diagram of an output (Adjusted Q) that has been programmed to occur N time units (t_U) later than another output (Q). The maximum amount of time between these two outputs will be

$$t_{DIFF} = N \times t_U + t_{SKEW1} \quad \text{Eq. 27}$$

and the minimum amount of difference between these two outputs is

$$t_{DIFF} = N \times t_U - t_{SKEW1} \quad \text{Eq. 28}$$

Where N is difference in the calculated tap delays between these two outputs. There is no need to add twice the t_{SKEW1} time in order to calculate the minimum and maximum time difference.

For example, the minimum and maximum time between the 1Qn and 2Qn outputs in a system configured as in *Figure 35* using a CY7B991−5 operating at 50 MHz would be

$$t_{DIFF(min)} = 2 * \left(\frac{1}{50 * 10^6 \times 26}\right) - .5 = 1.04\ ns \quad \text{Eq. 29}$$

$$t_{DIFF(max)} = 2 * \left(\frac{1}{50 * 10^6 \times 26}\right) + .5 = 2.04\ ns \quad \text{Eq. 30}$$

The time difference between the 3Qn and 4Qn outputs (which were not phase adjusted) and the 1Qn and 2Qn (which were phase adjusted in opposite directions) would be

$$t_{DIFF(min)} = \left(\frac{1}{50 * 10^6 \times 26}\right) - .5 = .269\ ns \quad \text{Eq. 31}$$

$$t_{DIFF(max)} = \left(\frac{1}{50 * 10^6 \times 26}\right) + .5 = 1.269\ ns \quad \text{Eq. 32}$$

These equations give the worst-case time value between outputs skewed by 1 time unit when operating at 50 MHz. No additional output skew parameters need to be added. The ordering, including skew, between two outputs that are phase adjusted, can always be determined from the functional input selections. This is shown in the above example where the minimum time between two output adjusted by one time unit was determined to be 269 ps.

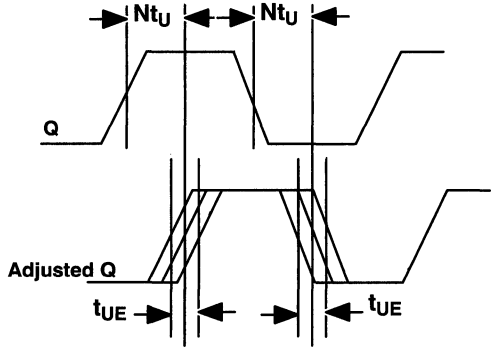


Figure 48. Programmable Adjustment Error

t_{SKEW2}: Output Skew (Rise-Fall, Nominal-Inverted, Divided-Divided)

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW2}		0.6	1.2	ns

Parameter	CY7B99x−7 Min.	Typ.	Max.	Unit
t_{SKEW2}		1.0	1.5	ns

This skew parameter specifies the amount of output skew between the rising or falling edge of a Nominal output and the opposite edge of an Inverted output as generalized in *Figure 49*. This parameter also applies to opposite edge transitions between Divided outputs.

For example, in *Figure 39,* the output skew between the opposite edge transitions of the the 3Qn and 4Qn outputs selected for divided mode is no greater than 1.2 ns. The magnitude of this number compensates for the different in the rising and falling edge rates.

t_{SKEW3} : Output Skew (Rise-Rise, Fall-Fall, Different Class Outputs)

Parameter	CY7B99x-5 Min.	Typ.	Max.	Unit
t_{SKEW3}		0.6	1.0	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW3}		0.7	1.2	ns

This output skew parameter specifies the maximum same edge transition difference between different class outputs. In *Figure 39*, the difference between the rising edge of the 4Qn or 3Qn outputs that are configured for divided mode and the rising edge of the 1Qn and 2Qn outputs will be no greater than 1 ns.

t_{SKEW4} : Output Skew (Rise-Fall, Nominal-Divided, Divided-Inverted)

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW4}		0.6	1.3	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW4}		1.2	1.7	ns

This output skew parameter specifies the maximum opposite edge transition difference between different class outputs as generalized in *Figure 49*. In *Figure 39,* the maximum difference between the opposite edge transition of the 4Qn or 3Qn outputs and the 1Qn and 2Qn outputs would be no greater than 1.3 ns

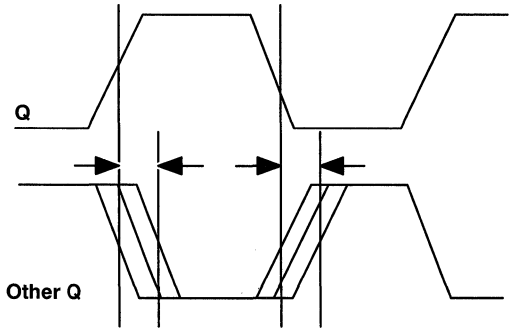


Figure 49. t_{SKEW2} and t_{SKEW4} Measurement

t_{SKEW5} : Device-to-Device Output Skew

Parameter	CY7B99x Min.	Typ.	Max.	Unit
t_{SKEW5}			0.2	ns

Taken by itself, this parameter has very little meaning in a system design. It must be used in conjunction with the other output skew parameters discussed above. This parameter states that the maximum variation in t_{PD} between two devices operating at the same frequency, temperature, and voltage will be no greater than 200 ps. This means that the device to device skew between two outputs used as the FB input will be no greater than 200 ps under these circumstances.

Figure 50 (an adapted version of *Figure 39*) shows two RoboClock devices configured as in *Figure 39* connected in parallel. To calculate the device to device skew for outputs configured for 0 t_U use the t_{SKEW0} parameter of each device plus t_{SKEW5} as in the following equation:

$$t_{SKEW} = t_{SKEW0(1)} + t_{SKEW0(2)} + t_{SKEW5}$$
$$t_{SKEW} = 0.5 + 0.5 + 0.2 = 1.2\ ns \qquad \text{Eq. 33}$$

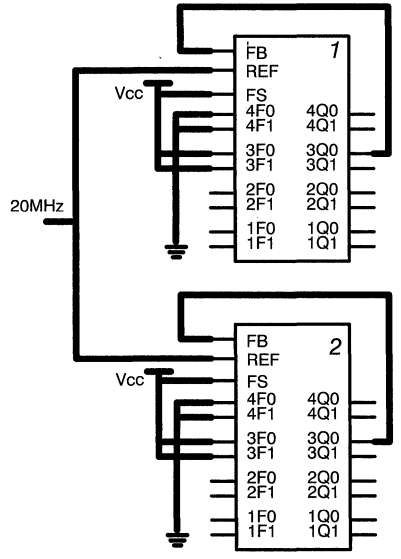


Figure 50. Part-to-Part Skew Example

This will give the output skew between the 2Qn and 1Qn output skew between devices 1 and 2. To calculate the output skew between the rising edges of the divided outputs and the 0-phase-adjusted outputs, use the following equation:

$$t_{SKEW} = t_{SKEW3(1)} + t_{SKEW3(2)} + t_{SKEW5}$$
$$t_{SKEW} = 1.0 + 1.0 + 0.2 = 2.2ns \quad \text{Eq. 34}$$

In general, to calculate the skews between two devices, add the relevant skew component from each device and add 200 ps.

Figure 51 gives a slightly more complicated example to illustrate how to calculate device-to-device skew. Device 3 has been added to the previous example. The outputs of this device are driving the REF inputs of devices 1 and 2. The only additional component to add to the output to output skew for devices 1 and 2 is the output-to-output skew of device 3. The part-to-part skew between devices 1 and 2 for outputs that are left unconfigured (1Qn and 2Qn) is now

$$t_{SKEW} = t_{SKEW0(3)} + t_{SKEW1(1)} + t_{SKEW1(2)} + t_{SKEW5}$$
$$t_{SKEW} = 0.5 + 0.5 + 0.5 + 0.2 = 1.7ns \quad \text{Eq. 35}$$

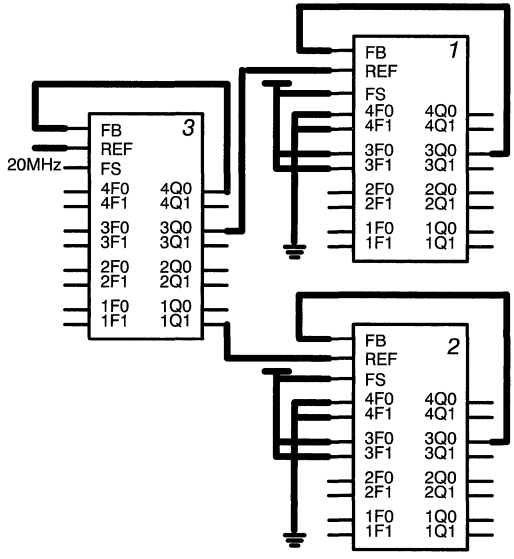


Figure 51. Devices in Parallel and Serial

t_{ODCV} : Output Duty Cycle Variation

Parameter	CY7B99x–5 Min.	Typ.	Max.	Unit
t_{ODCV}	– 1.0	0.0	+1.0	ns

Parameter	CY7B99x–5 Min.	Typ.	Max.	Unit
t_{ODCV}	– 1.2	0.0	+1.2	ns

This parameter specifies the difference in the output duty cycle from 50%. It is measured at 1.5V. This parameter indicates, for example, that the outputs of the CY7B99x–5 have a worst-case duty cycle of 42/58, and at 15 MHz the duty cycle is 48.5/51.5 worst case. The AC Characterization section contains data on how the value of this parameter varies with loading, voltage, and temperature.

t_{ORISE}, t_{OFALL} : Output Rise and Fall Time

Parameter	CY7B991–5 Min.	Typ.	Max.	Unit
t_{ORISE}	0.15	1.0	1.5	ns
t_{OFALL}	0.15	1.0	1.5	ns

Parameter	CY7B992–5 Min.	Typ.	Max.	Unit
t_{ORISE}	0.5	2.0	2.5	ns
t_{OFALL}	0.5	2.0	2.5	ns

Parameter	CY7B991–7 Min.	Typ.	Max.	Unit
t_{ORISE}	0.15	1.5	2.5	ns
t_{OFALL}	0.15	1.5	2.5	ns

Parameter	CY7B992–7 Min.	Typ.	Max.	Unit
t_{ORISE}	0.5	3.0	5.0	ns
t_{OFALL}	0.5	3.0	5.0	ns

The output rise and fall time are measured with different loads for each of the four devices. The CY7B991 devices (TTL) are tested with the load shown in *Figure 52*. The 130Ω over 91Ω load is the recommended parallel termination for 50Ω transmission lines. For the –7 devices a 50-pF load is used to test all AC parameters for the –5 devices a 30-pF load is used. *Figure 53* shows the CMOS device test load (CY7B992).

The TTL and CMOS devices are also measured between different voltage levels. *Figure 54* shows that

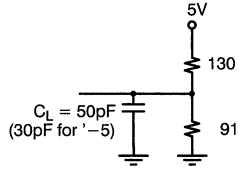


Figure 52. TTL AC Test Load

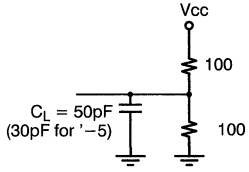


Figure 53. CMOS AC Test Load

the TTL rise and fall time parameters are measured between 0.8 and 2.0V, and *Figure 55* shows that the CMOS rise and fall times are measured between 20% and 80% of V_{CC}.

The outputs edge rates of these devices are controlled to about 1 V/ns to minimize the generation of system noise and transmission line effects. The AC Characterization section will give examples of how this parameter varies with loading, voltage, and temperature.

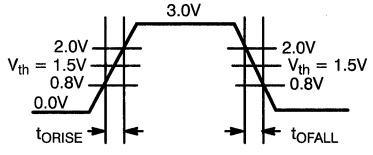


Figure 54. TTL Output Voltage Levels

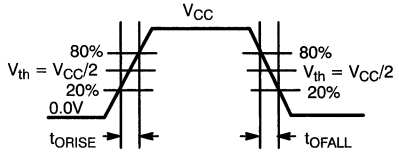


Figure 55. CMOS Output Voltage Levels

t_{PWH}, t_{PWL} : Output High and Low Time Deviation from 50%

Parameter	CY7B991–5 Min.	Typ.	Max.	Unit
t_{PWH}			2.5	ns
t_{PWL}			3	ns
Parameter	**CY7B992–5 Min.**	**Typ.**	**Max.**	**Unit**
t_{PWH}			3.5	ns
t_{PWL}			3.5	ns
Parameter	**CY7B991–7 Min.**	**Typ.**	**Max.**	**Unit**
t_{PWH}			3	ns
t_{PWL}			3.5	ns
Parameter	**CY7B992–7 Min.**	**Typ.**	**Max.**	**Unit**
t_{PWH}			5.5	ns
t_{PWL}			5.5	ns

The output pulse width high and low times are specified as deviations from an ideal 50/50 duty cycle. t_{PWH} is measured above the 2.0V (80% V_{CC}) for the TTL (CMOS) devices and t_{PWL} is measured below 0.8V (20% V_{CC}) for the TTL (CMOS) devices. The value of these parameters can be calculated from the combination of the t_{ODCV} parameter and the t_{ORISE} and t_{OFALL} parameters. The specifications can be calculated as follows

$$t_{PWH} = 2 * t_{OR/OF} * \left(\frac{V_{MAX} - V_{TH}}{V_{MAX} - V_{MIN}}\right) + t_{ODCV} \quad \text{Eq. 36}$$

$$t_{PWL} = 2 * t_{OR/OF} * \left(\frac{V_{TH} - V_{MIN}}{V_{MAX} - V_{MIN}}\right) + t_{ODCV} \quad \text{Eq. 37}$$

Where t_{OROF} represents either the rise time or fall time of the output since they are equal, V_{TH} represents the measurement threshold (TTL = 1.5V and CMOS = V_{CC}/2), V_{MAX} represents the maximum voltage point of rise and fall time measurements and V_{MIN} represents the minimum voltage point for rise and fall time measurements.

AC Characterization

Included with this application note are output rise time, output fall time, and output duty cycle varia-

tion versus temperature, voltage, capacitive loading, and termination voltage.

Output Rise Time

As explained previously, output rise time (t_{ORISE}) is the maximum amount of time it takes the output to rise from the 0.8V to the 2.0V level.

Figure 56 shows the variation in output rise time based on capacitive loading. This graph can be used to calculate the skew caused by the unequal loading of outputs.

Figure 57 shows the output rise time versus voltage over temperature. Within the normal operating limits of RoboClock (4.5 to 5.5V), the output rise time varies very little with respect to temperature. This means that, within a normal board environment, output rise time will not significantly vary due to minor variations in temperature or voltage.

Figure 58 shows the output rise time vs. termination voltage. In the section entitled Transmission Line Termination the parallel termination Thevenin voltage was specified as 2.06V for the CY7B991–x devices. This figure shows that because output rise time does not vary with termination voltage, convenient resistor value can be chosen that maintain $R_{TH} = Z_0$ within the limits of $1.5V < V_{TH} < 2.5V$.

Output Fall Time

Output fall time is the amount of time it takes for the output to swing from 2.0V to 0.8V.

Figure 59 shows the variation in output fall time with load capacitance. *Figure 60* shows the output fall time vs. chip voltage at various temperatures. Notice that there is almost no variation in output fall time due to changes in device temperature. *Figure 61* shows the output fall time vs. termination voltage.

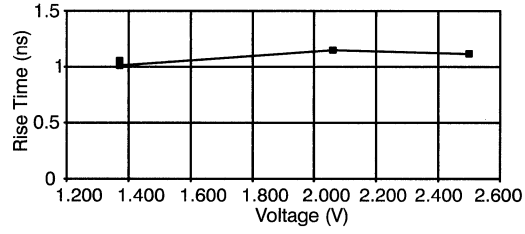


Figure 58. Rise Time vs. Termination Voltage

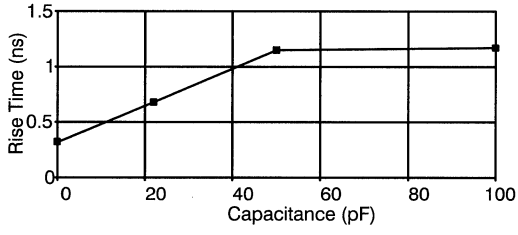


Figure 56. Rise Time vs. Capacitance

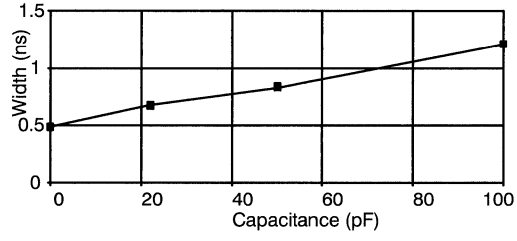


Figure 59. Fall Time vs. Capacitance

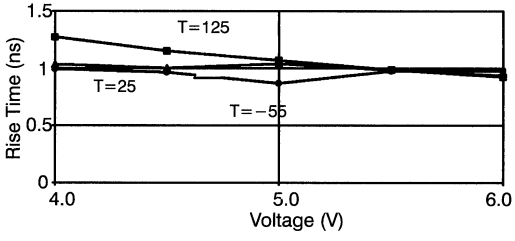


Figure 57. Rise Time vs. Voltage over Temperature

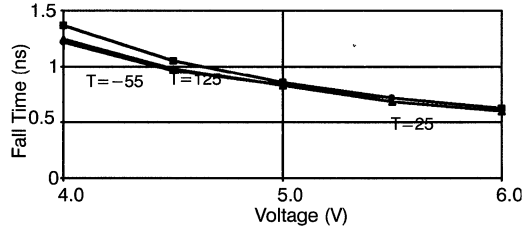


Figure 60. Fall Time vs. Voltage over Temperature

Output Duty Cycle Variation

Output duty cycle variation is the difference in the output pulse width from the ideal 50%. This parameter is measured at the 1.5V level. Characterization of this parameter was performed by measuring the output pulse width high. Measurements were taken at two different REF input cycle times (Tref): 50 ns and 12 ns.

Figure 62 and *Figure 63* show the variation in output duty cycle due to variations in output capacitance. These graphs indicate that if the outputs are loaded according to the datasheet specification, the output duty cycle will be very near 50%.

Figure 64 and *Figure 65* show the output pulse width high vs. device voltage over temperature. These graphs indicate that the propagation delay difference between the rising and falling edge of the outputs varies according to frequency. For operation within the ±10% V_{CC} and the commercial temperature range, the output duty cycle specification is ±500 ps.

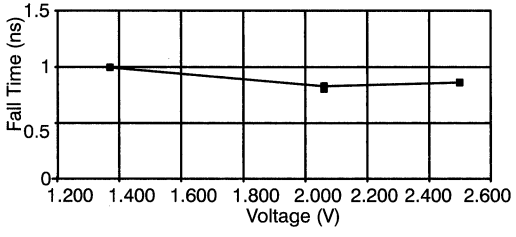


Figure 61. Fall Time vs. Termination Voltage

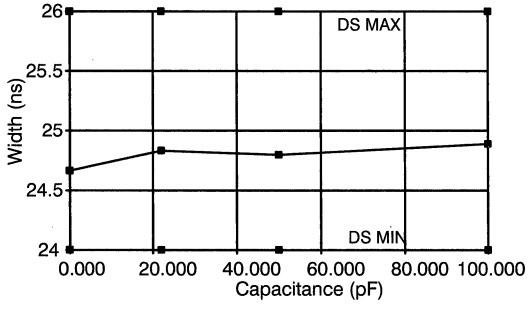


Figure 62. Pulse Width High vs. Capacitance t_{REF} = 50ns

Figure 66 and *Figure 67* show the output pulse width high vs. termination voltage. Both of these graphs indicate that with normal variance in termination

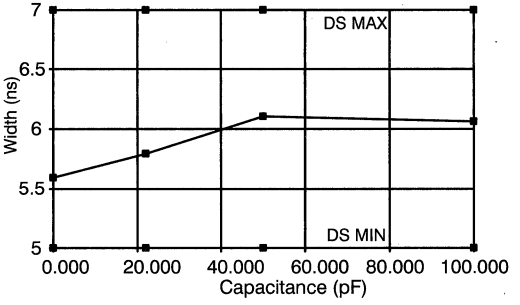


Figure 63. Pulse Width High vs. Capacitance t_{REF} = 12ns

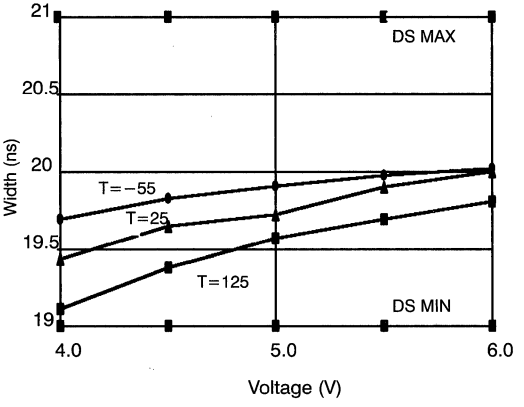


Figure 64. Pulse Width High vs. Voltage over Temperature

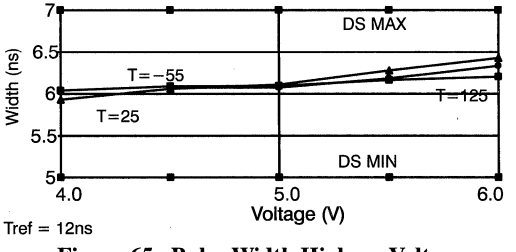


Figure 65. Pulse Width High vs. Voltage over Temperature

voltage the output duty cycle specification remains within the ±500 ps range.

Conclusion

RoboClock provides system designers with a multifunctional resource that solves most clock distribution problems. The third-generation PLL architecture based on a distributed phase clock oscillator allows phase shifting, division, multiplication and inversion of outputs with over 26,000 possible output configurations. These features combine to offer compensation for trace-length differences, elimination of set-up and hold time mismatches, multiplication of lower-frequency system clocks, division of system clocks for lower-performance system components, and phase inversion for multiphase clocking. All of these benefits are combined with extremely low skew outputs, low device propagation delay, and high-frequency operation to provide the most full-featured device available.

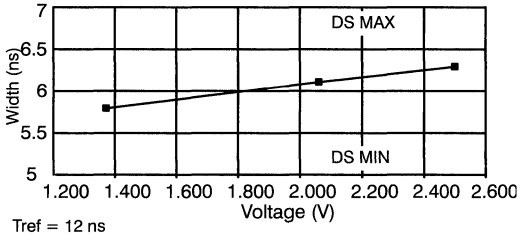


Figure 66. Pulse Width High vs. Termination Voltage

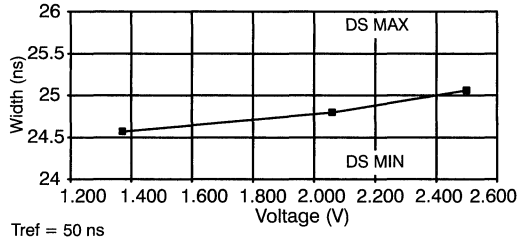


Figure 67. Pulse Width High vs. Termination Voltage

References

1. American Technical Ceramics, *The RF Capacitor Handbook,* Revision B, December 1991.
2. Blood Jr., William R., *MECL System Design Handbook*, Fourth Edition, 1988.
3. BTL Working Group, "A Guide to Backplane Electrical Performance Measurements," P1194.0/D2, July 1989.
4. Motorola, Inc., *Application Note AN1051* "Transmission Line Effects in PCB Applications," 1990.
5. Wescon Technical Conference, "Solving Clock-Distribution Problems in High-Speed Systems," Session 5, Nov. 1991.

Innovative Designs with the CY7B991/2/10/20 (RoboClock) Programmable Skew Clock Buffer

Overview

This article discusses various applications of the Cypress Phase Locked Loop-based, skew-defeating clock buffers known as RoboClock. It is assumed that the reader has a working understanding of RoboClock. If not, "Related Articles" shown below are recommended. Unlike traditional clock buffers, RoboClock enjoys the advantages of an internal, multi-tapped PLL, which offers designers two principal advantages: zero propagation delay and configurable phase control, relative to the reference clock.

Zero propagation delay is achieved through the presence of the internal PLL. Because of the properties of PLLs, RoboClock is able to synchronize itself to an incoming reference clock, allowing the buffered outputs to be coincident with the reference input, effectively acting as a zero propagation delay clock buffer. Conventional clock buffer solutions, even those that offer low skew, still have a finite amount of propagation delay. An application example later in this article demonstrates the constraints these delays force designers to operate under, and how RoboClock allows the designer to overcome these constraints.

Configurable phase control of distributed clocks allows the designer to overcome the debilitating effects of clock skew. With today's increasing clock rates, the amount of time required for a clock signal to travel across a circuit board becomes a significant portion of the clock period. This clock skew can result in timing specification violations. RoboClock offers designers the ability to manipulate the phase of the distributed clocks, and thereby compensate for clock skew.

Related Documents

For a more complete description of RoboClock as well as its internal PLL, the reader is encouraged to consult the following documents for additional information:

"CY7B9910/CY7B9920 Low Skew Clock Buffer" data sheet.

"Innovative RoboClock Application" published in the Cypress Semiconductor *Applications Handbook*.

"CY7B991/CYB992 (RoboClock) Test Mode" published in the Cypress Semiconductor *Applications Handbook*.

"Everything You Need to Know About CY7B991 and CY7B992 (RoboClock) But Were Afraid to Ask" published in the Cypress Semiconductor *Applications Handbook*.

"CY7B991/CY7B992 Programmable Skew Clock Buffer" data sheet, published in the Cypress Semiconductor *Data Book*.

Using RoboClock to Overcome a Timing Violation

This design example typifies how RoboClock can be used to solve timing margin problems. In this case, the problem is a register set-up time violation. Represented is an actual design implemented by a major telecommunications manufacturer.

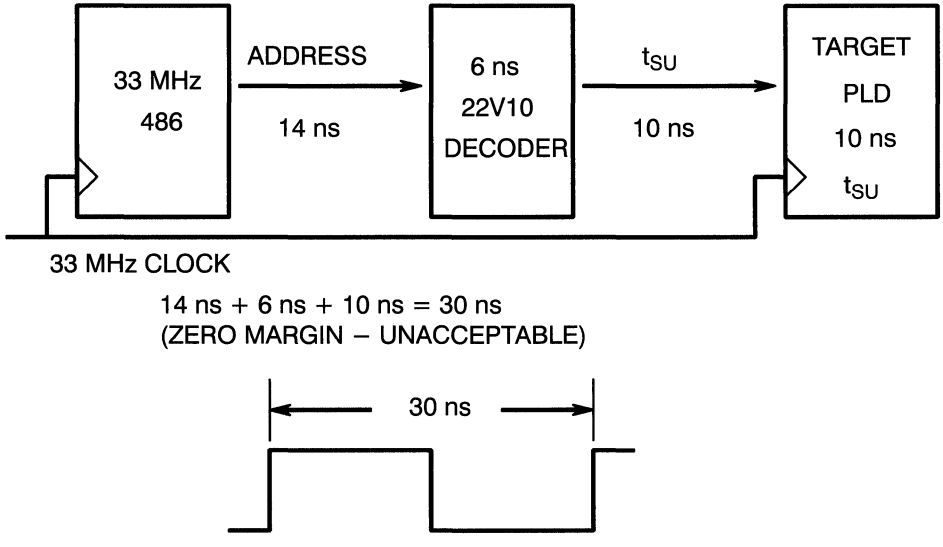


Figure 1. Timing Violation

In this application a 33-MHz 486 microprocessor's address has a critical path to the registered target PLD through a 6-ns 22V10. As shown in *Figure 1*, the address is guaranteed valid out of the 486 processor 14 ns after the initial rising clock edge, after which the address is decoded by a 22V10 (requiring an additional 6 ns) before needing to meet a required 10-ns register set-up time of the CPLD. Examination of the timing constraints shows that no margin is present—a situation deemed unacceptable by the designer.

The designer chose the RoboClock implementation shown in *Figure 2* in order to solve this timing margin

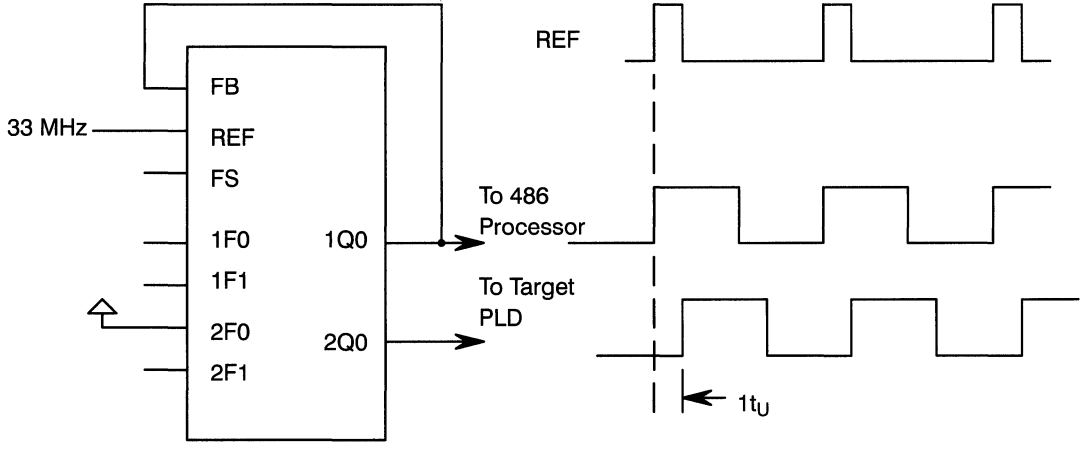


Figure 2. Timing Violation Solution

problem. Essentially, the designer used RoboClock to "move the clock", adjusting the phase of the 33-MHz clock input to the target PLD. As is shown, the 1F0 and 1F1 control inputs are allowed to "float" (the MID logic state), the resultant 1Q0 output is a buffered 33-MHz clock phase-aligned with the 33-MHz reference input. The 2F0 input is tied HIGH and the 2F1 input is allowed to float, configuring the 2Q0 output to be delayed by one timing unit (t_U), thus yielding 1.2 ns of margin to the circuit. The delayed 2Q0 output is then routed to the clock input of the target PLD, the 1Q0 clock is distributed to the clock input of the 486 processor.

In summary, the buffered clock signal, coincident with the reference clock, is distributed to the 486 processor, and a delayed clock is distributed to the target PLD, allowing the PLD's set-up time to be satisfied. An alternate solution would have been to distribute an advanced clock to the 486 processor, and distributing a nominal clock to the target PLD. Either solution may be implemented with RoboClock.

Note that the reference input as shown in *Figure 2,* need not be a "50-50" duty cycle signal in order for RoboClock to output a guaranteed 50–50 duty cycle clock. Duty cycle requirements are increasingly important for contemporary processors.

RoboClock as a Zero Propagation Delay Buffer

Clock speeds of 33 MHz and higher have become the norm in the modern design environment. Increasing clock rates mean decreasing clock periods, resulting in less "processing time" between rising clock edges. Conventional clock buffering methodology is no longer adequate for these applications because the propagation delay through a traditional "244" buffer is a significant portion of the clock period.

Even "high-performance, low skew" buffers suffer from some finite amount of propagation delay. This inherent delay translates into processing time lost to the designer. Fortunately, designers can benefit from PLL-based clock buffers, which are able to offer zero propagation delay. The following example is based on an actual implementation at a major manufacturer of ATM Network Adaptor cards.

Figure 3 shows RoboClock configured to operate in its most elegant mode, that of a zero propagation delay clock buffer. It is contrasted against a conventional clock buffering solution, such as a "244" buffer, which has inherent, performance degrading finite propagation delay. The specific version of RoboClock shown is the CY7B9910, which is a functional subset of the original, more fully featured, CY7B991. The 7B9910 or "Robo Jr." features an identical PLL core as its parent, and thus the identical excellent low-skew characteristics between buffered outputs. Robo Jr. was designed to be exclusively a low-skew, zero propagation delay clock buffer. It therefore lacks the previously described clock phase configurability that enables RoboClock to negate clock skew.

RoboClock as a Universal Clock Multiplier

Figure 4 shows RoboClock's ability to synthesize, with the addition of an external counter, any integer multiple of the reference frequency, up to a limit of 80 MHz. RoboClock has the built-in ability to multiply the reference clock by two and by four. Use of an external counter greatly expands this ability. This example is based on an actual design implemented by a major telecommunications manufacturer.

This frequency synthesis/multiplication is accomplished, as shown in this example, by feeding the terminal count output of a divide-by-three counter into the Feedback (FB) input of RoboClock. The relative phase difference present at the REF and FB inputs causes the internal PLL to adjust its output until these two inputs are phase aligned. This results in the outputs tripling in speed, from 20 MHz to 60 MHz. Additionally, the 60-MHz 4Q0 output is shown to be inverted. This was accomplished by setting the 4F0 and 4F1 control inputs as "High, High", the inverting configuration. Alternatively, the 60-MHz 4Q0 output could have been configured in a phase-adjusted mode—pushed forward several timing units or likewise pulled back. Thus the use of the external counter to accomplish clock multiplica-

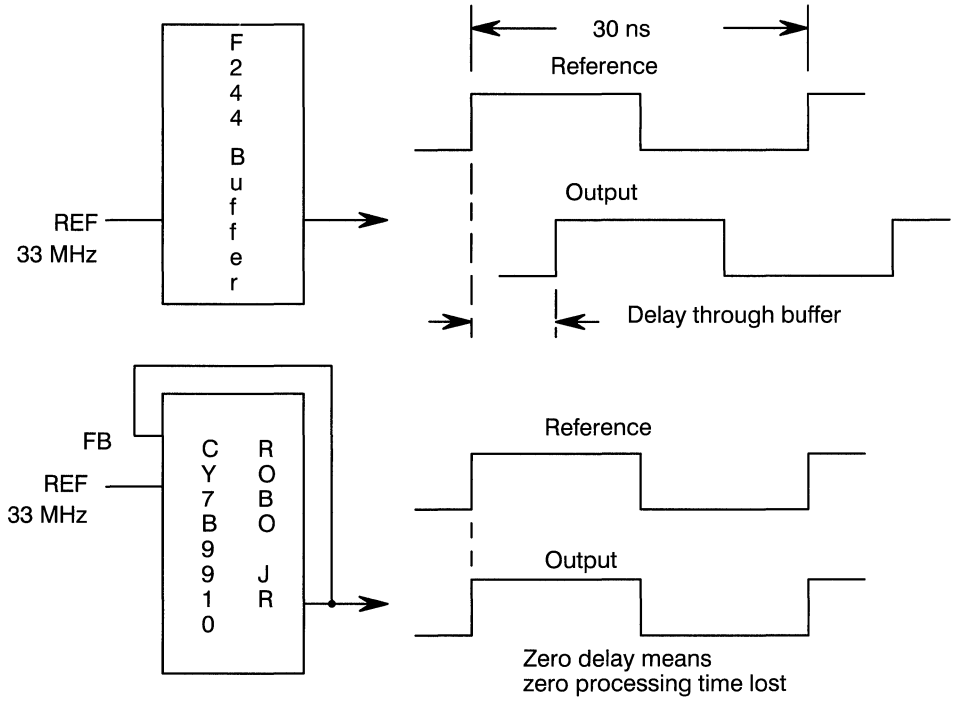


Figure 3. Zero Propagation Delay Buffer

tion in no way inhibits the normal features of RoboClock.

As shown in *Figure 4*, the multiplied outputs are slightly skewed from the Reference input. This phase difference results from the inherent "clock to output" delay of the external counter. Any counter will have some finite delay, which will manifest itself in this manner.

Should this skew be considered detrimental to the application, it can easily be eliminated on the 2Q, 3Q and 4Q outputs by adjusting the respective phases of the outputs. By the nature of the application, this skew will always be present between the 1Q output and the REF/FB input.

Gated RoboClock

From time to time, design requirements necessitate the "gating" of clock signals. Special care must be taken whenever this is done in order to prevent intermittent glitching of the distributed clock, the result of enabling and disabling. Clock glitches may result in minimum pulse width violations upon registers present on the circuit board. The application shown is based upon similar implementations in both an ESCON (Enterprise System CONnection) product as well as a high resolution graphics subsystem employed in a virtual reality product.

Should the output of RoboClock need to be gated, or three-stated, a viable method is shown in *Figure 5*. The CYBUS3384 is essentially a zero propagation delay (125 ps worst case) three-state buffer. The upper half of the CYBUS3384 is continually enabled, allowing a continuous wave form into the FB input of RoboClock. The output of the lower half of the CYBUS3384 is the gated clock, present when the CYBUS3384's $\overline{EN}$ signal is asserted (active LOW) and three-stated otherwise. The output enabling scheme shown, which has a register sampling the $\overline{EN}$ input upon the falling edge of the 1Q

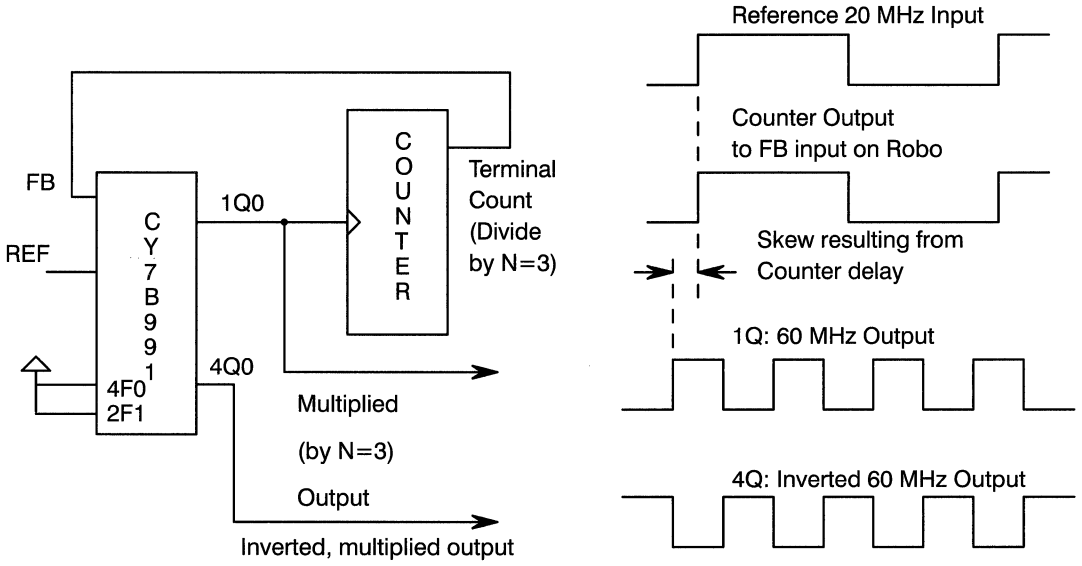


Figure 4. Universal Clock Multiplier

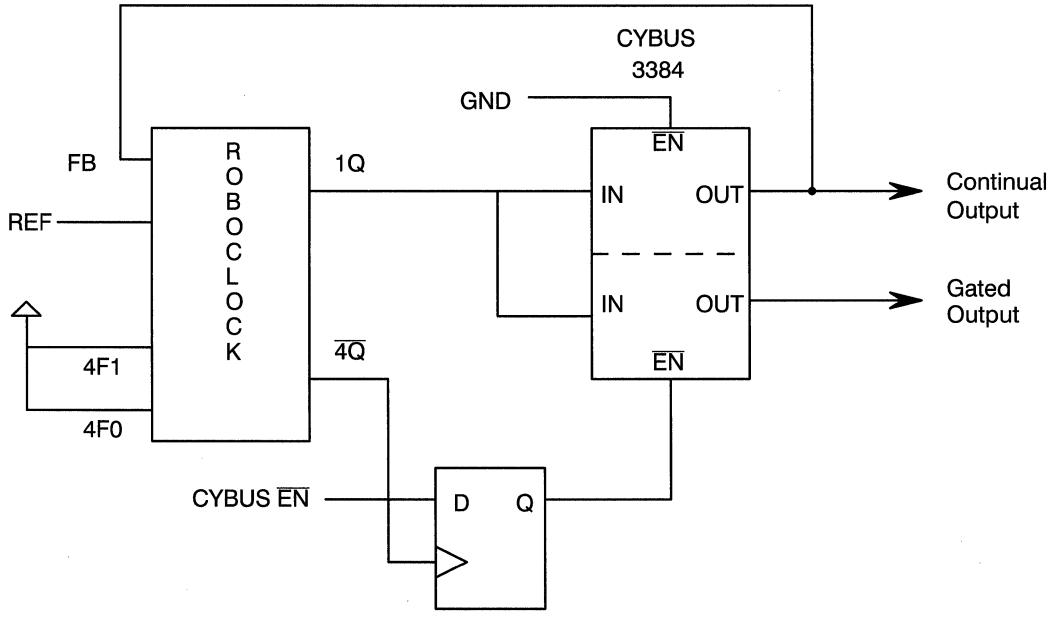


Figure 5. Gated RoboClock

clock (the 4Q output has been inverted, and thus a rising edge on 4Q corresponds to a falling edge on 1Q) guarantees that the gated clock output will never "glitch."

Continually Phase Adjusted Clock Source

As a result of the flexible configuration options within RoboClock, it is possible to achieve virtually 360 degrees of phase adjustment, allowing "placement" of clock edges throughout the period of a reference wave form. This functionality has been implemented in a telecommunications network analysis system used by Regional Bell Operating Companies (RBOCs) and is depicted in *Figures 2* and *7*.

In this application, a 33-MHz reference signal is dynamically phase adjusted by writing different values into a CMOS output level register, which in turn feeds the 3F and 4F RoboClock inputs. Note that the register used must have CMOS outputs, i.e., they must go "rail-to-rail" in order to satisfy the input level requirements of RoboClock. The register must also be capable of being three-stated, so that it can put the 3F and 4F RoboClock inputs into the "MID" state.

The application shown offers the designer the ability to subdivide the 30 ns period into thirteen slices, 2.4 ns apart. Each configuration of the 3F and 4F inputs corresponds to a different phase adjustment of the buffered clock, relative to the reference clock.

Conclusion

Today's high-performance design environments require the design engineer to work with and distribute high-speed clocks. By their nature these high-frequency clocks make the designer's task difficult. When these clocks have to be distributed over even relatively short distances, the effects of clock skew can make the designer's job impossible. The RoboClock family offers the design engineer opportunities to overcome a myriad of design challenges. Its ability to manipulate clock waveforms, and to counteract the effects of clock skew make it an integral part of the contemporary designer's repertoire.

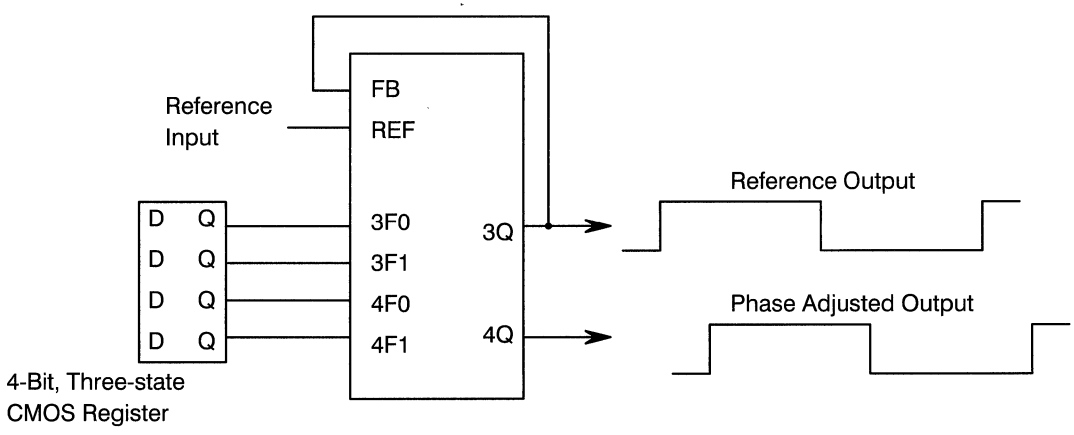


Figure 6. Continual Phase Adjustment

Control Inputs			
3F0	3F1	4F0	4F1
MID	MID	HIGH	MID
MID	MID	LOW	HIGH
MID	MID	MID	HIGH
MID	LOW	HIGH	MID
MID	LOW	LOW	HIGH
MID	LOW	MID	HIGH
MID	MID	LOW	MID
MID	MID	HIGH	LOW
MID	MID	MID	LOW
MID	HIGH	LOW	MID
MID	HIGH	HIGH	LOW
MID	HIGH	MID	LOW

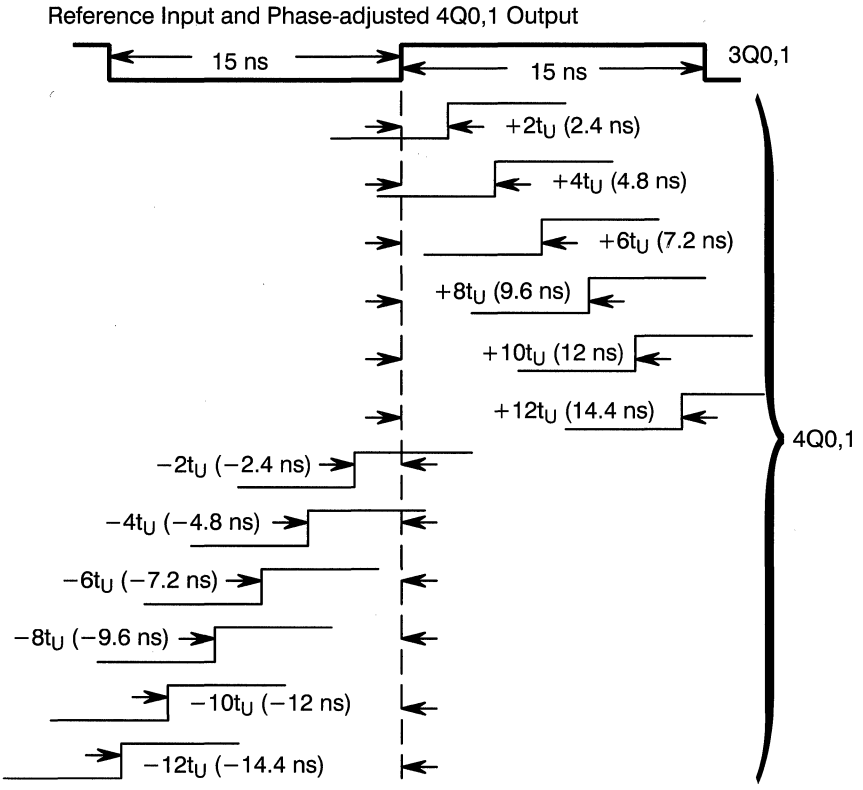


Figure 7. Continual Phase Adjustment

Generation of Synchronized Processor Clocks Using the CY7B991 or CY7B992

Introduction

Many modern systems use multiple processors operating simultaneously in order to increase performance and improve throughput. Timing analyses and interprocessor communications are significantly simplified if the clocks to the processors occur at exactly the same time. This application note explains the problem and presents a technique for generating synchronous clocks to two Intel 80960CA processors using the Cypress CY7B991 Programmable Skew Clock Buffer (PSCB), also known as RoboClock. The technique is then extended to "n" processors.

Design Requirements

The processors require 33-MHz clocks and are operated in the x1 mode (i.e., CLKMODE = HIGH). In this mode the output clock, PCLK1, PCLK2, are also 33-MHz and can be phase shifted plus or minus two nanoseconds from the input clock, CLKIN. This is due to the internal (2X) Phase-Locked Loop (PLL) in the processor. The minimum CLKIN LOW duration is 10 ns and the minimum CLKIN HIGH duration is also 10 ns. In addition, the maximum cycle-to-cycle CLKIN period variation is plus or minus 0.1%. Another requirement is that the RESET input to the processor be held LOW for at least 10,000 CLKIN cycles after V_{CC} and CLKIN have stabilized (are within their specifications) before it is allowed to go from LOW to HIGH.

Clock Interconnections

Figure 1 illustrates the interconnections required for clock synchronization. The connections are the same if the CLKIN frequency is 66 MHz. However, the CLKMODE input must be tied to ground. The PCLK1 output is then 33 MHz.

Theory of Operation

During power turn-on, the $\overline{\text{RESET}}$ input of each processor must be held LOW by external logic (not shown). Typical power supply turn-on times are in the 50 ms to 500 ms range. After the power supply and 33-MHz oscillator have stabilized, it takes 10,000 cycles of the 33-MHz CLKIN input to processor 1 before its PCLK1 output is within its specification. This is 300 microseconds.

During this time, the TEST input to the CY7B991 must be held HIGH. When this is done, the internal Phase-Locked Loop of the CY7B991 is disabled and the signal at the REF (reference) input is passed through to the 1Q0 output, and then to the CLKIN input of processor 2. Again, 300 microseconds must pass before the output PCLK1 of processor 2 is stable. Both processors are now running at 33 MHz, under control of the oscillator. The PCLK1 clocks of the processors, however, are not synchronized.

Synchronization of the Processor Clocks

The next step is to cause the TEST input of the CY7B991 to go from HIGH to LOW. This causes the Phase-Locked Loop within the CY7B991 to ad-

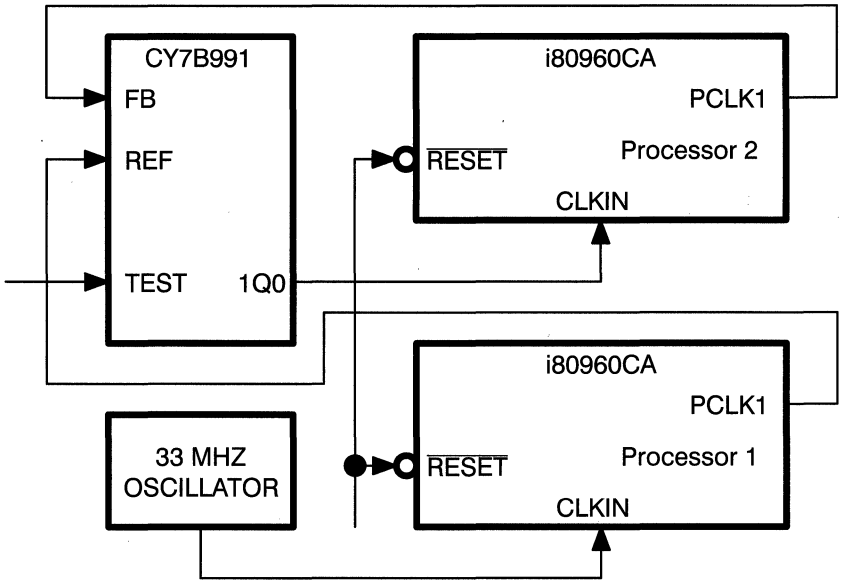


Figure 1. Clock Connections for Synchronization

just the phase and frequency of the 1Q0 output, which is driving the CLKIN input of processor 2, such that the rising edges of the signals on its FB and REF inputs are aligned. Because the PCLK1 output of processor 2 is a function of its CLKIN input, when the CY7B991 adjusts its 1Q0 output, the PCLK1 output follows. The result is that the PCLK1 outputs and, therefore, the CLKIN inputs of the two processors are synchronized. What this means is, that for all practical purposes, there is "zero delay" between the rising edge of the signal on the REF input and the signal on the FB input. However, what is more important is that this alignment is adaptive and dynamic because it occurs on a cycle-by-cycle basis, and, therefore, is not influenced by variations in power supply voltage or temperature. After the processor clocks are synchronized, the $\overline{\text{RESET}}$ lines to the processors can transition from LOW to HIGH.

CLKIN Cycle-to-Cycle Variation

The next step is to calculate the maximum cycle-to-cycle variation of the CLKIN input to processor 2 and make sure that it is within the 0.1% specification on the 80960CA data sheet. At 33 MHz the clock period is 30 ns, so $0.001 \times 30 \times 10^{-9} = 30$ picoseconds per cycle.

The Phase-Locked Loop of the CY7B991 requires approximately 50 microseconds to lock. This corresponds to 50 microseconds divided by 30 ns per cycle, or 1,667 clock cycles. The worst-case condition is that the two processor clocks are 180 degrees out of phase when the signal at the TEST input of the CY7B991 transitions from HIGH to LOW. One-half a cycle of a 30 ns period clock is 15 ns. Fifteen nanoseconds divided by 1,667 cycles is 9 picoseconds per cycle. This is much less than the plus or minus 30 picoseconds (60 picoseconds total) specified on the 80960CA data sheet

Synchronization of Many Processors to a Single Clock

Figure 2 illustrates how three processors can be synchronized. The first runs off of the oscillator and the other two are synchronized to the first by using two CY7B991s. The PCLK1 output of processor 1 is the

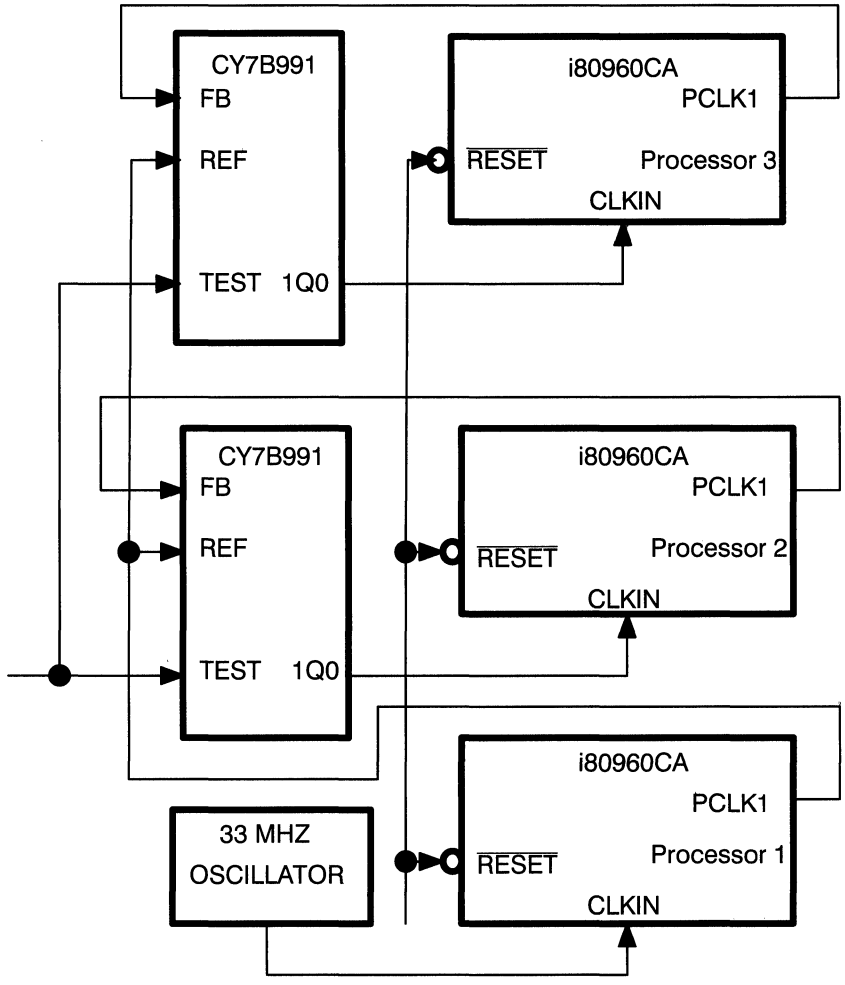


Figure 2. Clock Connections for Synchronization of Three Processors

reference for the two CY7B991s. Each controls the clock to a processor. Thus n−1 CY7B991s are required to synchronize n processors.

The advantage of using separate RoboClocks for processor 2 and processor 3 is that, because of the analog nature of the internal RoboClock PLL, the PCLK1 output of each is independently and dynamically adjusted, on a cycle-by-cycle basis, with the PCLK1 output of processor 1. This is accomplished by applying the PCLK1 output of processor 1 to the REF input of the two RoboClocks and tying the PCLK1 outputs of processors 2 and 3 to the FB inputs of two separate RoboClocks.

One CY7B991 can be used to control many processors if they do not have on-chip Phase-Locked Loops. Or, the system designer may choose to not use the processor clock output.

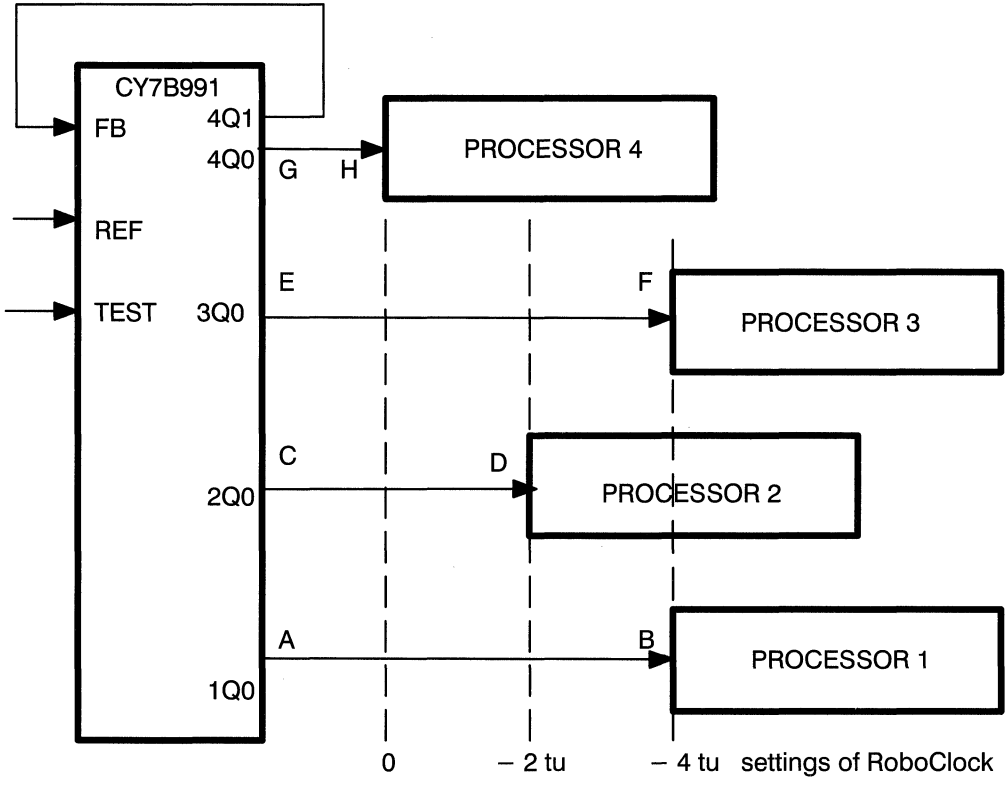


Figure 3. One RoboClock Driving Multiple Processors

Driving Multiple Processors From One RoboClock

Figure 3 illustrates one CY7B991 driving multiple processors that are located at different distances.

Advantages

The advantages of the configuration illustrated in *Figure 3* are (1), that one CY7B991 can drive up to seven processors using seven of the eight RoboClock outputs and (2), that the select inputs can be used to adjust the timing of the Q outputs to compensate for variations in trace length, so that the clocks to the processors arrive at exactly the same time.

For example, the propagation delay of trace G H is two timing units, that of trace C D is four timing units, and those of traces E F and A B are six timing units. It is required that the clocks to all of the processors arrive to each at the same time.

The first step is to select a "zero" point as a timing reference. This is the clock at processor 4, which is point H. However, in real life, the propagation delay of trace G H is two timing units. What RoboClock does is precisely align the rising edge of the signal at its FB input with the rising edge of the signal at its REF input. The length of the fed back output trace (4Q1 to FB) should be as short as possible. It not only simplifies the timing analysis, but also reduces the noise introduced into the PLL.

Limitations

There is no feedback from the clock outputs of the processors, so they cannot be individually, dynamically aligned with the REF clock, as is done in *Figure 2.* A second limitation is that the eight outputs of the CY7B991 are grouped in four pairs and are adjustable only as pairs. This means that a maximum of three (pairs of) processors can be aligned independently if each is a different distance away from RoboClock. By matching the trace length within pairs (i.e., 1Q0, 1Q1) up to seven processors can be driven, each with its own, dedicated output.

Determination of Delay Settings

For purposes of explanation, the "zero" is chosen at point H, which is the closest physical point to RoboClock. Point F is four timing units farther away than point H, so its signal must precede (timewise) that at point H, so that the signals at points F and H occur at the same time. Therefore, the select input controlling the 3Q outputs is set at −4 timing units. In a similar manner, the select input controlling the 2Q outputs is set at −2 timing units and that controlling the 1Q outputs is set at −4 timing units. When this is done, the signals at points H, F, D, and B occur at the same time, which is the zero point shown. The timing is shown in in *Figure 4* below.

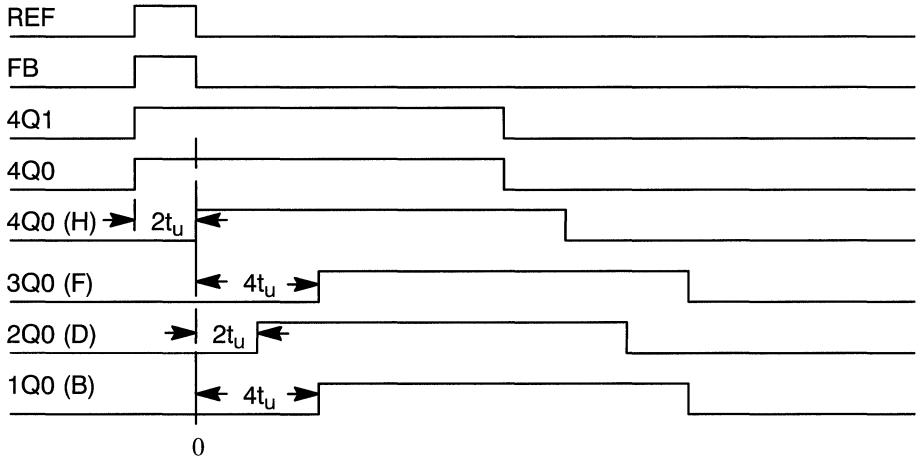


Figure 4. Timing Diagram for Figure 3 (before select control settings)

Innovative RoboClock Application

Introduction

This application note presents a unique application of RoboClock, using its complex and precise waveform generation capability to implement PWM (Pulse Width Modulation) to enhance color images and increase the resolution of laser printers. The first section of this application note provides a brief description of RoboClock and presents three methods that the user can employ to configure it. Second, a brief background on image and resolution enhancement is presented. Finally, the required waveform to implement the image enhancement and the configuration of RoboClock is presented.

Overview of RoboClock

The CY7B991 and CY7B992, commonly known as RoboClock, are programmable skew clock buffers capable of generating thousands of various clocking combinations. As shown in *Figure 1*, the eight high drive outputs are arranged in four pairs, which can be configured by three-level inputs (HIGH, LOW, and MID logic level). The internal PLL is fully self-contained and does not require any external components to operate. The PLL buffer stages are differential, greatly enhancing the robustness of the PLL operation in terms of jitter over voltage, and temperature variations.

Basically, the PLL aligns the output clock in both phase and frequency with the reference clock. The simplest mode of operation is the low-skew output mode. In this mode the outputs are virtually skewless. The maximum skew is only a few hundred picoseconds. Please refer to the CY7B991/992 datasheet for various skew specifications. The second mode is the programmable skew mode. The outputs of RoboClock can be skewed (advanced and delayed) by increments of one time unit (t_U), which is 0.7 to 1.5 ns, determined by the operating frequency and range of the PLL.

$$t_U = 1/(F_{nom} * N) \qquad \text{Eq. 1}$$

As shown in *Table 1*, the frequency range of the PLL is determined by the three-level FS input. For each frequency range, there is a corresponding integer "N" that can be used in *Equation 1* to calculate t_U. Up to $\pm 12 t_U$ skew can be achieved between the outputs of RoboClock (positive t_U represents delaying

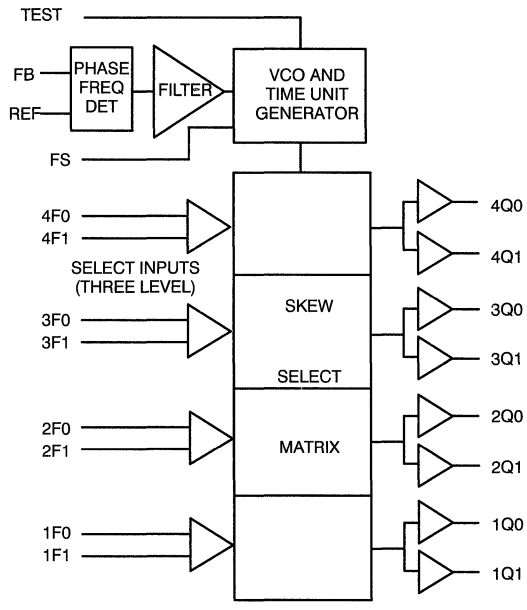


Figure 1. Logic Block Diagram

the output with respect to REF and negative t_U represents advancing the output with respect to REF).

The third mode of operation is the Multi-function mode. In this mode the outputs may be multiplied by 2 or 4, divided by 2 or 4, or inverted. Most importantly, the skew features can be combined with multiply, divide, and invert functions. This results in, literally, over 26,000 timing configurations. For more detailed information on the operation of the RoboClock, please refer to the following application notes "Using the CY7B991 with the 50-MHz 486 Cache Module and the 40-MHz R3000" and "Everything You Need to Know About CY7B991/CY7B992 (RoboClock) But Were Afraid to Ask." This application note is meant to complement the topics discussed in above mentioned application notes.

Table 1. Frequency Range Select and t_U Calculation

FS	f_{NOM} (MHz) Min.	f_{NOM} (MHz) Max.	$t_U = \frac{1}{f_{NOM} \times N}$ where N =	Approximate Frequency (MHz) At Which t_U = 1.0 ns
LOW	15	30	44	22.7
MID	25	50	26	38.5
HIGH	40	80	16	62.5

Usually, one of the outputs of RoboClock is used as the Feedback input. If the desired waveform is not directly generated by RoboClock, an imaginative user may run an output of RoboClock through a logic block, then send it back to the FB input of the RoboClock. Through this scheme, unlimited additional functions may be implemented by RoboClock. Note that in this case, all the other outputs of the RoboClock will be shifted by a period equal to the delay through the external logic block, because the PLL will align the FB input with the REF input, both in phase and frequency.

Cascading two RoboClocks in series will also dramatically increase the output possibilities. In this case, one of the outputs of the first stage will serve as the REF input for the second stage. Multiple feedback configurations are possible, which can result in an innovative set of outputs.

RoboClock Configuration Methodologies

Using One Small Table

The entire set of programmable skew configurations is summarized in a single small table shown in *Table 2*. Every possible combination can be driven from this small table. For example, if $+2t_U$ is required from 3Qx (3Q0 or 3Q1) outputs, based on *Table 2*, the corresponding 3Fx inputs should be set as 3F1= MID and 3F0=HIGH. Any one of 1Qx, 2Qx, or 4Qx outputs may be used as FB input, by leaving its corresponding 1Fx, 2Fx, or 4Fx inputs floating (i.e., 1F1= MID , 1F0= MID). Note that *Table 2* represents only the cases where the feedback is an output with no skew, divide, or invert function. Basically, a $0t_U$ output is used for FB input.

Table 2. Output Adjustment Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	1F0, 2F0, 3F0, 4F0	1Q0, 1Q1, 2Q0, 2Q1	3Q0, 3Q1	4Q0, 4Q1
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

Now, to generate additional output functions, if the feedback output is programmed to skew, divide or invert, then output functions of other outputs may not be directly read from *Table 2*. In this case, to figure out the final output function observed on the output, simply subtract whatever the feedback term is programmed to, from the output function programmed on the corresponding output. Therefore, by using only *Table 2* and the following simple algorithm, every single combination of RoboClock can be figured out.

Final Output Function = Output Function − FB Function

If there is any ambiguity, the following example should clarify the use of this method. Let's say $+7t_U$

of delay is required. Obviously, $+7t_U$ is not a choice, available in *Table 2*. However, any two functions from two different outputs of *Table 2* may be combined to achieve a desired function. For this example, there are several solutions, and only one of them will be presented. One way to achieve $+7t_U$ is to subtract $-3t_U$ from $+4t_U$.

$$+7t_U = +4t_U - (-3t_U)$$

Therefore, if 1Qx output is programmed to have $-3t_U$ of skew (1F1=LOW, 1F0=MID), and used as the FB input, and if the 3Qx is programmed to have $+4t_U$ of skew (3F1=HIGH, 3F0=LOW), the final output function observed on 3Qx will be $+7t_U$.

One exception to this simple rule is that if a divided output is used as the FB input, then the other outputs will be multiplied by the same factor (2 or 4). The reason for this is that the PLL will force the FB to align with the REF both in phase and frequency. Therefore, if the FB term is programmed to divide by 2, the PLL will speed up twice to force the FB term to align with the REF frequency. As an example, if advance by $6t_U$ and multiply by 4 function is required ($-6t_U$ and f*4), then

Final Function = $-6t_U$ – *(divide by 4)* => *($-6t_U$ and f*4)*

The solution for this example is to program 3Qx to divide by 4 (3F1=HIGH, 3F0=HIGH) and use it as FB, and program 4Qx to have $-6t_U$ of skew (4F1=LOW, 4F0=MID). The final function observed on 4Qx will be REF frequency multiplied by 4 and advanced by $6t_U$ ($-6t_U$ and f*4).

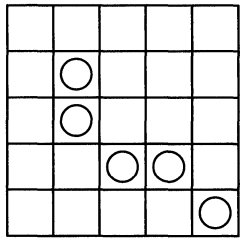
By this method, one can easily determine if a desired function can be implemented by RoboClock or not. RoboClock can generate a waveform composed of any two functions from two different outputs of *Table 2*.

Using Three Tables for Multiple Outputs

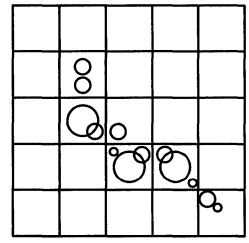
If multiple outputs with various functions are required, using the previous method could be a little cumbersome. All the possible combinations of RoboClock outputs are in three tables, illustrated in *Tables 3* through *5*. Each table represents all the possible output combinations with a given output connected to FB input. For example, once 3Qx output is used as FB input, then all the possible output combinations could be found in *Table 4*. These three tables are extremely valuable tools in determining what FB term to use and how to configure the RoboClock, when multiple outputs with various functions are required.

Once the required multiple functions are determined in terms of t_U, an effort should be made to locate one row in one of the three tables that contains the required functions. For example, if one of the desired functions is divide by 2 and delay $4t_U$ ($+4t_U$ and f/2), then by observation, that choice can be located in row 1 of *Table 3*, row 3 of *Table 4*, and row 3 of *Table 5*. Now, the one to be selected as a solution would depend on what the other required functions are, because once an output, which is programmed to perform a certain function, is selected as FB input, all the outputs of a RoboClock are limited to a single row found in *Tables 3* through *5*. If in the previous example, the second required function happens to be invert and skew by $4t_U$ ($+4t_U$ and INV), then the only solution is row 1 of *Table 3*. In this case 1Qx could be used as the FB input with its inputs hardwired to GND (1F1=LOW, 1F0=LOW), and 3F1=HIGH, 3F0=HIGH to generate ($+4t_U$ and f/2) function on 3Qx outputs, and set 4F1=HIGH, 4F0=HIGH to generate (+4tu and INV) function on 4Qx outputs. In this configuration, the 2Qx outputs could be programmed to have any one of $0t_U$ through $+8t_U$ skew. For example, if $+7t_U$ is another required output, then 2F1=High, 2F0=Mid will generate $+7t_U$ skew on 2Qx. Note that even though the 1Qx outputs are programmed to have $-4t_U$ skew, they are forced by PLL to align with the REF frequency, therefore 1Qx output could be used as a $0t_U$ output.

The Table method is recommended for multiple outputs with various function requirements. If the exact required outputs cannot all be found in one row, then the designer can use the three tables to understand the design choices that are available within the three tables. Based on the design requirements, the user can make a judgement on what outputs must exactly meet the required specs, and what outputs may be slightly compromised. If the required outputs are not found in one row of the three tables,



a. Standard unmodulated



b. Modulated laser beam with shorter "on" periods and variable-size dots.

Figure 2. Laser Images

and no compromise can be made on the requirements, then two or more RoboClocks may be used to meet the specific required outputs.

Using RoboClock in Resolution Enhancement of a Laser Printer

Background

Laser printers are no different from any other electrical systems, in that the higher the resolution or the accuracy of the system, the higher the complexity and cost of the system. It has been and will always be the goal of system and design engineers to achieve the highest performance and resolution at the lowest cost.

In the case of a laser printer, to achieve higher resolution than the nominal low-end 300 DPI (dot per inch), the throughput of the processor, size of the memory, and the glue logic should increase accordingly. In many cases, the additional hardware cost does not justify the enhancement in the resolution. A few years ago, a new technique called Resolution Enhancement Technology (RET) was developed by Hewlett Packard. The main advantage of this technique versus conventional resolution enhancement techniques is that the resolution enhancement is gained with hardly any increase in the throughput of the processor or memory size. Therefore, this approach is a very economical way of gaining enhanced resolution.

For obvious reasons, the entire laser printer industry is using some form of this technique. Various flavors of the same technique are being applied to different image-enhancement machines. The halftones or gray scales are common in most laser printers. The underlying technique is fairly simple. This is accomplished by modulating the laser beam, as opposed to the conventional "on" or "off" state of the laser beam, for the entire cycle time. The laser beam could be turned "on" or "off" 25%, 50%, 75% or 100% of the period. By this, large and small dots can be produced on a given image, therefore gaining much higher "perceived" resolution compared with images constructed by only one size dot. The varying size dots produce much smoother text files and generate much sharper images through shades of gray. Refer to *Figure 2a* and *2b*. The same idea is used in color image enhancement where much smoother and more pleasant images may be produced in a given color image by dilating black dots and shrinking the red. This filtering or color enhancement feature can be used to produce various special effects, or simply be employed to create more appealing color images. Please refer to *Figure 3a* and *3b*. To further clarify this technique, the true resolution, technically, still remains the same, but the images are "perceived" to be higher resolution. It does not matter how the "better image" was

a. Black dot dilated

b. Red dot shrunk

Figure 3. Color Enhancements

created, as long as it looks good and the cost of the hardware is affordable.

Design Implementation

RoboClock is used to generate precise complex waveforms needed for laser beam modulation. The particular laser system discussed in this application note requires eight levels of modulation, which consists of 100% on, 75% on, 50% on, 25% on, 100% off, 75% off, 50% off, and 25% off. The eight waveforms are shown in *Figure 4*.

Note that all waveforms are synchronized to the rising edge of the system clock.

Analyzing the entire circuitry of the laser printer is beyond the scope of this application note, and only the waveform modulation section is discussed. The system clock runs at 66.67 MHz, which translates into 15 ns cycle time. The simplified diagram of the modulation section is shown in *Figure 5*. The modulation section consists of a RoboClock, a 256K*4 SRAM that contains the pixel information, four NOR gates with complemented outputs, and an 8:1 MUX. In this application, since the laser head interface uses ECL levels, 500 ps ECL NOR gate with complemented outputs (MC10E101) and ECL 8:1 MUX (MC10E163) is used. Unused inputs of the quad four input NOR gates are tied LOW. The TTL outputs of CY7B991 are translated into ECL levels by Cypress Semiconductor high-speed, low-skew TTL to ECL translator (CY10E384L). To keep the modulation logic diagram simple, the translate block is not shown in *Figure 5*. Note that, generally, 74AS logic parts are used to implement external logic functions, which requires no translate logic to interface with RoboClock.

The 66.67-MHz system clock is fed to the REF input of the RoboClock. RoboClock generates very precise waveforms and, with one level of gating, all the six modulated waveforms are produced and fed to the 8:1 MUX. For this design, RoboClock generates precise 90-degree phase-shifted, true and complemented versions of the 66.67-MHz REF input frequency. Note that only six waveforms are generated. The "100% on" and "100% off" modes are hardwired HIGH and LOW to the 8:1 MUX. Three bits of the SRAM are used to select one out of eight possible modulated signals. The output of very fast MUX is directly sent to the laser head. Therefore, all eight levels of modulated waveforms are present at the input of the MUX at all times. Only one is routed to the laser head, depending on the required modulation level stored in the SRAM.

Generally, one should be very cautious about using the output of a MUX, since during the period when MUX select bits are changing, the MUX output will usually be glitching, until the MUX select bits are stabilized. This behavior is due to the fact that all the select bits do not arrive at exactly the same time. Even if they did arrive at the same time, delay path variations and logic switching internal to MUX may create a glitch on the output. As a word of caution, the above mentioned scheme should not be used for a clocking scheme. When a new MUX input is se-

lected, there will probably be a glitch on the output. If the first cycle glitch can be tolerated or masked, then this scheme can be used for clock distribution. A delayed clocked version of the MUX output could be safely used for clock distribution. Obviously, the delay should be larger than the maximum propagation delay of MUX. For this particular application, the glitch is not as important, because the total duration of ON and OFF times of the laser beam is the concern, not the rising or falling edges of the waveform. Also, the laser head is turned off during the MUX address selection, totally masking any possible glitches.

Configuring RoboClock and Design Analysis

A close observation of waveforms shown in *Figure 5* reveals the fundamental idea behind generating all six modulated waveforms. Simply by gating the 90-degree phase-shifted REF with the true and complemented version of the REF clock, all six waveforms are generated. For simplicity's sake, lets call the 90-degree phase-shifted waveform $+4t_U$, 66.67-MHz clock F, and the inverted clock $\overline{F}$. Let's look at how each one is generated.

75% HIGH: $(+4t_U)$ OR (F)

50% HIGH: F

25% HIGH: $(+4t_U)$ NOR $(\overline{F})$

75% LOW: $(+4t_U)$ NOR (F)

50% LOW: $\overline{F}$

25% LOW: $(+4t_U)$ OR $(\overline{F})$

Note that very fast NOR gates with true and complemented outputs were selected to achieve uniform delay for all outputs. Also, the 50% HIGH and 50% LOW signals are routed OR gates configured as buffers to ensure matched delay signals. Please note that all three RoboClock outputs, $+4t_U$, F, and $\overline{F}$, have the same number of loads. It is very impor-

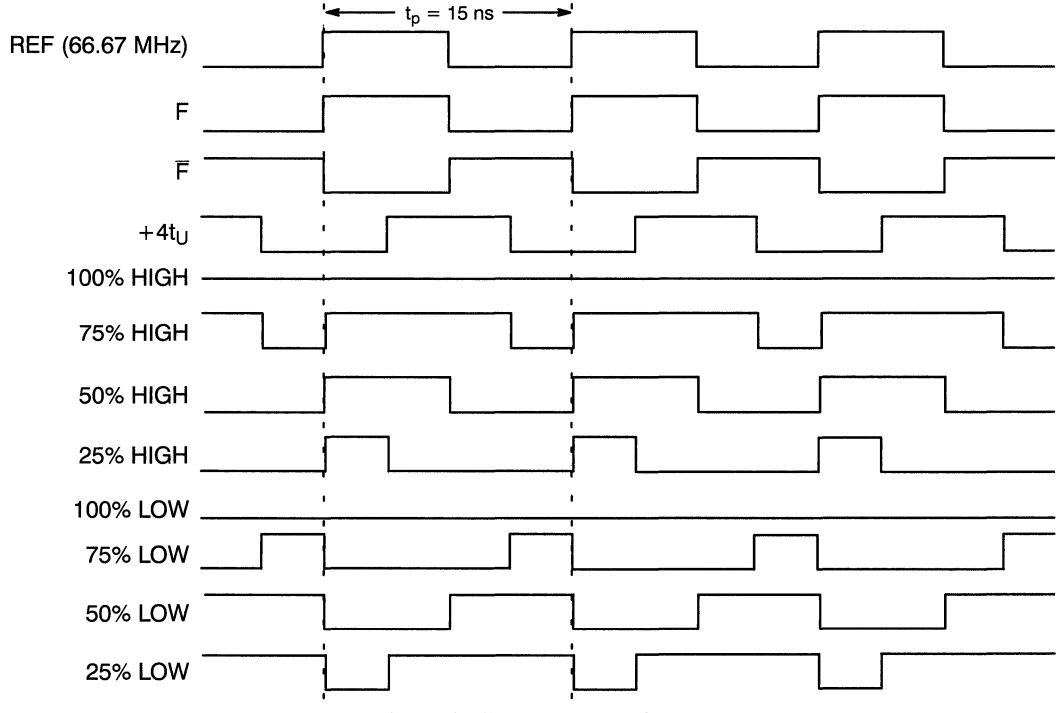


Figure 4. Generated Waveforms

tant during layout to match all the trace lengths from RoboClock to NOR gates and from the NOR gates to the MUX to prevent undesirable skew, which will translate into phase shift and pulsewidth variation on the laser beam.

Over voltage and temperature variation, all the outputs of the RoboClock are very stable. The PLL inside the RoboClock is constructed from differential stages, which makes it self-compensating against voltage and temperature variations. Consequently, RoboClock generates robust output waveforms in terms of phase and frequency. The external OR gates may distort the waveform due to the effects of voltage and temperature variation.

Earlier, the 90-degree phase-shifted waveform was equated with $+4t_U$. Let's see how that was derived. Based on *Table 1*, each time unit is calculated by the following equation:

$$t_U = 1/(F_{nom} * N) \qquad \text{Eq. 1}$$

Where, as indicated in the *Table 1*, N can be any one of 44, 26, or 16 integer numbers, depending on the maximum output frequency of the RoboClock. Since the output frequency is 66.67 MHz, then FS is selected to be HIGH (frequency range of 40 to 80 MHz), N = 16, and F_{nom} = 66.67 MHz. By simply plugging the numbers in the *Equation 1*, the time unit or the t_U can be calculated as:

$t_U = 1/(66.67\ MHZ * 16)$

$t_U = 0.9375\ ns$

In terms of phase shift, if 66.67 MHz or 15-ns cycle time is 360 degrees, then the 90-degree phase-shift is essentially 15 ns divided by 4.

90 Degree Phase Shift = 15 ns / 4 = 3.75 ns

Therefore, the number of time units to shift to obtain 90-degree phase-shift, is simply derived by dividing 3.75 ns by t_U.

Number of Time Units = 3.75 ns / 0.9375 = 4

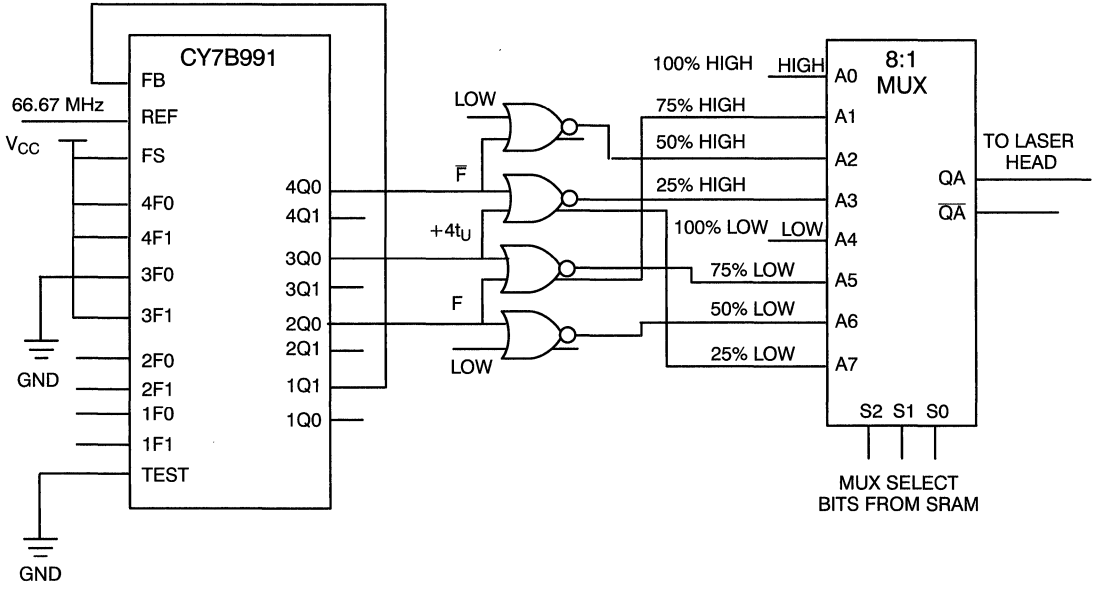


Figure 5. Simplified Laser Modulation Diagram

Therefore, in FS = HIGH mode, 4 t_U translates into 90-degree phase-shift. An observant reader might have already noticed the fact that in FS = HIGH mode, N is equal to 16, based on *Table 1*. This means that, in FS = HIGH mode, an entire cycle or 360 degrees, is equivalent to the delay through 16 stages of ring oscillator, and each stage represents one t_U (In FS = LOW the number of delay stages or N is 44 and in FS = MID it is 26. As shown in *Figure 6*, note that the actual number of ring oscillator buffer stages is half the N, because each cycle contains a LOW and HIGH period, which means to complete a full cycle the signal propagates through the ring oscillator twice.) In order to derive a 90-degree phase-shift, all one needs to do is to multiply N by 1/4 (where 90/360 = 1/4 cycle). Therefore, 16/4 = 4 time units, in FS = HIGH mode, represents a 90 degree phase shift.

The same simple methodology can be used to figure out the number of time units of delay or advance to implement a n arbitrary degree of phase shift. The number of units of skew (T_U needed for an arbitrary phase shift is calculated as follows:

$$\#T_U = \frac{N - phaseshift}{360} \qquad \text{Eq. 2}$$

Rounding this number to the nearest integer will introduce a small phase error from the desired phase shift. For example, if 60 degree phase shift is required when FS = LOW, then:

Required Phase Shift = 60/360 = 1/6 cycle

*Number of Time Units = N * 1/6 = 44/6 = 7.33* t_U

Since the number of PLL stages for each FS mode is an integer number, then the nearest time unit shift, in this case, will be seven. Obviously, this will create a phase error of 0.33 t_U.

Let's go back and discuss how the +4t_U, F and $\overline{F}$ waveforms are generated by RoboClock. Since multiple outputs from a single RoboClock is expected, as an exercise, let's use the three-table method. There are several solutions for the current requirement; only one of the simplest is presented. By observing *Table 3*, titled as "1Qx/2Qx Output Connected to FB Input," one may select the 1Q0 to be used as FB, and leave the corresponding inputs floating (1F0=1F1=MID). Thus, essentially, we have ac-

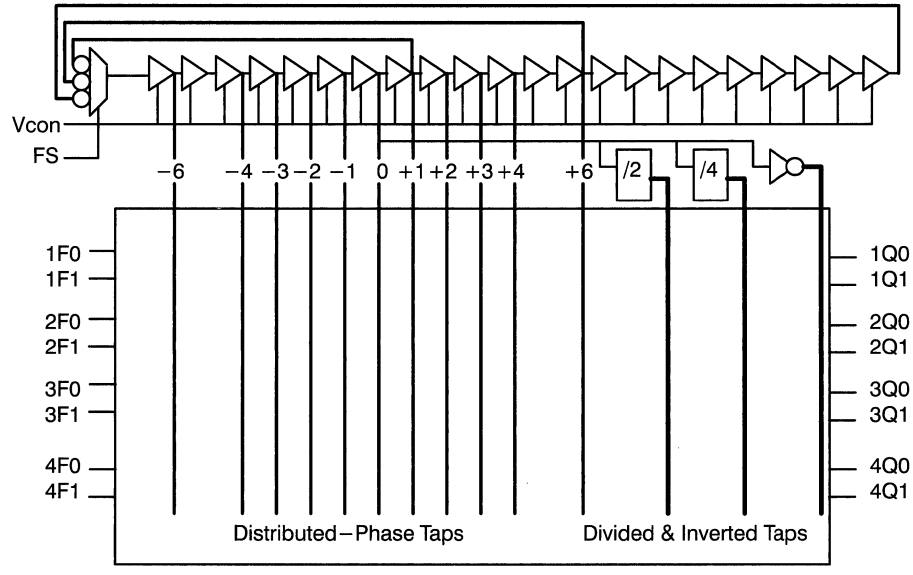


Figure 6. Distributed-Phase Clock Oscillator and Output Adjust Matrix

cess to all the terms available in row two of the given table. Now, by selecting the inputs, the RoboClock may be configured to generate various waveforms. By setting 2F0=2F1=MID or floating the 2Fx inputs, the 2Q0 and 2Q1 will generate the required F signal (also, 1Qx could have been used for F signal). Setting 3F0=LOW and 3F1=HIGH will generate $+4t_U$ signal at 3Q0 and 3Q1. Finally, setting 4F0=4F1=HIGH will generate the $\overline{F}$ signal. As mentioned earlier, by fixing the feedback term, in this case, all the elements of the 2nd row of *Table 3* are available for the user. RoboClock is a flexible clock distribution buffer that may be reconfigured easily during the prototyping phase of a design. For example, if instead of generating a 0 T_U output on 2Qx, it is required to have the 2Qx signals advanced by $2t_U$, then this can be accomplished simply by setting 2F0=HIGH and 2F1=LOW. This is one of the commonly used features of RoboClock that offers thousands of variations for prototyping purposes. Often, during prototyping phase, some modification in the clock or the waveform is required.

RoboClock, with its thousands of configurations, resolves some of the unexpected timing problems. In fact, during prototyping, if multiple timing variations are expected, it is advised to use a three-state register to drive the RoboClock inputs. Then, each output of the register must have a 10K pull-up and 10K pull-down resistor, to ensure that MID level is held at half the supply voltage when the register is three-stated. In this case, the user may write a word in the input register, and by doing so, reconfigure the entire operation of the RoboClock, without using any jumpers. Note that not all the inputs need to be reconfigurable for a given design. Often, a couple of reconfigurable signals are all that is needed. In that case, most inputs may be hardwired and the inputs needed to reconfigure various outputs may be registered with the 10K pull-up and pull-down resistors.

Summary

RoboClock was used to generate very precise complex waveforms to enhance color images and increase the resolution of laser printers. Even though RoboClock is widely used for clock distribution, this application note presented an alternative use of RoboClock for complex precise waveform generation.

Table 3. 1Qx or 2Qx Output Connected to FB Input (Part 1)

Feedback Section		2Qx Output Section									
1Qx(2Qx)→FB		Configuration Block	2Qx(1Qx) Outputs with respect to REF								
			2F1 (1F1)	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		2F0 (1F0)	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
L	M			−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t
L	H			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	L			−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t
H	L			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	M			−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t
H	H			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t

Table 3. 1Qx or 2Qx Output Connected to FB Input (Part 2)

Feedback Section		3Qx Output Section									
1Qx(2Qx)◗FB		Configuration Block	3Qx Output with respect to REF								
			3F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, f/4
L	M			+3t, f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t, f/4
L	H			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, f/4
M	L			+1t, f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t, f/4
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, f/4
M	H			−1t, f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t, f/4
H	L			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, f/4
H	M			−3t, f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t, f/4
H	H			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, f/4

Table 3. 1Qx or 2Qx Output Connected to FB Input (Part 3)

Feedback Section		4Qx Output Section									
1Qx(2Qx)◗FB		Configuration Block	4Qx Output with respect to REF								
			4F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, INV
L	M			+3t, f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t, INV
L	H			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, INV
M	L			+1t, f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t, INV
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, INV
M	H			−1t, f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t, INV
H	L			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, INV
H	M			−3t, f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t, INV
H	H			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, INV

Table 4. 3Qx Output Connected to FB Input (Part 1)

Feedback Section		1Qx, 2Qx Output Section									
3Qx ➧ FB		Configuration Block	1Qx (2Qx) Output Delay to REF								
			1F1, (2F1)	L	L	L	M	M	M	H	H	H
3F1	3F0		1F0, (2F0)	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		−4t, f*2	−3t, f*2	−2t, f*2	−1t, f*2	0t, f*2	+1t, f*2	+2t, f*2	+3t, f*2	+4t, f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t, f*4	−3t, f*4	−2t, f*4	−1t, f*4	0t, f*4	+1t, f*4	+2t, f*4	+3t, f*4	+4t, f*4

Table 4. 3Qx Output Connected to FB Input (Part 2)

Feedback Section		3Qx Output Section									
3Qx ➧ FB		Configuration Block	4Qx Output with Delay to REF								
			4F1	L	L	L	M	M	M	H	H	H
3F1	3F0		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	−6t, f*2	−4t, f*2	−2t, f*2	0t, f*2	+2t, f*2	+4t, f*2	+6t, f*2	INV, f*2
L	M			+6t, f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t, INV
L	H			+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, INV
M	L			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, INV
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, INV
M	H			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, INV
H	L			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, INV
H	M			−6t, f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t, INV
H	H			0t, f*2	−6t, f*4	−4t, f*4	−2t, f*4	0t, f*4	+2t, f*4	+4t, f*4	+6t, f*4	INV, f*4

Table 5. 4Qx Output Connected to FB Input (Part 1)

Feedback Section		1Qx, 2Qx Output Section									
4Qx ➧ FB		Configuration Block	1Qx, 2Qx Output with respect to REF								
			1F1, 2F1	L	L	L	M	M	M	H	H	H
4F1	4F0		1F0, 2F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		−4t, f*2	−3t, f*2	−2t, f*2	−1t, f*2	0t, f*2	+1t, f*2	+2t, f*2	+3t, f*2	+4t, f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t, INV	−3t, INV	−2t, INV	−1t, INV	0t, INV	+1t, INV	+2t, INV	+3t, INV	+4t, INV

Table 5. 4Qx Output Connected to FB Input (Part 2)

Feedback Section		1Qx, 2Qx Output Section									
4Qx ➧ FB		Configuration Block	4Qx Output with respect to REF								
			3F1	L	L	L	M	M	M	H	H	H
4F1	4F0		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	−6t, f*2	−4t, f*2	−2t, f*2	0t, f*2	+2t, f*2	+4t, f*2	+6t, f*2	0t, f/2
L	M			+6t, f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t, f/4
L	H			+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, f/4
M	L			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, f/4
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, f/4
M	H			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, f/4
H	L			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, f/4
H	M			−6t, f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t, f/4
H	H			INV, f/2	−6t, INV	−4t, INV	−2t, INV	0t, INV	+2t, INV	+4t, INV	+6t, INV	INV, f/4

CY7B991 and CY7B992 (RoboClock) Test Mode

This application note discusses the Test mode capabilities of the CY7B991 and CY7B992 (RoboClock) devices. It begins with an introduction to these devices and then discusses how to use the Test mode features.

Introduction

The RoboClock family consists of two parts: the CY7B991 and CY7B992. The CY7B991 has TTL (0 to 3V) outputs and the CY7B992 has CMOS (0 to Vcc) outputs. Each device will drive 50Ω terminated transmission lines. *Figure 1* shows the PLCC and LCC pin configurations for these devices.

RoboClock (*Figure 2*) employs a phase-locked-loop architecture. Connecting an output to the FB (feedback) input of the device causes the PLL to synchronize and align this output both in phase and in frequency with the REF (reference) input. This results in very low input to output delay and allows a system to connect RoboClocks in parallel for clock distribution while maintaining very low skew between various clocks from different devices.

RoboClock contains eight outputs grouped in four sets of two. Two function select lines (xF0, xF1) control the functionality of each pair of outputs (xQ0, xQ1). The outputs of an output pair operate identically.

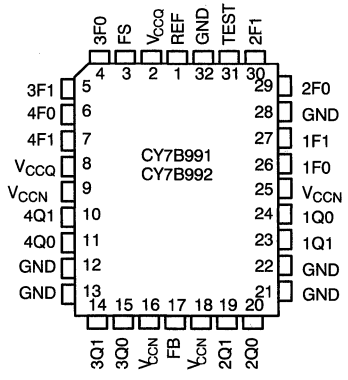


Figure 1. PLCC and LCC Pin Configuration

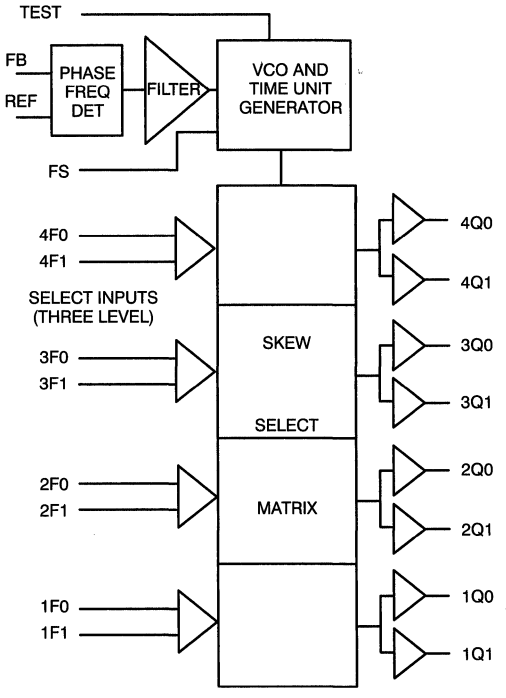


Figure 2. RoboClock Block Diagram

Table 1. Programmable Skew Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	**1F0, 2F0, 3F0, 4F0**	**1Q0, 1Q1, 2Q0, 2Q1**	**3Q0, 3Q1**	**4Q0, 4Q1**
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

Each pair of three-level function select inputs allows you to hardwire the operation of each output pair to one of nine delay or functional configurations. Each function select input pin can be connected to V_{CC} (HIGH), left unconnected (MID), or connected to ground (LOW). *Table 1* shows the programmable skew configurations available on each output pair. The function select configurations in *Table 1* assume that the output connected to FB is set for "zero" skew.

Table 1 shows the range of t_U over which an output may be skewed with respect to the REF input. t_U is a function of the frequency at which the 1Q0 output is operating. RoboClock offers frequency coverage with three ranges from 15 MHz to 80 MHz with the use of the three-level FS (frequency select) input. *Table 2* shows the operating frequency range for each of the three levels of FS. The appropriate FS level selection must be made such that the anticipated operating frequency of the 1Q0 output is within the specified limits. There may be two acceptable levels for the FS pin when operating at certain frequencies. The appropriate connection of the FS pin, in this case, would be based on the value of the time unit, t_U, required for the application. *2* also shows an equation that can be used to calculate t_U as well as the approximate operating frequency where t_U is equal to 1 ns.

For example, according to *2*, a system using RoboClock with a clock speed of 33 MHz would leave the FS pin unconnected. The programmable time unit, t_U, based on this operating frequency, would be

$$t_U = \frac{1}{f_{1Q0} \times N} = \frac{1}{33\ \text{MHz} \times 26} = 1.17\ \text{ns} \qquad \text{Eq. 1}$$

In other words, you can adjust the position with which the rising and falling edges of the outputs move with respect to the corresponding REF input edge with a resolution of 1.17 ns when operating the device at 33 MHz. At 25 MHz the t_U could be either .91 ns or 1.54 ns depending on whether the FS pin is tied LOW or left unconnected, respectively.

Table 2. Frequency Range Select and t_U Calculation

	f_{1Q0} (MHz)		$t_U = \frac{1}{f_{1Q0} \times N}$	
FS	**Min.**	**Max.**	**where N =**	**Approximate Frequency At Which t_U = 1.0 ns**
LOW	15	30	44	22.7 MHz
MID	25	50	26	38.5 MHz
HIGH	40	80	16	62.5 MHz

Test Mode Features

In some situations you may need to stop the PLL of the device. For instance, in many board-level testing applications you may need to supply a clock input to the system that may not meet the REF input requirements of RoboClock. This scenario can occur in bed-of-nails testing or single-step microprocessor execution. Use of the TEST input of RoboClock will allow operation in single-step mode.

The TEST input is a three-level input. In normal system operation, this pin is connected to ground, allowing RoboClock to operate as previously explained. (For testing purposes, any of the three-level inputs can have a removable jumper to ground, or be tied LOW through a 100Ω resistor. This will allow an external tester to change the state of these pins.)

If the TEST input is forced to its mid or HIGH state, the device will operate with its internal phase-locked-loop disconnected. The TEST input must be forced to less than 1V to insure its LOW level, to $V_{CC}/\pm 500$ mV to insure its MID level, and to $V_{CC} - 1V$ to insure its HIGH level.

When RoboClock is put in Test mode, after a few REF cycles, input levels supplied to REF will appear at all outputs after a 15- to 80-ns delay. The circuit effectively becomes a long chain of delay elements. The level on the TEST input affects the length of time it takes for the REF signal to propagate through each delay element. When the TEST input is forced HIGH, each delay element will be selected to have its shortest delay (< 700 ps). This is known as "contracted" mode. When the TEST input is forced to its mid state, the delay through each element will be as long as possible (> 1.5 ns). This is referred to as "extended" mode.

The level placed on the FS pin also determines the operation of RoboClock when it is in Test mode. The FS input is used to control the number of delay elements that the REF input will propagate through, as shown in *Figure 3*. When FS is held HIGH, REF will pass through only the last 13 delay stages. When FS is placed in the MID or LOW position, REF will propagate through all 22 delay elements.

In contrast with normal operation (TEST tied LOW), FB will not have any affect on the operation of the outputs. All outputs will function based only on the connection of their own function select inputs (xF0 and xF1) and the waveform characteristics of the REF input.

Outputs that have the divide-by-two output configuration selected will change state at every second REF input, and outputs that have the divide-by-four

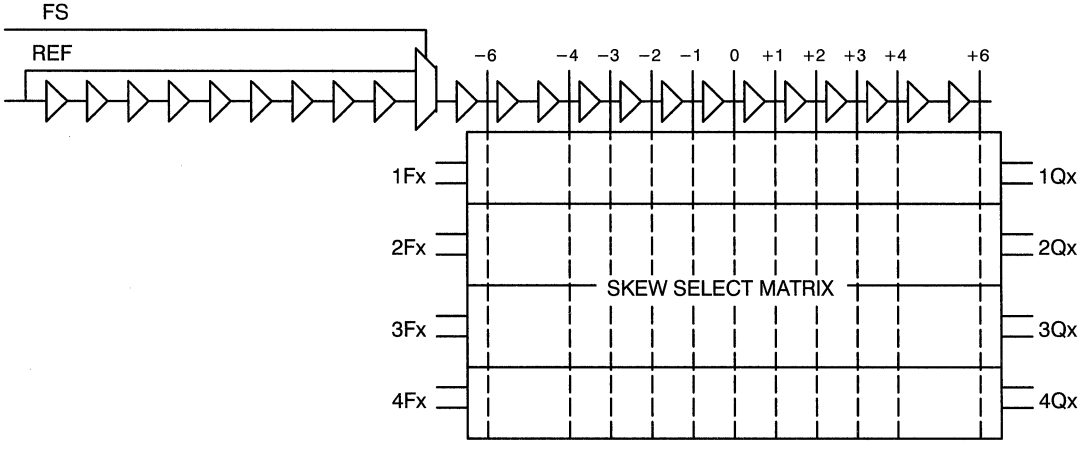


Figure 3. RoboClock Test Mode

option selected will change state at every fourth REF input. An output selected for inverted operation will drive the opposite sense of the REF input.

A counter reset is available for the divided outputs. To reset the counters, the 3F0 and 4F0 function selects must be placed in their MID position and a clock applied to the REF input. If the 3Qx or 4Qx outputs are then selected for a divided function (3Fx = LOW, LOW, HIGH, HIGH or 4Fx = LOW, LOW) then the 4Qx or 3Qx outputs will be in their HIGH state. The first REF clock will cause these outputs selected for divided operation to transition LOW and, subsequent REF clocks will cause these outputs to continue their normal output divided output pattern.

Conclusion

RoboClock's Test mode feature stops the phase-locked-loop allowing board-level testing and evaluation. This mode allows operation at frequencies below the minimum operating frequency. It also provides the ability to apply input pulses with varying width and period to the device without requiring the cycle-to-cycle frequency accuracy necessary to keep the feedback loop in lock.

Bus Products – 8

Bus Products Section Contents and Abstracts

This document provides answers to the questions most frequently asked by customers who are evaluating and using Cypress VMEbus Interface products. These answers will serve as an introduction for each topic. Separate application notes cover these topics in more complete detail.

This application note describes the use of the CY7C960/961 Slave VME Interface Controller in a simple slave VME board design. This slave VME board is fully compliant with the VME64 Specification and contains both SRAM and DRAM. Emphasis is placed on the design of the region decoder, SWAP buffer, interrupt logic, DRAM interface and the connections to the CY7C964 Bus Interface Logic Circuits. Included at the end of this application note is a printout of the VHDL code used to implement some of the logic used on the board.

This application note introduces the CY7C964. CY7C964 operating modes and features are described. Also discussed is the ease of use with either the VIC64 or VIC068A. A sample circuit schematic is included showing the CY7C964 to VIC interface.

Information is provided on the different signals present on the CY7C964 and the potential problems that could be encountered when using the device. This application note compliments the information provided within the VIC64/CY7C964 Design Notes.

This application note gives a broad overview of the VIC068A. It outlines some of the major features of the device including: master write posting, slave write posting, read-modify-write cycles, block transfer cycles, interprocessor communication facilities, and interrupt handling.

This application note explains some of the implementation details of interfacing the VIC068A to a Motorola MC68020 microprocessor. Emphasis is given for A24/D16 type designs. Resetting the VIC068A is given much attention in this application note. A ROM remapping circuit is described showing how the MC68020 obtains its stack pointer and program counter at reset. A sample schematic shows how to interface the VIC068A to the MC68020. PLD equations are given which provide the address decoding. Finally, master and slave cycles are described showing how all these pieces are used together to provide a full function interface.

This application note provides high-level as well as low-level details of interfacing VIC/VAC to TMS320C40. This allows for techniques to be implemented to minimize design time for subsequent efforts since this design has not been optimized for either size or speed. The Design Requisites section provides the design goals established prior to design as well as relevant background regarding devices involved. Hardware details, including schematics and programmable logic source code, represent the central focus of the paper. In addition, software initialization of the chip, set by the TMS320C40, is covered. Throughout this note, it is assumed that the reader is familiar with the TMS320C40 architecture, the basics of the VIC068A/VAC068A, and the VMEbus and its protocol(s).

This application note provides a VIC64 (or VIC068A) designer with proven tips and examples for both configuring and operating the VIC64 or VIC068A. The software described was based on an actual VIC64 design. This application note also describes configuring the CY7C964 address comparator functions. Sample C source files are also described (the actual source files are available on the Cypress BBS) showing a Block Transfer utility.

This application note shows how the VIC64 can be interfaced to a Motorola 68040 microprocessor operating at 40 MHz. The issues and assumptions that go into designing such an interface are considerable and complex; thus, this application note will not attempt to design a complete VME board that can do everything. It will cover some of the issues that are pertinent when designing a 68040-based VMEbus board and will focus on the circuitry required for VIC64 to 68040 interfacing.

This application note describes an interface between the CY7C611A SPARC microprocessor and the VIC64. The interface described within this application note couples the synchronous bus of the CY7C611A to the asynchronous bus of the VIC64. The interface is high performance and preserves many of the features necessary for VMEbus applications, such as the memory exception facility.

The application note discusses the high and low level implementation of a the interface. A CY7C361 and 22V10 PLDs implement the design. State diagrams and timing waveforms are included. The PLD source files for the design are available on the Cypress Semiconductor BBS.

This application note provides an example of how to design a "dumb" slave-only VME board that does NOT have a local microprocessor. The article focuses on the design of a VME arbiter between a Slave VIC (SVIC) and the host microprocessor (Motorola 68020).

This application note gives a quick overview of the RACEway products and support materials available from Cypress Semiconductor.

This document describes PitCREW, a RACEway interface. PitCREW is an I/O data port for RACEway. It defines a simple FIFO interfaced local data port which is a slave to its RACEway port. The PitCREW has an internal DMA engine which moves blocks of data between RACEway nodes and its FIFO port.

This document describes PitCREWjr, a RACEway interface. PitCREWjr is a simple full-duplex on-ramp to the RACEway fabric. The device has a standard RACEway port and a FIFO port. The controller functions as a RACEway slave or master, moving data between RACEway and local FIFOs.

Frequently Asked Questions about the VMEbus Products

The following questions are frequently asked by customers who are evaluating and using Cypress VMEbus Interface products. These answers will serve as an introduction for each topic. Separate application notes cover these topics in more complete detail.

Section I. Questions Regarding Reset

1. What are the requirements to reset the VIC at power-up?

To properly reset the VIC at power-up, it is required that the VIC see a falling edge on the $\overline{\text{IRESET}}$ signal after the following criteria have been met:

1. The input voltage has reached 5V.
2. The CLK64M clock input is operating within the required specifications.
3. All VMEbus signals are within VMEbus specifications.
4. Local input and three-state I/O signals are driven to a deasserted value (LD[7:0] and LA[7:0] may be left floating).

$\overline{\text{IPL0}}$ must be asserted no earlier than 16 ns (20 ns for military devices) after $\overline{\text{IRESET}}$ has been asserted. This will initiate a global reset. The minimum pulse width for $\overline{\text{IPL0}}$ is 50 ns. See section 12 of the *VIC068A User's Guide* for more details.

2. What is the best way to implement a power-up reset?

Best results have been obtained when the power-up reset is initiated through software during system boot. That is, dedicate two external register bits to be tied to the $\overline{\text{IRESET}}$ and $\overline{\text{IPL0}}$ signals. During system boot-up, have the processor write to these bits in a way that first asserts the $\overline{\text{IRESET}}$ signal, then asserts the $\overline{\text{IPL0}}$ signal, then negates the $\overline{\text{IPL0}}$ signal, and finally negates the $\overline{\text{IRESET}}$ signal. Since the processor must be operational before the VIC, this implies that the $\overline{\text{RESET}}$ output signal may not be used to reset the processor. Sample SPARC™ assembler code for this type of reset may be found in the application note "Software Considerations for the VIC64."

As the VIC must see a falling edge on $\overline{\text{IRESET}}$ when the system is stable (see question 1, above), an RC network should not be used to reset the VIC on power-up.

3. Can the VIC or the local module be remotely reset over the VMEbus?

The assertion of $\overline{\text{SYSRESET}}$ on the VMEbus will reset the internal circuitry and selected internal register bit fields on the VIC. This is referred to as a system reset because $\overline{\text{SYSRESET}}$ is typically used to reset all modules on the VMEbus.

If an individual module reset is desired (without resetting the entire system), ICR7 (Interprocessor Communication Register 7) bit 6 can be set. This will assert $\overline{\text{HALT}}$ and $\overline{\text{RESET}}$ from the VIC, which can be

used to reset local bus devices on a specific module. However, when this bit is set, no external VMEbus masters can access the VIC, so provisions must be made to issue an $\overline{\text{IRESET}}$ from the local side. Asserting $\overline{\text{IRESET}}$ (for a minimum of 20 ns) will cause the VIC to initiate an internal reset. Upon being granted the local bus (or if no grant is asserted to the VIC within 1 µs, a timer will expire and the VIC will proceed as if it had been granted), the VIC will drive $\overline{\text{HALT}}$ and $\overline{\text{RESET}}$ for 200 ms intervals until $\overline{\text{IRESET}}$ is deasserted. When the VIC detects $\overline{\text{IRESET}}$ deasserted at the end of the 200 ms timeout period, it will deassert $\overline{\text{HALT}}$ and $\overline{\text{RESET}}$, bringing the local module out of reset. Upon the assertion of $\overline{\text{IRESET}}$, the VIC will change the state of its internal registers. The internal registers must be reloaded.

For power-up reset, a global reset must be used (to ensure that all internal VIC registers are set to their default values). *See questions 1 and 2.*

4. Does the VIC drive the local bus when $\overline{\text{IRESET}}$ is asserted?

No. After $\overline{\text{IRESET}}$ is asserted, the VIC attempts to arbitrate for the local bus. If the VIC is granted the bus or a 1 µs timer expires, the VIC will assert $\overline{\text{HALT}}$ and $\overline{\text{RESET}}$, deassert its local bus request, place all three-state outputs in high-Z, and begin a 200-ms timeout period. If $\overline{\text{IRESET}}$ is still asserted after 200 ms, additional 200-ms timeout periods follow until $\overline{\text{IRESET}}$ is deasserted.

Section II. Questions Regarding Interrupts

5. Can the VIC queue up multiple interrupts with the same IPL value?

No. The VIC will queue all pending interrupts that are on different levels. If back-to-back interrupts are required on the same level, the first interrupt will have to be handled before the second interrupt is recognized. It is legal for the VIC to continue to drive the IPL lines to the same level if back-to-back local interrupts are requested on the same level, but the interrupts must be requested sequentially.

6. Is there a way to check the level of VMEbus interrupts in the VIC?

If the interrupt was generated by writing to the VIRSR (VMEbus Interrupt Request/Status Register), the level can be checked by reading the VIRSR. Otherwise, the only way to check the level is to allow the local processor to perform the interrupt acknowledge cycle. The proper vector will be generated, which should allow software to determine the interrupt level by jumping to the specific interrupt handler. The vector can also be seen with a logic analyzer during the interrupt acknowledge cycle.

7. What is the minimum pulse width for the LIRQ signals?

One CLK64M clock period. The LIRQ lines are internally registered by the VIC. Therefore, if the local interrupt request lines are asserted for at least one 64-MHz clock period, the VIC is guaranteed to sample and recognize the asserted request lines on a CLK64M clock edge.

8. When does the VIC latch the IPL lines?

$\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, and $\overline{\text{IPL0}}$ are the local priority encoded interrupt request signals. They are used to interrupt the local processor. These signals emulate the Motorola 68K interrupt mechanism. The IPL lines are latched on the assertion of the $\overline{\text{FCIACK}}$ signal. $\overline{\text{FCIACK}}$ should be asserted by the processor to tell the VIC that an interrupt is being acknowledged. Once the VIC detects the assertion of $\overline{\text{FCIACK}}$, it samples LA[3:1] to determine whether the interrupt acknowledge is for the VIC's pending interrupt. If the acknowledge was intended for the VIC, it will either pass the acknowledge to the VMEbus (for VMEbus initiated interrupts) or provide the appropriate acknowledge signals to the local bus (for local bus initiated interrupts). The IPL lines can change after the $\overline{\text{FCIACK}}$ signal is deasserted. The assertion of $\overline{\text{DSACK0}}$ or $\overline{\text{DSACK1}}$ by the VIC indicates that the acknowledge matches the interrupt level that the VIC is currently requesting.

Section III. Questions Regarding Register Operations

9. Can the VIC registers be programmed over the VMEbus?

VIC registers (other than the ICF registers) cannot be directly programmed over the VMEbus. They can be accessed, however, by having the address decoder drive CS to the VIC.

Section IV. Questions Regarding Arbitration

10. How must the local bus arbiter operate?

The VIC (or any other local bus master) will assert its own $\overline{\text{LBR}}$ whenever it needs to access the local bus. The arbiter must assert a specific $\overline{\text{LBG}}$ for one master allowing the access to occur. The VIC will maintain its $\overline{\text{LBR}}$ until it no longer wants the local bus. It is up to the system designer to pick an arbitration scheme (assigning priorities to each master, insuring that no master will be "starved" off of the bus, etc.). Arbiters must also monitor the $\overline{\text{DEDLK}}$ signal to prioritize the local bus grant to the VIC during deadlock situations.

Once the VIC has been granted the local bus, it is important that the $\overline{\text{LBG}}$ signal to the VIC not be removed until its $\overline{\text{LBR}}$ is deasserted. The VIC will keep its $\overline{\text{LBR}}$ asserted through its entire cycle.

11. Can $\overline{\text{LBG}}$ on the VIC be tied HIGH?

Only if the designer can insure that the VIC will never be the local bus master. The VIC requires local bus mastership when there are VME slave accesses, VME block transfers, or VME DRAM refreshes performed on the board.

12. Does the VIC support early release of $\overline{\text{BBSY}}$?

Yes. If the Release When Done release mode has been selected, the VIC will deassert $\overline{\text{BBSY}}$ upon the last assertion of $\overline{\text{AS}}$.

Section V. Questions Regarding Deadlock

13. When is $\overline{\text{DEDLK}}$ asserted?

When the $\overline{\text{MWB}}$ signal or $\overline{\text{FCIACK}}$ and a valid slave select occur at the same time, the VIC will assert $\overline{\text{DEDLK}}$ to force the processor to remove $\overline{\text{MWB}}$ or $\overline{\text{FCIACK}}$ and retry the transaction later. The VIC will not detect a deadlock situation when $\overline{\text{CS}}$ or $\overline{\text{IFCSEL}}$ is asserted (a VIC register access) at the same time as a valid slave transaction to the VIC.

14. How does the system recover from a deadlock?

If a deadlock occurs, the VIC will assert the $\overline{\text{DEDLK}}$ signal (or a combination of $\overline{\text{DEDLK}}$ and $\overline{\text{LBERR}}$ and/or $\overline{\text{HALT}}$, which can be programmed to occur on deadlocks). $\overline{\text{DEDLK}}$ must go to the arbiter to prioritize the local bus grant to the VIC (so it can perform the slave access). During a deadlock the processor will not have access to the VME bus as a master until the slave transaction has been completed. All other local transactions will not be affected by the deadlock.

15. Can deadlocks be disallowed?

If the system designer can guarantee that no master will try to access local memory on a VMEbus board, the board does not have to support deadlocks. Otherwise, they cannot be disallowed.

Section VI. Questions Regarding Block Transfers

16. Can block transfers be interrupted or aborted?

The only way to abort a block transfer is by asserting $\overline{\text{LBERR}}$. However, when $\overline{\text{LBERR}}$ is asserted, the status will be saved (bits in the DMASR, etc.). Also the assertion of $\overline{\text{LBERR}}$ will cause the VIC to assert VMEbus $\overline{\text{BERR}}$, which can have severe system ramifications. If block transfers are taking too much local bus/VMEbus bandwidth, the block size should be shortened or the block should be broken up using interleaving. Breaking up the block is a cleaner solution.

17. What is the maximum block transfer?

The VMEbus specification prohibits the crossing of 256-byte boundaries during block transfers (2K-byte boundaries for VME64). The VIC allows for larger block transfers by deasserting AS, incrementing the address, and reasserting AS without relinquishing the VMEbus whenever a boundary is crossed. The boundary crossing feature is enabled by setting bit 2 in the BTDR, Block Transfer Definition Register (bit 7 for the VIC64 with 2K-byte boundaries).

Without using CY7C964s or the VAC with the VIC, the maximum block transfer is 256 bytes (2^8). This is because the VIC only has direct control over the lower order VME address lines (A[7:1]).

If a VAC or CY7C964s are used in conjunction with the VIC068, 64K bytes (2^{16}) can be transferred in a block. For the VIC64, the maximum block size is 16M bytes (2^{24}). The increase in block size is due to the fact that the VAC or CY7C964s give complete access to the 32 VME address signals so the block address can be incremented past A7. The 64K-byte VIC and 16M byte VIC64 constraints are due to the fact that there are two eight-bit registers in the VIC068 (BTLR0 and BTLR1) and three eight-bit registers in the VIC64 (BTLR0, BTLR1, and BTLR2) to define and control the block transfer length.

18. Can the VIC perform D8 block transfers?

No. The least significant bit of BTLR0 should be cleared. If the least significant bit is set, the block transfer length is ignored and only one burst is performed.

Section VII. Questions Regarding Slave Operations

19. Can the VIC be used to implement a slave-only interface without using a microprocessor?

This can be done, but external logic must be provided to load the VIC's internal registers. Please see the Application Note entitled "Using VIC068A on a Board Without a Microprocessor," *Cypress Applications Handbook*, 1993. Cypress also offers slave-only interface chips, CY7C960 and CY7C961.

20. Can SLSEL0 and SLSEL1 be programmed to respond to more than one address space each?

No. Each slave select signals can only respond to one address space at a time.

Section VIII. Questions Regarding Modeling/Schematic Capture

21. Are schematic capture libraries available for the VIC?

A VIC schematic in OrCAD is available on the Cypress BBS (408-934-2954).

22. Are simulation libraries available for the VIC?

Verilog models are available for the VIC068A, VIC64, VAC068A, and CY7C964. Verilog behavioral models of standard VMEbus transactions are available as well. They work with Cadence's Verilog package. Contact your local Cypress Field Applications Engineer to obtain them.

Section IX. Questions Regarding Electrical Characteristics

23. What are the thermal characteristics for Cypresses VMEbus products?

Package	Theta JC (Degrees C/Watt)	Theta JA (Degrees C/Watt)	Description
B144	11.0	38.0	144-Pin Plastic PGA
G145	4.0	24.0	144-Pin Ceramic PGA
N160	13.0	34.3	160-Pin PQFP
A144	7.2	45.1	144-Pin TQFP
U162	6.5	26.0	160-Pin CQFP
N65	17.7	81.3	64-Pin TQFP 14mm
A64	18.2	108.0	64-Pin TQFP 10mm
U65	3.0	80.7	64-Pin CQFP
G68	4.0	28.4	68-Pin Ceramic PGA

24. What is the maximum power consumption for the VIC?

The VIC and the VAC consume 0.75W max each. The I_{CC} is rated at 150 mA max. The parts typically consume 50 mA.

Section X. Miscellaneous Questions

25. Is there a test mode/pin to three-state all of the VIC's outputs for testing purposes?

No.

26. Can all of the VIC's inputs and outputs be treated as synchronous signals clocked off of CLK64M?

No. All inputs and outputs should be treated as asynchronous. There are internal synchronizers to sync the external signals to the CLK64M clk for the purpose of running the VIC's internal state machines synchronously, but there are no guaranteed timing relationships between any of the signals and CLK64M.

27. Does the VIC have internal clamping diodes?

The signals are clamped to 5V (to help prevent overshoot problems). There are no clamping diodes to GND.

28. What values of capacitors are recommended for decoupling?

0.10 µF for AC bypass and 100 pF (or 470 pF) for high frequency decoupling. Four of each is recommended. They should be laid out as close to the V_{CC} pins as possible with wide traces (if possible) to eliminate some of the inductive effects.

29. What kind of throughput can be expected from the VIC?

The design group was able to achieve 61.6 Mbytes per second using the VIC64, 30 Mbytes per second using the VIC068. Over 70 Mbytes per second is possible using the VIC64. This maximum is usually dependent on system constraints rather than interface components.

30. What is the die size for the VIC068?

315x300 mils for the VIC068A and VIC64, 313x300 mils for the VAC068A, and 144x133 mils for the CY7C964.

31. Using the VIC with CY7C964s (or the VAC), is there any way to avoid violating the 2-inch VMEbus rule?

Users should consider this rule as a guideline. The rule is nearly impossible to meet using any standard VMEbus interface chipset. Traces from the VIC/CY7C964s/VAC to the VMEbus connectors should be kept as short as possible.

32. How many CY7C964s should be used with the VIC?

Each CY7C964 controls 8 bits of both address and data. The VIC068A and VIC64 also control 8 bits of address and data. Users can determine how many CY7C964s are needed to complete their interface by determining which address and data transactions will be supported. An A32/D32 interface would require three CY7C964s. See the *VIC64/CY7C964 Design Notes* from Cypress Semiconductor for more information on the CY7C964 and how to connect it to the VIC.

33. How many gates are in the VIC068A/VAC068A?

19,435 in the VIC068A; 21,250 in the VIC64; 18,106 in the VAC06A; 3000 in the CY6C964. The transistor counts are as follows: 80,000 for the VIC068A, 85,000 for the VIC64, 75,000 for the VAC068A, and 12,000 for the CY7C964.

34. What is the capacitive loading on the VIC signal lines?

5 pF on inputs. 7 pF on outputs. 13 pF on bidirectional signals.

35. How many words can be write posted to the VIC from the local and the VMEbus side?

One longword can be write posted from either side.

36. Which VIC signals have metastability protection?

Metastability is a problem with all asynchronous, clocked designs. If a valid level is not reached on the input to a clocked element (flip-flop, etc.) within the specified set-up and hold window, the condition called "metastability" can occur. The output of the clocked element is unpredictable. It may be driven to a valid output level or even oscillate. Eventually the output will settle to a valid level, but the settling time may also be unpredictable. There are several ways to combat metastability problems. One of the most common techniques involves "double clocking" the input. Two clocked elements are placed, in series, in the signal path. Even if the first clocked element goes metastable, the odds are good that the output will have settled to a valid state before the set-up and hold window of the second element is reached.

All of the VMEbus strobe inputs to the VIC are metastability-hardened and carry with them 2–3 CLK64M cycles of synchronization delay. $\overline{\text{DSi}}$, $\overline{\text{DTACK}}$, and $\overline{\text{BERR}}$ are also metastability-hardened. $\overline{\text{AS}}$ has both an asynchronous path and a metastability-protected path. When performing slave transfers, the asynchronous path is used.

The VME data bus, address bus, AM5-0, LWORD, WRITE, and all of the local bus signals are not metastability-hardened.

37. Is there any example "C" code available for programming the VIC?

Yes. A file named SAMPCODE.EXE is available on the Cypress BBS (408) 943–2954. This is a self-extracting file.

Using the Slave VIC (CY7C960/961)

Many VME boards, especially I/O boards, need only be aware of VME Slave transactions. Most commercially available VME interface chips are capable of both Master and Slave VME transactions and require some local intelligence, such as a microprocessor, to reset and program the interface chip. I/O-only boards do not need a microprocessor since information is simply passed to and from the I/O without being processed in between (at least in the simplest case) so the addition of a microprocessor, or any other kind of intelligence such as a state machine, only adds to the cost of the interface in design time, board space, and money. The most common solution to the problem of a slave-only interface is an FPGA, which still adds extra cost in the form of design time, board space, and the cost of the FPGA.

A better solution to this problem is Cypress's Slave VME Interface Controller (SVIC) Family: the CY7C960 and the CY7C961. An SVIC, along with four Bus Interface Logic chips (CY7C964), implements a complete VME64-compliant slave-only VME interface that requires no microprocessor and occupies minimum board space.

Index

CY7C960/961 Features

- Full VME64 Slave transaction support
- DRAM/Refresh Controller
- CY7C964 Control Interface
- I/O (Chip Select Output) Controller
- VMEbus Interrupter
- Address Modifier (AM) Code Discriminator
- Slave Address Region Decoder
- Limited Master Support (CY7C961 only)

Slave VIC Operation Overview

Figure 1 shows the internal blocks that comprise the CY7C960. The CY7C960 Slave VMEbus Interface Controller (SVIC) provides the board designer with an integrated, full-featured VME64 interface. This 64-pin device can be programmed to handle every transaction defined in the VME64 specification. The CY7C960 contains all the circuitry needed to control large DRAM arrays and local I/O circuitry without the intervention of a local CPU. There are no registers to read or write, and no complex command blocks to be constructed in memory. The CY7C960 simply fetches its own configuration parameters during the power-on reset period.

After reset, the CY7C960 responds appropriately to VMEbus activity and controls local circuitry transparently. The CY7C960 controls a bridge between the VMEbus and the local DRAM and I/O. Once programmed, the CY7C960 provides activities such as DRAM refresh and local I/O handshaking in a manner that requires no additional local circuitry.

The VMEbus control signals are connected directly to the CY7C960. The VMEbus address and data signals are connected to companion address/data transceivers which are controlled by CY7C960. The CY7C964 VMEbus Interface Logic Circuit is an ideal companion device. The CY7C964 provides 8 bits of data and address logic that has been optimized for VME64 transactions. In addition to providing the specified drive strength and timing for VME64 transactions, the CY7C964 contains all of the circuitry needed to multiplex the address/data bus for multiplexed VMEbus transactions. It contains counters and latches needed during BLT (Block Transfer) operations. It also contains address comparators which can be used in the board's Slave Address Decoder. For a 6U or 9U application, four CY7C964 devices are controlled by a single CY7C960. For 3U applications, the CY7C960 controls two CY7C964 devices and an address latch.

The design of the CY7C960 makes it unnecessary to know the details of the VMEbus transaction timing and protocol. The complex VMEbus activities are

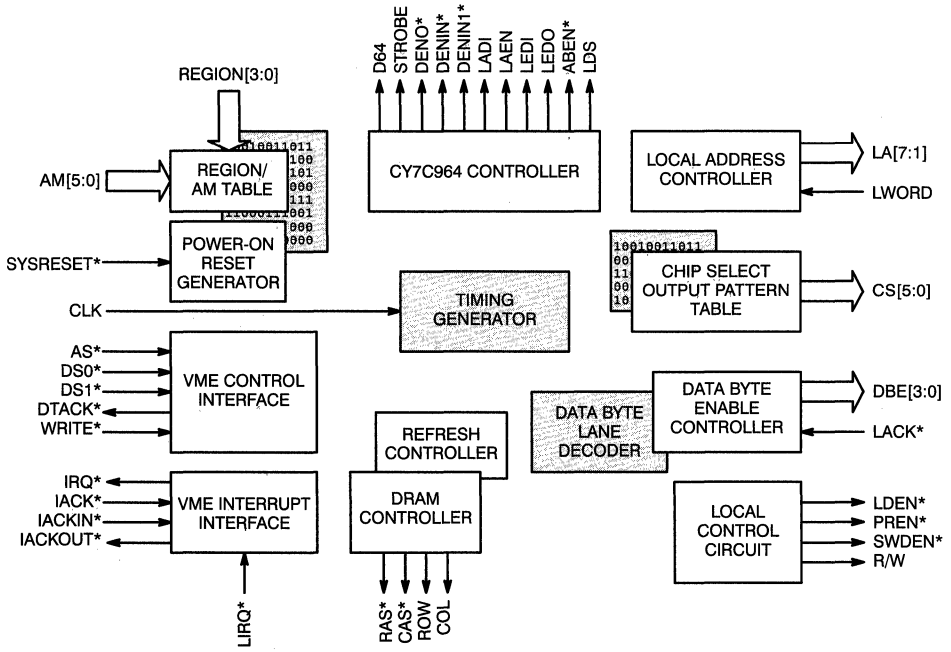


Figure 1. Internal Block Diagram of the CY7C960

translated by the CY7C960 to be simple local cycles involving a few familiar control signals. Similarly, it is not necessary to understand the operation of the companion device, the CY7C964; all control sequences for the part are generated automatically by the CY7C960 in response to VMEbus or local activity. If more information is desired, consult the CY7C964 chapter in the VIC64 Design Notes (available separately).

VMEbus transactions supported by CY7C960 include D8, D16, D32 (including UAT), MD32, D64, A16, A24, A32, A40, A64 single cycle and block transfer reads and writes, Read-Modify-Write cycles (including multiplexed), and Address-only (with or without Handshake). The CY7C960 functions as a VMEbus Interrupter, and supports the new Auto Slot ID standard and CR/CSR space. The CY7C960 also handles LOCK cycles, although full LOCK support is not possible within the constraints of the CY7C960 pinout. (full LOCK support is included in the CY7C961).

On the local side, no CPU is needed to program the CY7C960 nor to manage transactions. All programmable parameters are initialized through the use of either the VMEbus or a serial PROM. As the CY7C960 incorporates a reliable power-on reset circuit, parameters are self-loaded by the device at power-up or after a system reset. If the VMEbus is used to provide parameters, a VMEbus Master provides the programming information using a protocol that is compliant with the Auto Slot ID protocol from the new VME64 specification.

The architecture of the SVIC includes several functions that remove most of the VMEbus problems from the board designer's shoulders. All VMEbus control and response is automatic; the user loads the Region/AM table during configuration, and the CY7C960 then handles all appropriate VMEbus transactions. The CY7C964 controller works in lock step with the VMEbus Control Interface, providing the correct timing and control for the transaction in process. Local circuitry such as DRAM or I/O is simplified by the Refresh Controller, the DRAM Controller, and the Output Pattern Table. Block transfers are supported by the Local Address Controller together with the CY7C964 circuitry. Local timing is determined during configuration, and handshaking is available from the Data Byte Enable Controller. Local Interrupts are supported through the VME Interrupt Interface. The CY7C960 contains an internal Power-on Reset circuit, and also responds to a VMEbus SYSRESET*.

General Overview

Figure 2 illustrates a block diagram of a slave-only VME interface using one CY7C960/961 and four CY7C964s. No external glue logic is required when using the SVIC. The SVIC directly drives up to 6 Chip Selects (CSs) and four Data Byte Enables (DBEs) for interfacing to local resources. Depending on the requirements of your design, there may be a need for some external logic to implement a SWAP buffer, DRAM address interface, interrupt generation, and/or REGION decoding. The extent of this external logic would consist mainly of buffers (244s and 245s) and a PLD. The amount and complexity of external logic required is scalable depending on the requirements of your design. This application note concentrates on the design of these external logic components and on the interconnection of these components to the SVIC. The reference design for this application note is the SVIC Evaluation Board. All of the design examples will be in reference to the SVIC Evaluation Board including both discrete component usage and VHDL code.

Design Issues

DRAM Interface

The SVIC can be programmed (through the use of the WINSVIC software, as explained in the SVIC Users Guide) to operate in one of two modes: DRAM/IO or I/O Only. While in DRAM/IO mode the SVIC is capable of controlling a bank of DRAM through the use of RAS* (Row Address Strobe) and CAS* (Column Address Strobe) signals along with performing DRAM refresh (programmable timings). In order to speed up the access to DRAM, every time the AS* (Address Strobe) goes LOW on the VMEbus, the RAS* signal goes LOW on the SVIC causing the row address to be pre-latched into the DRAM. If the cycle was not meant for the DRAM then no harm was done, since a RAS-only cycle does not cause any reading or writing from/to the

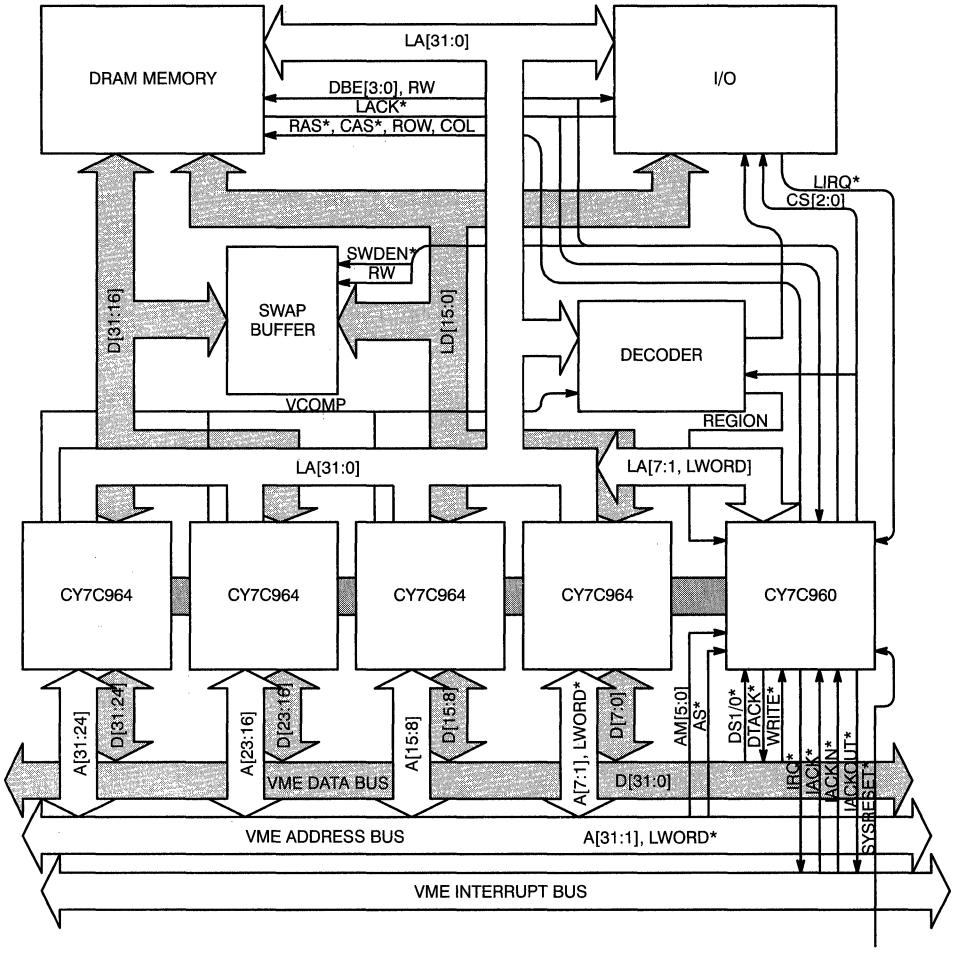


Figure 2. Block Diagram of Slave VME Board using SVIC

DRAM. But if the cycle is meant for the DRAM then half of the DRAM access has already occurred with only the CAS part of the cycle remaining.

Due to the fact that the address passes through the CY7C964s and not the SVIC itself, external buffers (244s) are required to separate the row and column address from the full address. Enabling these 244s at the proper time is accomplished by the ROW and COL outputs from the SVIC. An example of how our SVIC Evaluation Board implements this is illustrated in the next section entitled Design Examples.

Another important issue to deal with is distinguishing DRAM accesses from I/O accesses (when DBE[3:0] are used as CAS*). If the DBEs (Data Byte Enables) are programmed to act as CAS*, an assertion of DBE due to an I/O access will look like an assertion of CAS* to the DRAM, and will thus complete a RAS-CAS DRAM access. A solution to this issue is to gate a Chip Select from the SVIC with the DBEs to determine when the CAS* input on the DRAM should be driven LOW. An example of how our SVIC Evaluation Board accomplishes this can

be found in the Design Examples section that follows.

Swap Buffer

Most modern designs utilize memories that are 32 bits wide. The VME64 Specification allows for transactions that are 8, 16, 32, and 64 bits wide, which require reads and writes to resources in 8-, 16- and 32-bit-wide slices that may or may not be aligned to word boundaries. If 8- or 16-bit-wide transactions to 32-bit-wide local resources are to be allowed on your board, a Swap Buffer, comprised of 245s and controlled by the SVIC, needs to be included in the design of the slave board. If transactions are to be limited to the size of the local data size (i.e. only D32 to 32-bit-wide local data or only D16 to 16-bit-wide local data) the Swap Buffer can be omitted and the local data bus can be tied directly to the CY7C964s. Our SVIC Evaluation Board utilizes a Swap Buffer for performing 8-, 16-, 32-, and 64-bit transactions to 32-bit-wide memory. An example of how to implement the Swap Buffer can be found in the Design Examples section that follows.

Region Decoder

One of the most flexible features of the SVIC is the ability to react differently depending on where in the slave board's local address map a VME transaction is destined. Think of the local address map as being logically broken up into blocks of space referred to as regions. The local address map can be broken up into as many regions (up to 16) as required by your design. The size of each region is completely arbitrary and each region need not be of the same size. For example, 4 MBs of DRAM may sit in one region while 32K of SRAM may sit in another. The SVIC is told which region of the local memory map is being addressed based on what value is being asserted onto the REGION inputs.

The SVIC has four REGION inputs when in I/O Mode and three REGION inputs when in DRAM Mode. The value that is asserted on the REGION inputs is the job of the Region Decoder. The most common method used to determine which REGION value should be asserted to the SVIC is VME address decoding.

A comparison between the VME address that is placed on the VMEbus by the Master, and the VME address space in which the Slave board sits (Slave Base Address) will determine if the current VME transaction is destined for this particular Slave board. If the SVIC is to handle one and only one set of VME transactions (i.e., always A16 and A24 transactions), a comparison of the VME address and the Slave Base Address will be all that is required when deciding which REGION value to assert. In this example, a 'true' from the comparison logic will indicate that it is this board that is being addressed and that the region that has been programmed to allow A16 and A24 transactions should be asserted to the SVIC's REGION inputs.

If the SVIC is required to react differently when accessing different local resources, i.e. A16 (but not A24 or A32) transactions when addressing SRAM space and A16 and A32 (but not A24) transactions to DRAM space, the fact that it is this board being addressed is not enough to determine which REGION value to assert to the SVIC since the SVIC is required to react differently depending on which part of SVIC local address map is being addressed. In this case, further VME address decoding must be done by the Region Decoder to determine which region of the SVIC board is being addressed.

During initialization the SVIC is loaded with its configuration parameters. The configuration parameters are chosen using a free, Cypress-supplied software called WINSVIC. The WINSVIC software allows you to choose the configuration that is applicable to your design and outputs a file consisting of your chosen parameters encoded into 380 bits. These 380 bits are fed into the SVIC during initialization to fully configure the device. These configuration parameters consist of global parameters (those parameters that define the general operation of the chip) and Region parameters (those that define what type of VME transactions that the SVIC is allowed to handle and which Chip Selects will be driven if the current VME transaction is handled by the SVIC).

The SVIC is loaded with 16 sets of Region parameters when in I/O Mode and 8 sets of Region parameters when in DRAM Mode. Out of these many sets of Region parameters only one set is valid and being

used to define the operation of the SVIC at any one time. Which set of Region parameters that the SVIC should consider valid is determined by the user through the use of the REGION inputs (i.e., placing 3H on the REGION inputs will tell the SVIC to use the Region number 3 parameters when deciding if the current VME transaction should be handled).

The role that the Region parameters play in determining the operation of the SVIC is as follows:

1. Master places VME address, VME data (if a write), Address Modifier Codes (AM Codes), and strobes onto the VMEbus.
2. SVIC sees the strobes, waits a programmed period of time (known as the Decode Delay) and samples the REGION inputs.

   At this time the SVIC knows what type of VME transactions it will respond to.
3. SVIC looks at the AM Codes on the VMEbus (which define what type of transaction the Master is requesting) and compares the type of transaction requested with the types of transactions that it is allowed to handle (based on Region parameters).
4. If there was a match between requested and allowed transactions, the SVIC will drive the programmed Chip Selects (CS) and will handle the requested transaction. If there was not a match the SVIC would ignore this VME transaction.

Because the REGION inputs are driven by local logic, the determination of which region is being addressed at any given time is determined by the designer of the Region Decoder. The purpose of the Region Decoder is to determine if the address on the VMEbus falls into the address map of the SVIC. The address map of the SVIC can consist of up to 16 different regions, each of which can be of different sizes.

Figure 3 is an example of how a VME address can be mapped into regions. The first thing to note is that at least one region must not exist in the local address map. In this example, Regions 0 and 3 and Regions 7 thru 15 do not exist in the local address map. The SVIC should be programmed to ignore all AM codes when the REGION inputs are being driven with 0 or 3 or 7 thru 15. When the VME address does not fall within the Slave board's address space, it is one of these unused or 'turned-off' regions that should be asserted to the SVIC.

Another thing to notice is how the address map is decoded into regions. This example assumes that the SVIC is being addressed when the most significant byte (A[31:24]) of the address is FF (Slave Base Address = FFxxxxxx). The next nibble (A[23:20]) determines what region is being addressed and the rest of the address (A[19:0]) is decoded as the offset within the region. This address decoding scheme assumes 32-bit addresses. Because VME addresses can be of varying sizes, a design that would allow accesses in different address modes (A16, A32, etc.) will need to be aware of what address mode is being used for each transaction. Because this information is encoded in the AM Codes, the easiest thing to do

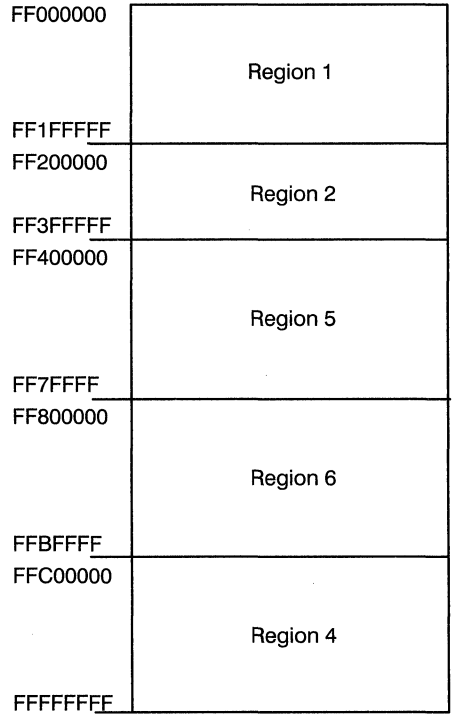


Figure 3. Example of an SVIC Address Map

is to include the VMEbus AM Code along with the address when decoding the region.

As this address map illustrates, regions need not be of the same size. The regions do not need to be in numerical order nor do all the regions need to appear in the address map.

Local Interrupts

The SVIC has one interrupt request pin (LIRQ*) available to local resources. Assertion of the LIRQ* pin by local resources causes a VME interrupt to occur. Upon acknowledgement of the VME interrupt by a master, through the use of the IACK daisy chain, the SVIC informs the local logic to place a Status/ID word onto the local data bus. This Status/ID word is read by the responding master and the interrupt acknowledge sequence is complete.

If more than one interrupter exists on the local side of the SVIC each interrupter must share the LIRQ* pin but can drive a different Status/ID word. It is the Status/ID word that truly distinguishes one interrupter from another. If more than one interrupt is pending at the same time it is up to local logic to perform interrupt priority. The complexity and size of the local interrupt logic is a function of the number of interrupters on the local side and the priority algorithm being implemented.

A64/A40 Support

The SVIC is capable of performing transactions in A64 and A40 address space. A64 addresses are transmitted over the VMEbus by multiplexing the 32-bit address and the 32-bit data buses that are available to 6U and larger VME cards. A40 addresses are transmitted over the VMEbus by multiplexing the 24-bit address and the 16-bit data buses that are available to 3U and larger VME cards. To support A64/A40 BLTs, the upper bits of the address (which are carried on the data bus) must be latched into external buffers for use in later cycles. The address is latched into and driven out of these latches (373s) at the proper time by signals that are sourced by the SVIC. If the SVIC is not programmed to handle A64 or A40 transactions then these external latches can be omitted from the design.

CY7C964 Interface

CY7C964s are directly controlled by the SVIC for use as the address and data glue logic between the VME and Local buses. The actual interconnections between the SVIC and the four CY7C964s is documented in the next section (Design Examples).

MD32 Support

Additionally, the VME64 Specification supports 32-bit-wide data transfers on 3U VME cards known as MD32 transactions. 3U VME cards only have a 16-bit data bus and a 24-bit address bus available to them. In order to transfer 32 bits of data at a time, the two buses are multiplexed with two bytes of data carried on the data bus and the other two bytes of data being carried on the address bus. Additions to a design for support of MD32 transactions include the control of the upper two CY7C964's DENIN* and DENIN1* (Data Enable In) inputs. The DENIN* and DENIN1* pins on 964–2 and 964–3 should be connected to the modified DENIN signals (MOD_DENIN* and MOD_DENIN1*, respectively, see *Figure 4*) and are only required if D64 transactions are to be supported on the same board.

Design Examples

DRAM Interface

Figure 5 illustrates how an SVIC can be interfaced to a bank of DRAM. The SVIC Evaluation Board uses a 4-MB 70-ns SIMM as the DRAM bank. This 4-MB SIMM requires ten bits of address and uses a 32-bit (4-byte) data word. The SIMM also has a separate CAS* (which is generated by the FLASH375) and RAS* for each data byte. The FLASH375 filters out DBE[3:0] assertions due to I/O access and allows DBE[3:0] assertions meant for the DRAM to be passed out to the CAS[3:0] lines.

Three buffers (244s) are used for separating the row and column address from the local address. Enabling of the row and column address buffers is accomplished by the SVIC by the assertion of ROW and COL. The latching of the address into the DRAM is controlled by the SVIC with the RAS* and CAS* signals.

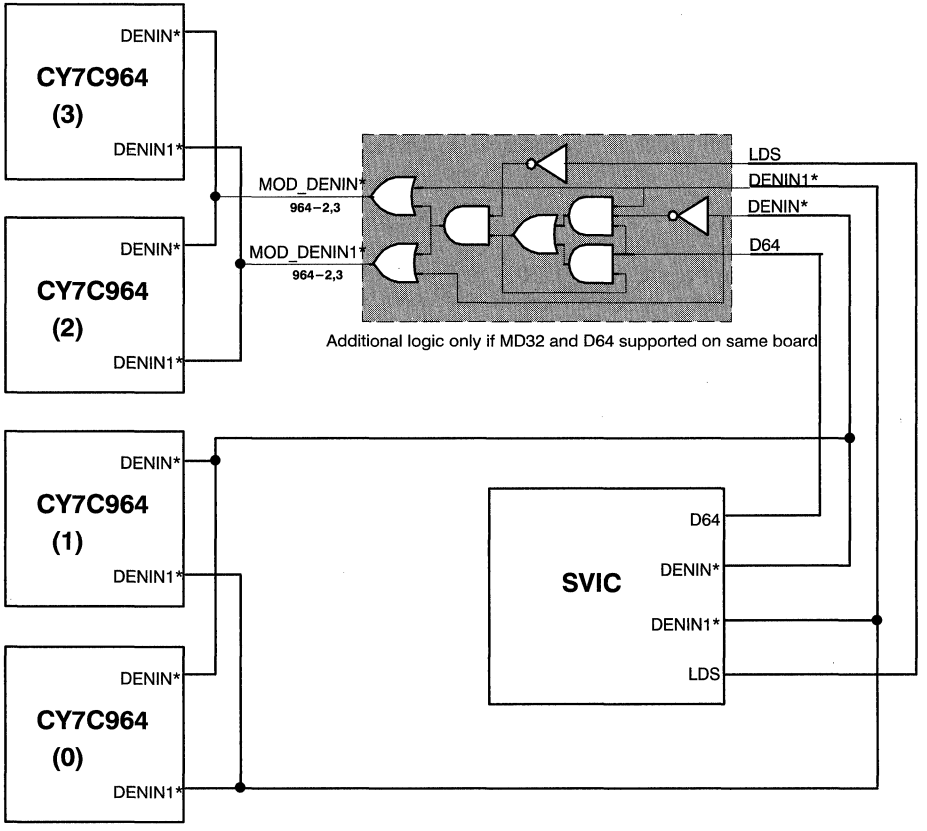


Figure 4. Additional Logic for MD32 Support

Swap Buffer

Figure 6 shows the implementation of the Swap Buffer on the SVIC Evaluation Board. The Swap Buffer is simply two '245 transcievers with the DIR and EN* control lines connected directly to the SVIC. The purpose of the Swap Buffer is to place LD[31:16] onto the LD[15:0] lines, and vice versa, for performing D16 transactions to 32-bit local resources.

Region Decoder

The Region Decoder for this SVIC Evaluation Board is designed to take full advantage of the CY7C960/961. Each of the 16 possible regions can be individually addressed regardless of the VME address space (A64, A40, A32, A24, and A16) being used. Because of the amount of logic and I/O pins used in the SVIC Evaluation Board Region Decoder, it was decided to write the decoder in VHDL (see Appendix A) and program it into a FLASH375 PLD. A simple diagram showing the inputs and outputs to our Region Decoder can be seen in *Figure 7.* The Region Decoder itself would fit into a smaller PLD but since several other parts of the Evaluation Board design were placed into a PLD (such as the Interrupt Logic) the FLASH375 was used due to the need for many I/O pins (especially for 32 bits of address and 32 bits of data). Most Region Decoders should require no more than 15–20 I/O pins and 50–100 gates.

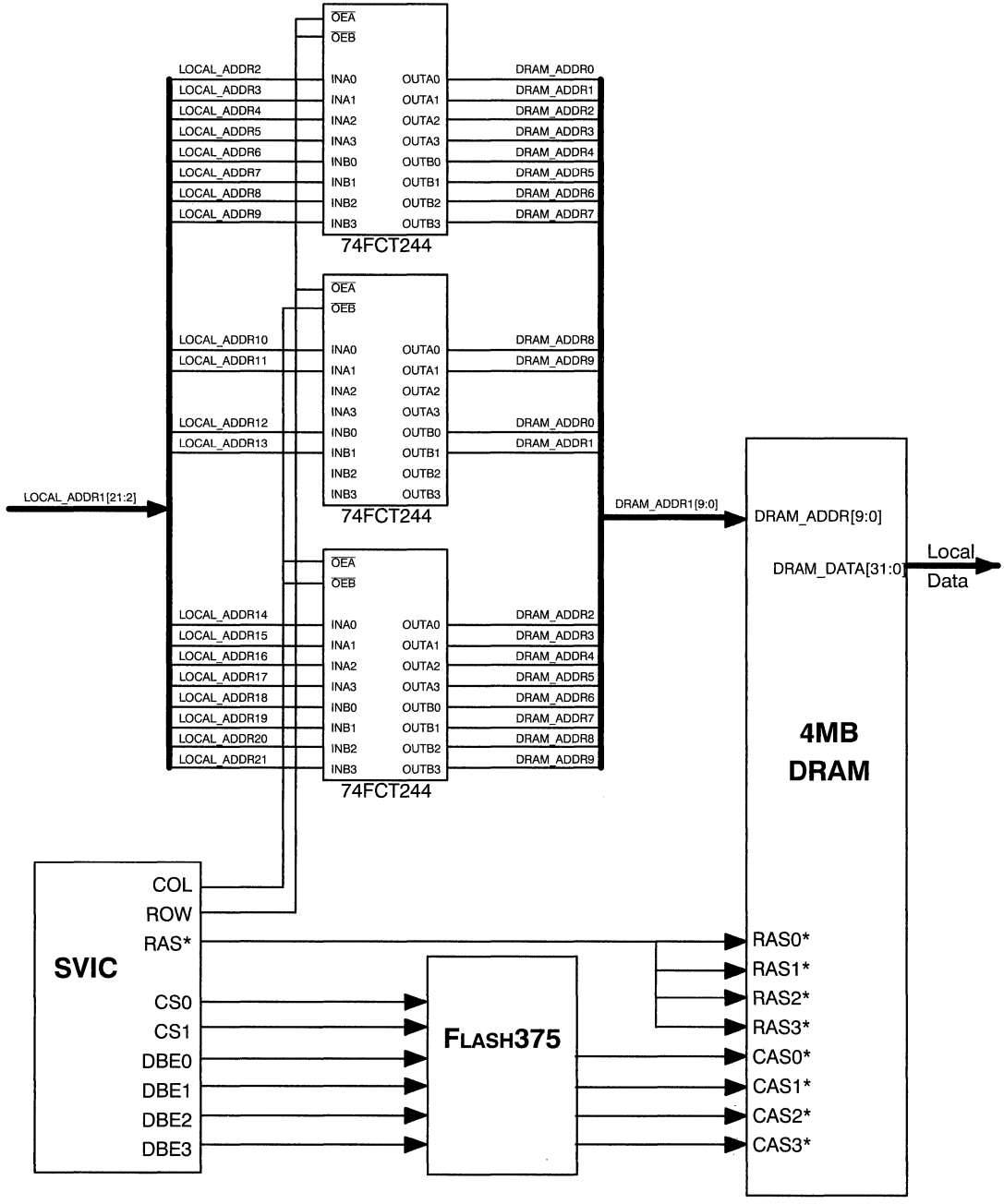


Figure 5. DRAM Interface Logic Example

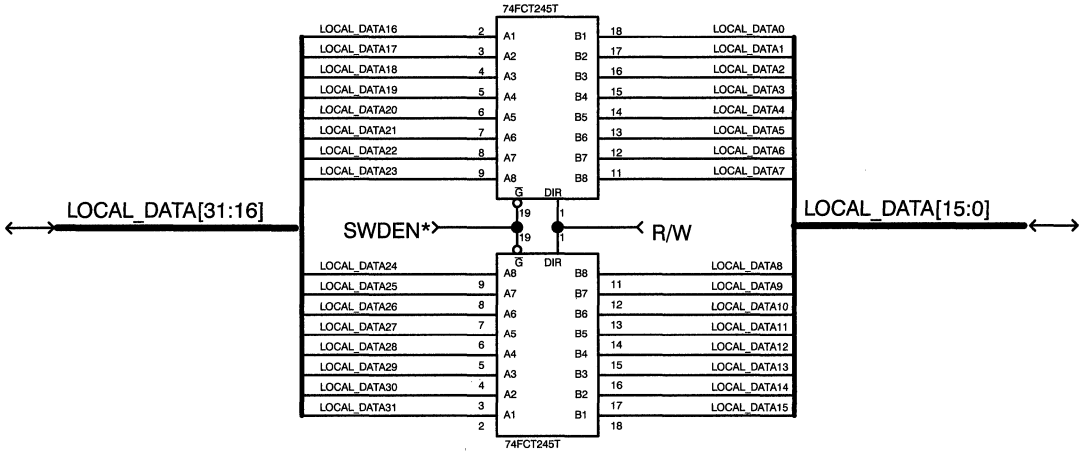


Figure 6. SWAP Buffer Implementation Example

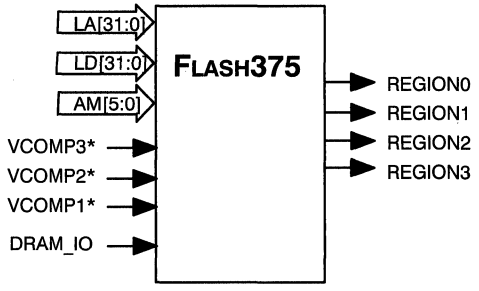


Figure 7. Inputs and Outputs of the Region Decoder Logic

We mapped the SVIC Evaluation Board into the VME address space as follows: the four most significant bits of the VME address are decoded to determine if it is this board that is being addressed. If it is this board that is being addressed then the next four significant bits are decoded as the region.

The challenge is to determine which are the most significant address bits. For example, in A32 space the most significant address bits start at A[31] but in A40 space the most significant bits start at D[15]. The only way to know which address space is being used by the VME Master that is initiating the transaction is to decode the AM Codes from the VME-bus. The AM Codes tell where the most significant VME address bits lie and the address bits tell which region is being addressed (if any).

The Region Decoder VHDL code begins with a CASE statement that uses AM Codes to determine which addressing mode is being used by the VME Master. The use of all 6 bits of the AM Code in the CASE statement was for ease of reading and not by necessity. All that would be required to determine the addressing mode is the three most significant bits of the AM Code.

Once the addressing mode is determined (i.e., the location of the most significant address bits is found) it can be determined if it is this board that is being addressed. Performing an address comparison on the four most significant address bits determines this. For A64 and A40 transfers the address bits themselves must be looked at, but for A32, A24, and A16 transfers the CY7C964s can be used to perform the comparison.

Each CY7C964 performs a comparison between the 8 bits of VME address that it is attached to and a Compare Address and Mask value that are written into each CY7C964 during configuration. A comparison between the 8 bits of VME address and the Compare Address (w/Mask) will result in the

VCOMP output from the CY7C964 being driven LOW (see the VIC64/CY7C964 Design Notes).

If the comparison produces a match it must be determined which region is being addressed. For many designers this may be a fixed region that will require no further decoding of the address. The SVIC Evaluation Board allows all 16 regions to be addressed by a VME Master by driving the second most significant nibble of the address onto the REGION inputs. The driving of the REGION3 input of the SVIC is controlled by an input to the Region Decoder on the SVIC Evaluation Board called DRAM_IO. This functionality was included to allow the Evaluation Board to function in both DRAM/IO Mode (3 REGION inputs) and I/O Mode (4 REGION inputs) depending on how the SVIC is programmed. Most slave boards will operate in only one mode, depending on what resources have been designed onto the board, so it will be known how many REGION inputs must be driven by the decoder thus eliminating the need for the DRAM_IO input function.

Local Interrupts

The VHDL Code located in Appendix A contains the code used on the SVIC Evaluation Board for the Interrupt Logic. *Figure 8* shows the inputs and outputs to the Local Interrupt Logic. The Evaluation Board is capable of generating VME interrupts from four different local sources, each with its own Status/ID word. The Interrupt Logic VHDL Code also handles AUTO ID and the Compare and Mask loading of the CY7C964s.

LIRQ* (Local Interrupt Request) will be driven LOW when one or more of the LIRQi* inputs on the FLASH375 are driven LOW. When LDEN* (Local Data Enable) is driven LOW and MWB* (Module Wants Bus) is HIGH, a value must be driven onto the Local Data (LD) bus. The value that must be driven onto the LD bus will either be a Status/ID associated with a local interrupt, the STATUS/ID associated with VMEbus Initialization (AUTO ID) or the Compare and Mask for the CY7C964s.

The Local Interrupts have been assigned priority in the VHDL Code with LIRQ1* having the highest

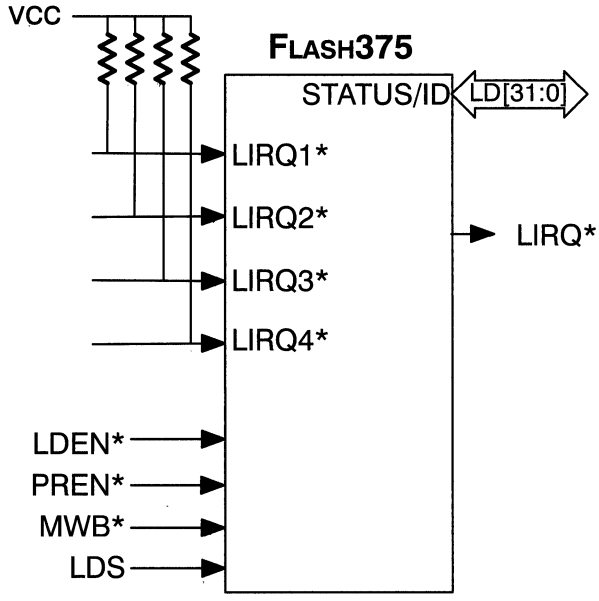


Figure 8. Inputs and Outputs to Local Interrupt Logic

priority and LIRQ4* having the lowest. *Table 1* is a summary of what is driven onto the Local Data bus when LDEN*=0.

Table 1. Summary of What Is Driven onto the Local Data Bus when LDEN*=0.

What is driven onto the Local Data bus	When it is driven onto the Local Data bus
Interrupt Status/ID	LDEN*=0, MWB*=1, PREN*=1, LIRQi*=0
AUTO ID Status/ID	LDEN*=0, MWB*=1, PREN*=1, LIRQi*=1
CY7C964 Compare	LDEN*=0, MWB*=1, PREN*=0, LDS=1
CY7C964 Mask	LDEN*=0, MWB*=1, PREN*=0, LDS=0

A64/A40 Support

The A64/A40 Support built into the SVIC Evaluation Board consists of latches ('573/'373s) on the Local Data (LD) bus for use in latching the address bits that are carried on the LD bus during multiplexed address cycles (see *Figure 9*). A40 support requires the latching of LD[15:0] while A64 support requires the latching of LD[31:0]. The control equations for latching and enabling (LA_UP_ADDR and EN_UP_ADDR) are located in the VHDL Code in Appendix A.

CY7C964 Interface

The SVIC Evaluation Board utilizes four CY7C964s to act as the bridge between the VMEbus and the local buses. The interconnections between the CY7C961 and the CY7C964s are summarized in *Table 3.* The table is organized with one row for each CY7C964 pin (or bus for A, D, LA, LD) and one column per each CY7C964 (964–0, 1, 2, 3). The last column of *Table 3* is for users of the CY7C960. An entry in this column should replace the entries in the other columns in that row when the CY7C960 is being used.

All signals are sourced from the SVIC unless the name of a source appears in parentheses under the signal name. For example, in the row below (*Table 2*): the LCIN* pin on the least significant CY7C964 (964–0) should be connected to VCC, the LCIN* on the next CY7C964 should be connected to GND, LCIN* on 964–2 should be connected to the LCOUT* pin on 964–1 and LCIN* on 964–3 should be connected to the LCOUT* pin on 964–2. Since the last column is empty there is no difference in the connections to the LCIN* pin when using the CY7C960 as apposed to using the CY7C961.

Table 2. Example Row from *Table 3*

CY7C964 Pin	964–0 LSB	964–1	964–2	964–3 MSB	If Using the CY7C960
LCIN*	VCC	GND	LCOUT* (964–1)	LCOUT* (964–2)	

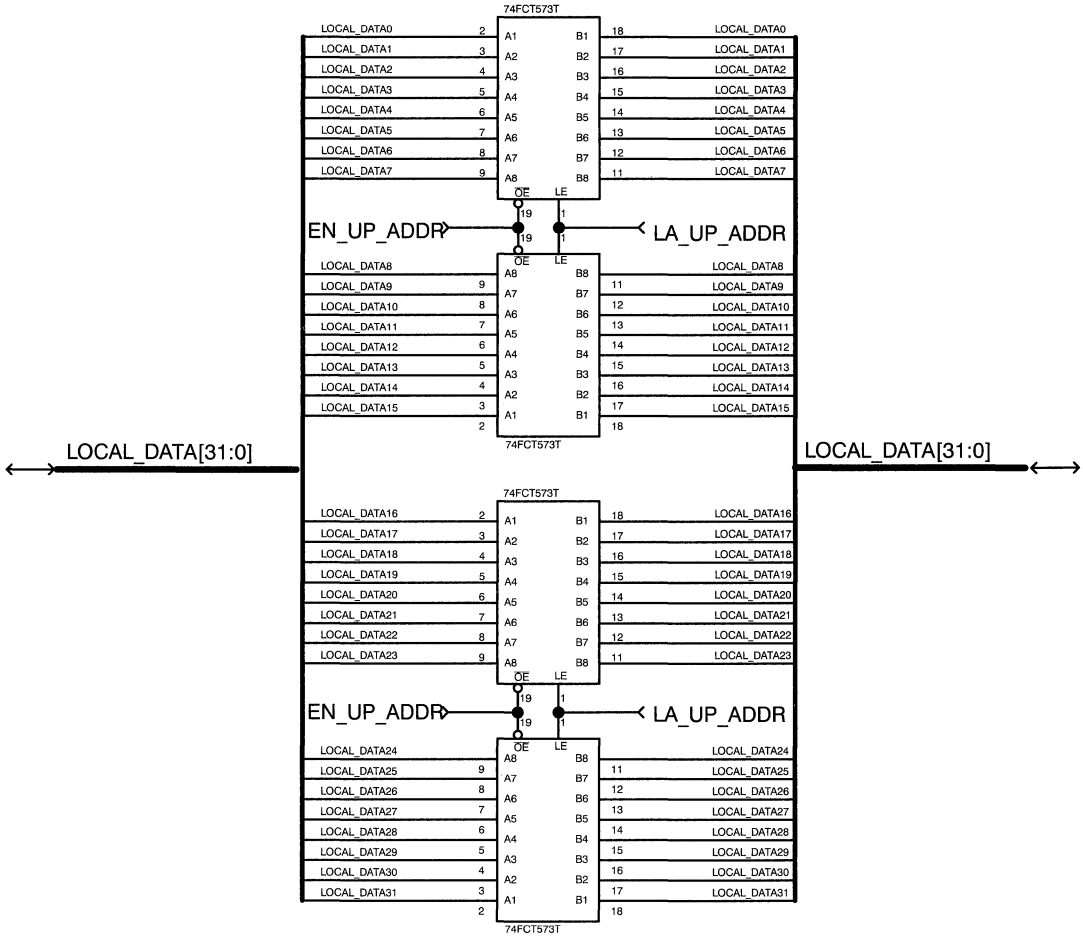


Figure 9. Additional Logic for A64/A40 Support

Table 3. Connections Between the SVIC and Four CY7C964s

CY7C964 Pin	964–0 LSB	964–1	964–2	964–3 MSB	If Using the CY7C960
A[7:0]	A[7:1],LWORD (VME)	A[15:8] (VME)	A[23:16] (VME)	A[31:24] (VME)	
D[7:0]	D[7:0] (VME)	D[15:8] (VME)	D[23:16] (VME)	D[31:24] (VME)	
LA[7:0]	LA[7:0] (LOCAL)	LA[15:8] (LOCAL)	LA[23:16] (LOCAL)	LA[31:24] (LOCAL)	
LD[7:0]	LD[7:0] (LOCAL)	LD[15:8] (LOCAL)	LD[23:16] (LOCAL)	LD[31:24] (LOCAL)	
ABEN*	ABEN*	ABEN*	ABEN*	ABEN*	
BLT*	BLT*	BLT*	BLT*	BLT*	VCC
D64	D64	D64	D64	D64	
DENIN*	DENIN*	DENIN*	DENIN1*	DENIN1*	
DENIN1*	DENIN1*	DENIN1*	DENIN*	DENIN*	
DENO*	DENO*	DENO*	DENO*	DENO*	
FC1	FC1	FC1	FC1	FC1	GND
LCOUT*	N/C	LCIN* (964–2)	LCIN* (964–3)	N/C	
LDS	LDS	LDS	LDS	LDS	
LADI	LADI	LADI	LADI	LADI	
LAEN	LAEN	LAEN321	LAEN321	LAEN321	VCC on 964–1,2,3 LAEN on 964–0
LEDI	LEDI	LEDI	LEDI	LEDI	
LEDO	LEDO	LEDO	LEDO	LEDO	
LADO	VMECNT	LADO	LADO	LADO	GND
LCIN*	VCC	GND	LCOUT* (964–1)	LCOUT* (964–2)	
MWB*	MWB*	MWB*	MWB*	MWB*	VCC
STROBE*	STROBE*	STROBE*	STROBE*	STROBE*	
VCOMP*	AS NEEDED	AS NEEDED	AS NEEDED	AS NEEDED	
VCIN*	GND	GND	VCOUT* (964–1)	VCOUT* (964–2)	
VCOUT*	N/C	VCIN* (964–2)	VCIN* (964–3)	N/C	

MD32 Support

MD32 Support on the SVIC Evaluation Board consists of creating modified DENIN*/DENIN1* (MOD_DENIN*/MOD_DENIN1*) signals for use in control of the two most significant CY7C964s. If MD32 and D64 transactions are to be supported on the same board, the entries in the CY7C964 Interface table for DENIN* and DENIN1* should be replaced with the entries in *Table 4*.

Table 4. Modified DENIN connections for MD32 Support

CY7C964 Pin	964–2	964–3
DENIN*	MOD_DENIN*	MOD_DENIN*
DENIN1*	MOD_DENIN1*	MOD_DENIN1*

Required Resistors

The following signals need pull-up or pull-down resistors:

PULL-UP: BLT*, MWB*, ABEN*, DENO*, PREN*

PULL-DOWN: LAEN

In addition, if the CY7C960 is being used, FC1 and LADO on the CY7C964s must be tied LOW.

Serial PROM

The SVIC needs to be configured at power-up. The configuration consists of approximately 380 bits of serial data into the part from either the VMEbus or through the use of a serial PROM from the local bus. There are several serial PROMs that are compatible with the SVIC: the AT&T ATT1718 and ATT1736, Xilinx XC1718, XC1736 and XC1765 and Atmel 'Configurator' AT17C65, AT17C128. The numbers following the 17 in each of the part numbers indicate the number of Kbits that the part holds. All of these PROMs have a programmable RESET/Output Enable (R/OE) pin, and the SVIC expects the RESET to be active HIGH. The RESET/OE on these PROMs are programmed to be active HIGH by writing ones into a special memory location. The memory location that must be written (with ones) varies by PROM size. The memory addresses are shown in *Table 5*.

Table 5. PROM Addresses

PROM Size	Address
18K	8DC–8DF
36K	11B8–11BB
65K	2000–2003
128K	4000–4003

Active HIGH Reset: fill address with ones

Figure 10 illustrates the connections between the SVIC and the serial PROM. The R/OE pin should be connected to the PREN* output of the SVIC. R/OE should also have a pull-up resistor to ensure that the internal pointer is reset to the first position. The Chip Enable (CE) pin can be either tied LOW or tied to the R/OE pin, the Clock (CLK) pin should

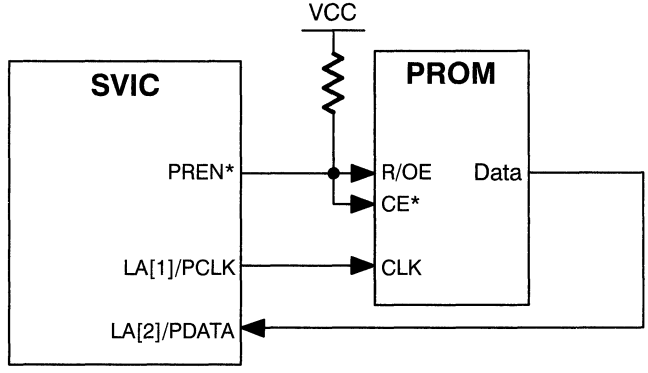


Figure 10. Connection of SVIC to Serial PROM

be connected to the LA[1]/PCLK pin of the SVIC and, finally, the Data (D) pin should be connected to the LA[2]/PDATA pin on the SVIC.

Summary

This application note has shown how easy it is to design a fully VME64-compliant Slave VME board using the Cypress Slave VME Interface Controller (SVIC) Family (CY7C960/961). Along with four CY7C964s (Bus Interface Chips), a PLD, and a small amount of TTL logic, a Slave VME board capable of D8 thru D64/A16 thru A64 transactions can easily be designed in a short amount of time.

Discrete components and VHDL code were used to design the little off-chip logic that was used on the SVIC Evaluation Board. Along with examples on how to interface the SVIC to the VMEbus, significant examples on how to interface the SVIC to DRAM and I/O were also discussed. The design of optional logic, such as the SWAP Buffer and Local Interrupt logic was explained for those boards requiring it.

A discussion of regions was included to help in the understanding of this topic. Also included for completeness was a discussion on which serial PROMS could be used and where resistors should be added.

The CY7C960 or CY7C961 along with four CY7C964s comprises the most complete and easy to design fully VME64-compliant Slave VME Interface on the market today.

Appendix A. VHDL Code

```
-- vhdl code for the SVIC EVAL Board

use work.GATESPKG.all;
use work.cypress.all;
use work.rtlpkg.all;



ENTITY logic IS
  PORT(D64, LDS, DENIN, PREN, DENIN1, LEDI, LDEN, MWB, LIRQ1, LIRQ2,
    LIRQ3, LIRQ4, DRAM_IO, CS0, CS1, DBE0, DBE1, DBE2, DBE3, RW, RESET,
    SYSRESET: IN BIT;
    MOD_DENIN, MOD_DENIN1, LA_UP_ADDR, EN_UP_ADDR, LIRQ, CE0, CE1, CE2,
    CE3, CAS0, CAS1, CAS2, CAS3, OE, SVIC_RESET: OUT BIT;
    SELECTLM: INOUT BIT;
    LA: IN x01z_VECTOR(31 downto 8);
    AM:  IN BIT_VECTOR(5 downto 0);
    VCOMP: IN BIT_VECTOR(3 downto 1);
    REGION:  OUT x01z_VECTOR(3 downto 0);
    LD:  INOUT x01z_VECTOR(31 downto 0));

  ATTRIBUTE PIN_NUMBERS OF logic : ENTITY IS
    "LD(0):2 LD(1):3 LD(2):4 LD(3):5 LD(4):6 LD(5):7 LD(6):8 LD(7):9 "
    &"LD(8):11 LD(9):12 LD(10):13 LD(11):14 LD(12):15 LD(13):16 LD(14):17
     LD(15):18 "
    &"LD(16):23 LD(17):24 LD(18):25 LD(19):26 LD(20):27 LD(21):28
     LD(22):29 LD(23):30 "
    &"LD(24):32 LD(25):33 LD(26):34 LD(27):35 LD(28):36 LD(29):37
     LD(30):38 LD(31):39 "
    &"LA(8):159 LA(9):158 LA(10):157 LA(11):156 LA(12):155 LA(13):154
     LA(14):153 LA(15):152 "
    &"LA(16):150 LA(17):149 LA(18):148 LA(19):147 LA(20):146 LA(21):145
     LA(22):144 LA(23):143 "
    &"LA(24):138 LA(25):137 LA(26):136 LA(27):135 LA(28):134 LA(29):133
     LA(30):132 LA(31):131 "
    &"AM(0):42 AM(1):43 AM(2):44 AM(3):45 AM(4):46 AM(5):47 "
    &"VCOMP(3):122 VCOMP(2):123 VCOMP(1):124 "
    &"REGION(0):113 "
    &"REGION(1):51 REGION(2):58 REGION(3):53 "
    &"LIRQ1:85 LIRQ2:84 LIRQ3:83 LIRQ4:82 "
    &"DENIN:119 DENIN1:118 MOD_DENIN:117 MOD_DENIN1:116 LA_UP_ADDR:115
     EN_UP_ADDR:114 "
    &"D64:139 LDS:129 PREN:72 LEDI:128 LDEN:127 MWB:126 SELECTLM:125 "
    &"LIRQ:75 DRAM_IO:98 CE0:89 CE1:88 CE2:87 CE3:86 DBE0:94 DBE1:93
     DBE2:92 DBE3:91, CS0:19 "
    &"RW:77 OE:78  SYSRESET:66 RESET:67 SVIC_RESET:68 CAS0:97 CAS1:96
     CAS2:95 CAS3:112";

END logic;
```

Appendix A. VHDL Code (continued)

```
ARCHITECTURE arch_logic OF logic IS



    signal VL1N18: bit;
    signal VL1N26: bit;
    signal VL1N28: bit;
    signal VL1N31: bit;
    signal VL1N36: bit;
    signal VL1N40: bit;


    signal STATUS_ID: BIT_VECTOR(31 downto 0) := X"FFFFFFFF";
    signal STATUS_EN: BIT  := '0';
    signal REGION_TEMP:  BIT;

--     for all: AND2 use entity work.AND2(archAND2);

--     for all: INV use entity work.INV(archINV);

--     for all: AND3 use entity work.AND3(archAND3);

--     for all: OR2 use entity work.OR2(archOR2);



begin
-------------------------------------------------------------------------------
-- This is the logic for SELECTLM (when writing the REMOTE MASTER
-- registers)
-------------------------------------------------------------------------------

  SELECTLM <= '0' WHEN ((FXB(LA(31)) = '1') AND (FXB(LA(30)) = '1') AND
    (FXB(LA(29)) = '0') AND (FXB(LA(28)) = '0')) ELSE '1';



-------------------------------------------------------------------------------
-- This is the logic for RESET
-------------------------------------------------------------------------------

  SVIC_RESET <= RESET AND SYSRESET;
```

Appendix A. VHDL Code (continued)

```
-------------------------------------------------------------------------------
-- This is the logic for driving the CASi inputs to DRAM
-- CASi is driven both during DRAM refresh and data access but not during
-- I/O access
-------------------------------------------------------------------------------

  CAS0 <= DBE3 OR (CS1 AND (NOT CS0));
  CAS1 <= DBE2 OR (CS1 AND (NOT CS0));
  CAS2 <= DBE1 OR (CS1 AND (NOT CS0));
  CAS3 <= DBE0 OR (CS1 AND (NOT CS0));

-------------------------------------------------------------------------------
-- This is the logic for the latch and enable signals for A40/A64 UPPER
-- ADDRESS
-------------------------------------------------------------------------------

  LA_UP_ADDR <= (NOT SELECTLM) AND LEDI;

  EN_UP_ADDR <= LDEN OR MWB;

-------------------------------------------------------------------------------
-- This is the logic for controlling the CE*, OE* signals to each bank of
-- SRAM in I/O space
-------------------------------------------------------------------------------


  CE0 <=  DBE3 OR CS0;
  CE1 <=  DBE2 OR CS0;
  CE2 <=  DBE1 OR CS0;
  CE3 <=  DBE0 OR CS0;

  OE <=   NOT RW;



-------------------------------------------------------------------------------
-- This is the cross-connected SWAP BUFFER logic required for MD32 and D64
-- on same board
-------------------------------------------------------------------------------

    VL1I1: AND2
        port map(A => D64,
                 B => VL1N31,
                 Q => VL1N28);

    VL1I11: INV
        port map(A => DENIN,
                 QN => VL1N18);
```

Appendix A. VHDL Code (continued)

```
    VL1I2: AND3
        port map(A => DENIN1,
                 B => VL1N18,
                 C => D64,
                 Q => VL1N26);

    VL1I3: OR2
        port map(A => VL1N26,
                 B => VL1N28,
                 Q => VL1N31);

    VL1I33: AND2
        port map(A => VL1N36,
                 B => VL1N31,
                 Q => VL1N40);

    VL1I38: OR2
        port map(A => DENIN1,
                 B => VL1N40,
                 Q => MOD_DENIN);

    VL1I39: OR2
        port map(A => VL1N40,
                 B => DENIN,
                 Q => MOD_DENIN1);

    VL1I4: INV
        port map(A => LDS,
                 QN => VL1N36);



-------------------------------------------------------------------------------
-- This is the REGION DECODER
-------------------------------------------------------------------------------

reqion:    PROCESS
     BEGIN
     CASE AM is
       WHEN "000100" | "000011" | "000001" | "000000" =>  --A64 AM Codes
         IF LD(31 downto 28) = "1110" THEN
           REGION(2 downto 0) <= LD(26 downto 24);
           REGION_TEMP <= FXB(LD(27));
         ELSE REGION(2 downto 0) <= "000";
           REGION_TEMP <= '0';
         END IF;
```

Appendix A. VHDL Code (continued)

```
WHEN "110100" | "110101" | "110111" =>   --A40 AM Codes
   IF LD(15 downto 12) = "1110" THEN
     REGION(2 downto 0) <= LD(10 downto 8);
     REGION_TEMP <= FXB(LD(11));
   ELSE    REGION(2 downto 0) <= "000";
      REGION_TEMP <= '0';
   END IF;

WHEN "001000" | "001001" | "001010" | "001011" | "001100" | "001101"
   | "001110" | "001111" =>  --A32 AM Codes
   IF VCOMP(3) = '0' THEN
     REGION(2 downto 0) <= LA(26 downto 24);
     REGION_TEMP <= FXB(LA(27));
   ELSE    REGION(2 downto 0) <= "000";
     REGION_TEMP <= '0';
   END IF;

WHEN "101111" | "110010" | "111000" | "111001" | "111010" | "111011"
   | "111100" | "111101" | "111110" | "111111" =>  --A24 AM Codes
   IF VCOMP(2) = '0' THEN
     REGION(2 downto 0) <= LA(18 downto 16);
     REGION_TEMP <= FXB(LA(19));
   ELSE    REGION(2 downto 0) <= "000";
      REGION_TEMP <= '0';
   END IF;

WHEN "101001" | "101100" | "101101" =>  --A16 AM Codes
   IF VCOMP(1) = '0' THEN
     REGION(2 downto 0) <= LA(10 downto 8);
     REGION_TEMP <= FXB(LA(11));
   ELSE    REGION(2 downto 0) <= "000";
     REGION_TEMP <= '0';
   END IF;

WHEN "011000" | "011001" | "011010" | "011011" | "011100" | "011101"
   | "011110" | "011111" =>  --USER1 AM Codes
   IF VCOMP(3) = '0' THEN   --A32 Modes
     REGION(2 downto 0) <= "101";  --FORCED TO REGION 5
     REGION_TEMP <= '0';
   ELSE    REGION(2 downto 0) <= "000";
     REGION_TEMP <= '0';
   END IF;
```

Appendix A. VHDL Code (continued)

```
      WHEN "010000" | "010001" | "010010" | "010011" | "010100" | "010101"
        | "010110" | "010111" =>  --USER2 AM Codes
        IF VCOMP(2) = '0' THEN
          REGION(2 downto 0) <= "010";  --FORCED TO REGION 10
          REGION_TEMP <= '1';
        ELSE    REGION(2 downto 0) <= "000";
          REGION_TEMP <= '0';
        END IF;

      WHEN OTHERS =>  --DEFAULT REGION
        REGION(2 downto 0) <= "000";
        REGION_TEMP <= '0';

    END CASE;
    END PROCESS;

    region_buffer: triout PORT MAP(REGION_TEMP, DRAM_IO, REGION(3));
--DON'T DRIVE REGION(3) WHEN IN DRAM MODE (DRAM_IO = 0)

-------------------------------------------------------------------------------
-- This is the INTERRUPT LOGIC
-------------------------------------------------------------------------------

    LIRQ <= (LIRQ1 AND LIRQ2) AND (LIRQ3 AND LIRQ4);
    STATUS_EN <= (NOT LDEN) AND MWB;

b1:   FOR i IN 0 TO 31 GENERATE
        bx: triout PORT MAP(STATUS_ID(i), STATUS_EN, LD(i));
      END GENERATE;

interrupt: PROCESS
    BEGIN
    IF LDEN = '0' THEN
     IF (LIRQ1 = '0' AND PREN ='1') THEN STATUS_ID(7 downto 0) <= X"01";
     ELSIF (LIRQ2 = '0' AND PREN = '1') THEN STATUS_ID(7 downto 0) <=
      X"02";
     ELSIF (LIRQ3 = '0' AND PREN = '1') THEN STATUS_ID(7 downto 0) <=
      X"03";
     ELSIF (LIRQ4 = '0' AND PREN = '1') THEN STATUS_ID(7 downto 0) <=
      X"04";
     ELSIF (PREN = '1') THEN STATUS_ID(7 downto 0) <="01010101";
     ELSIF (PREN = '0' AND LDS = '1') THEN STATUS_ID <= X"EEEEEE00";--compare
     ELSIF (PREN = '0' AND LDS = '0') THEN STATUS_ID <= X"0F0F0FFF"; --mask
     END IF;
    END IF;
    END PROCESS;

end arch_logic;
```

Using the CY7C964 with VIC

The CY7C964 is a flexible collection of byte-wide (8-bit) transceivers, latches, counters, multiplexers, and comparators that provide VMEbus interface designs with a low-cost alternative to PLDS, ASICs, or discrete logic devices. It is based on a standard cell design that incorporates patented line drivers for reduced ground bounce and high-noise immunity.

The CY7C964 is a companion part to the Cypress VIC068A and VIC64 VMEbus Interface Controller devices. It is compatible with all operating modes of either device, including dual address path, block transfers, block transfer initialization cycles, local DMA control, and D64 64-bit VMEbus block transfers (when used in conjunction with the VIC64). Signal naming conventions correspond directly to the VIC068A/VIC64 buffer control signals and the CY7C964 can be directly connected to these signals. The device can also be used as a generic interface building block. CY7C964s are cascadable allowing easy interfacing to any width bus. By combining multiple logic functions into one discrete part, the CY7C964 saves board space and reduces power consumption.

The CY7C964 has two main operating modes, byte width and word width. The byte-width configuration of the device has 8 local address, 8 local data, 8 VMEbus address, and 8 VMEbus data signals. In the byte-width modes, two methods are available for loading the VMEbus master block transfer address counters. This counter is loaded using the nominal VIC block transfer initiation cycle, or alternatively from the local data bus. Loading the VMEbus master block transfer counter from the local data bus decouples the board's local address map from that of the VMEbus. This allows boards that consume a great deal of local address space to still be able to view the entire VMEbus address region. More information on both of the VMEbus master block transfer address counter initialization techniques is provided in the following sections.

In word-width mode the local and VMEbus data signals change to 16-bit address or data paths. This mode expands all of the features of the device to 16 bits in width, with the exception of the D64 multiplexer. This multiplexer is disabled in word-width mode. Since protocols such as block transfer initiation cycles remain compatible in word-width mode, the device becomes useful as a 16-bit bus interface block for non-VMEbus applications.

CY7C964 Features

- Directly connects to VIC068A or VIC64
- Internal counters for block transfers
- Internal multiplexers for D64 block transfers
- Internal comparators for address decoding
- Supports VIC068A/VIC64 dual address path option
- Supports cascadable operation
- Directly drives VMEbus address and data
- Directly drives local address and data bus signals
- Reduces components for compact board design
- Low power requirements
- Available in 64-pin QFP

CY7C964 Block Diagrams

The CY7C964 is an array of optimally controlled counters, comparators, general registers, and multiplexers. A small amount of state logic is also present within the device. This state logic monitors bus cycles that are issued to the device and places the component in the appropriate mode. The state logic is implemented as an asynchronous rather than a synchronous sequential state machine. These hidden internal-state or configuration bits are set and cleared automatically by monitoring the arrival times of various input signals.

The configuration bits, and other input signals to the device, select the operating mode (byte width or word width), as well as the initialization method for the internal VMEbus master and local block transfer counters. The block diagrams in *Figures 1, 2,* and *3,* show an equivalent internal representation of the device for each operational mode.

Byte-Width Mode I

In this mode, the CY7C964 operates as a conventional byte-width slice of VIC compatible interface logic. The device conforms to the standard VIC block transfer initiation cycle and includes multiplexers for D64 block transfer operations and comparison logic for VMEbus address decoding.

Counters C1, C2, and C3, latch L8, and multiplexers S3 and S5 form the core of the block transfer address generation logic. C1 is the local master block transfer address counter, C2 is the VMEbus slave block transfer address counter, and C3 is the VMEbus

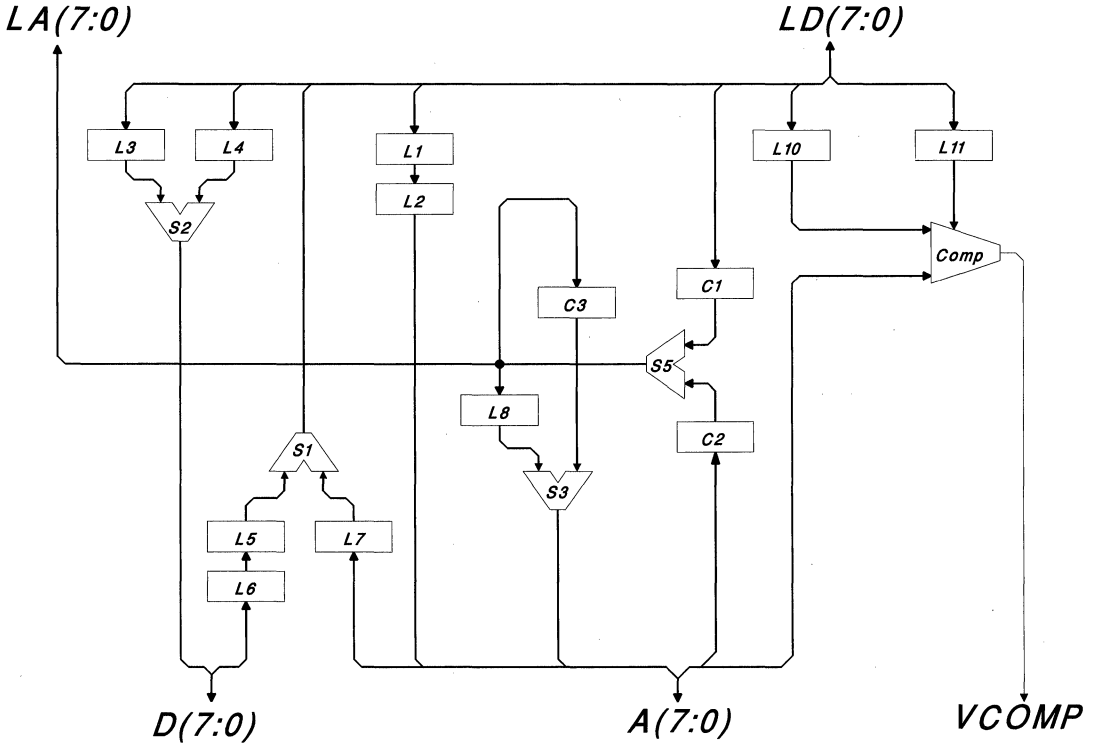


Figure 1. CY7C964 Byte-Width Mode I Block Diagram

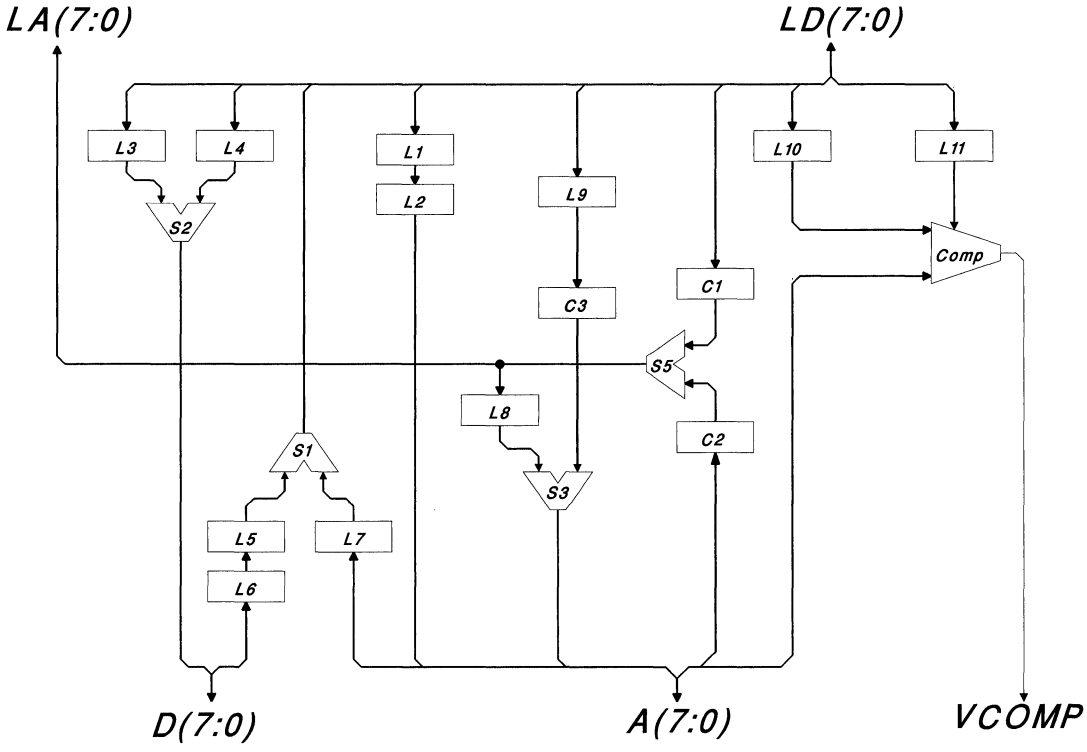


Figure 2. CY7C964 Byte-Width Mode II Block Diagram

master block transfer address counter. As shown in the block diagram, counter C1 loads from the local data bus LD[7:0], C3 loads from the local address bus LA[7:0], and counter C2 loads from the VMEbus address bus A[7:0]. Multiplexer S5 selects the source for the local address either through C2 (which is also used for single cycle operations) or C1. Latch L8 and multiplexer S3 provide the support for the Dual Address Path feature. Single cycle VMEbus master transfers can occur using L8 during the interleave periods of master block transfers without corrupting the contents of C3.

Latches L10, L11, and comparator COMP form the VMEbus address comparison logic. L11 is the Address Mask register that enables and disables bits of the address comparator. L10 is the Address Comparison register which contains an 8-bit value that is matched against A[7:0]. When the enabled bits of L10 match the corresponding signals of A[7:0], the VCOMP* output is asserted (Low). Writing 1's to all bits of L11 disables the comparison logic. In this case all values of A[7:0] match, causing the VCOMP output to be asserted continuously. Loading comparison register L10 clears and enables all bits of the mask register L11. Therefore, during system initialization, comparison register L10 must be loaded first, then bits can be disabled within mask register L11.

Latches L1, L2, L3, and L4, and multiplexer S2 combine to form a high performance D64 block transfer data pipeline and multiplexer. During D64 block transfer operations data is fetched from local memory and transferred to latch L1. A second memory fetch is required to assemble the 64-bit word. This data is stored in L3. When L3 is updated, the data in L1 is moved to L2. This allows the VIC

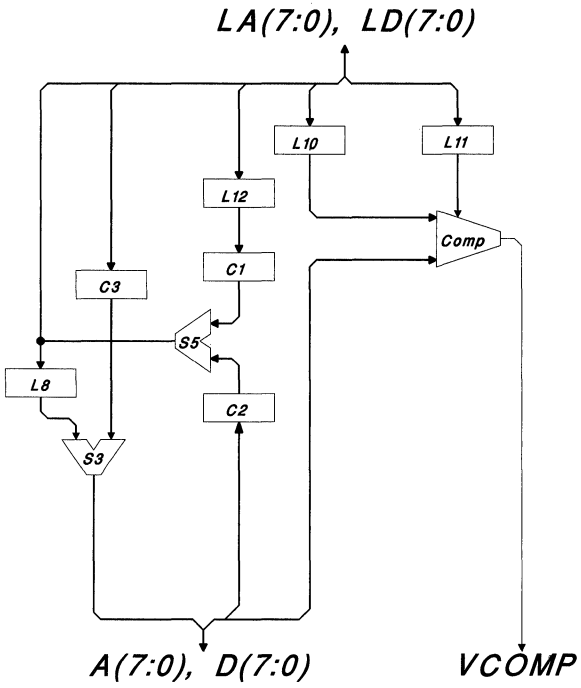


Figure 3. CY7C964 Word Width Mode Block Diagram

to prefetch the next word of information from local memory while the VMEbus D64 data transfer operation is in progress.

Latches L5, L6, L7 and multiplexer S1 form the VMEbus D64 block transfer data demultiplexer. Data is latched from the VMEbus into latches L6 and L7, simultaneously. The data is then moved to the local data bus through multiplexer S1.

This is the nominal operating mode of the CY7C964. Control signals connect to the corresponding buffer control signal on the VIC, with the exception of the DENIN* and DENIN1* inputs. Refer to the following section, Interfacing to the VIC64 and VIC068A, and to the *VIC64/CY7C964 Design Notes* for additional information on DENIN*, DENIN1*, and other control signals.

Byte Width Mode II

This mode is nearly identical to the previous byte-width mode with one exception. The VMEbus master block transfer counter, C3, loads from the local data bus LD[7:0] rather than the local address bus LA[7:0]. The main benefit of this operating mode is that the entire VMEbus address space is available for data transfers. Loading C3 from the local address bus may preclude some addresses from the VMEbus, because they are being decoded locally. For example if EPROM is located at address 0x00000000, this address may not be accessible across the VMEbus. Using the CY7C964 in this mode requires performing one additional bus cycle during the block transfer initiation.

The CY7C964 operates in byte-width mode if the BLT* and MWB* input signals on the CY7C964 are swapped. In other words, the MWB* signal on the

VIC connects to the BLT* input on the associated CY7C964s. The same rule applies for the BLT* output on the VIC. It connects to the MWB* input on the CY7C964s. The CY7C964 monitors the arrival time of these two signals and expects to load the master block transfer counter from the local data bus LD(7:0) if BLT* is asserted prior to MWB*. Swapping these two signals does not change the operation of any other feature on the device, however, there are two things to consider when using this mode.

The address decode signal that drives MWB* on the CY7C964 connects to the BLT* output on the VIC. This signal should be driven with an open collector or three-state device to allow the VIC to control the signal during block transfer operations.

Block transfer initiation cycles also change. The VMEbus master block transfer address counter loads from the local data bus one cycle before the actual block transfer initiation cycle. The subsequent cycle is a typical block transfer initiation cycle with the local data bus containing the local DMA address. The local address bus is ignored by the CY7C964s during this cycle, but not by the VIC. The low-order byte of the local address bus LA[7:0] must contain the correct VMEbus master block transfer address. This is necessary because the VIC cannot be programmed to load the VMEbus master block transfer address from the local data bus. For more information on Byte Width Mode II refer to the *VIC64/CY7C964 Design Notes*.

Word-Width Mode

The second main operating mode of the CY7C964 is the word-width mode. This mode of the device works well for VMEbus address control functions. All of the address related functions (local master block transfer counter, C1, VMEbus slave block transfer counter, C2, VMEbus master block transfer counter C3, and the address comparison logic) expand to 16 bits. The address and data buses on the part combine to form two 16-bit buses. A high-drive-strength, 16-bit bus and a medium-drive-strength bus are formed from A[7:0], D[7:0] and LA[7:0], LD[7:0] respectively. D[7:0] and LD[7:0] are the least significant sections of each of these buses.

In this mode one additional latch (L12), is located between the local address bus and local master block transfer counter, C1. This latch allows the local master block transfer counter to be loaded from the local address bus prior to loading the VMEbus master block transfer counter, C3. This is necessary since both the VMEbus master block transfer counter, C3, and the local master block transfer counter, C1, are loaded from the same local address bus. When counter C3 loads during the block transfer initiation cycle, the contents of latch L12 are moved to counter, C1.

All other functions available in this mode operate in a similar manner as in the byte width mode. For further information on this mode and detailed timing information refer to the *VIC64/CY7C964 Design Notes*.

Interfacing to the VIC64 and VIC068A

Previously, interfacing the VIC068A to the VMEbus required a significant amount of LSI and MSI devices. With the advent of 64-bit VMEbus block transfers and the VIC64, the external discrete device count for a full functional interface implementation expanded. The CY7C964 has been developed to combat this problem by incorporating the functions of much of this external logic into a single package. Use of the CY7C964 shortens system design, debug, and manufacturing cycle times. This removes the burden of performing worst-case and critical timing analysis on the VMEbus and VIC buffer control sections of a system design. Local control signals other than those directly connected to the VIC64 or VIC068A have been kept to a minimum. *Figure 4* shows a full function D64 VMEbus interface implemented using the CY7C964, VIC64 and all VMEbus interface local support logic. This example interface features:

- Dual Path Address Operation
- Slave BLT Cycles During Master BLT Interleave
- Software Programmable Slave VMEbus Address
- Write Posting
- VIC Mail Box Interrupt Messaging Support

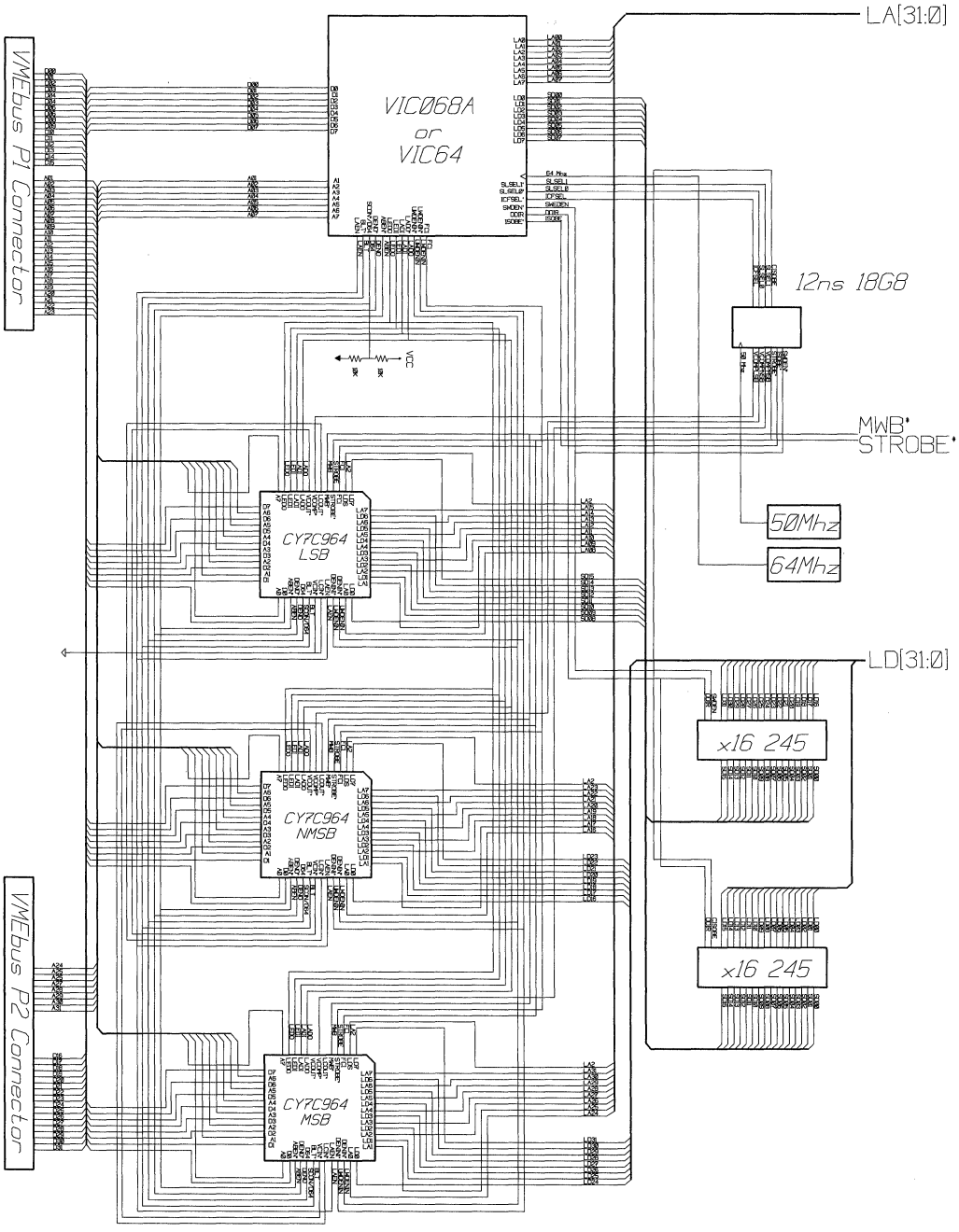


Figure 4. CY7C964/VIC VMEbus Interface

The interface can be dissected into 5 functional sections for the purpose of discussion. These sections are:

- VMEbus Signal Group
- VIC Buffer Control Signal Group
- CY7C964 Local Signal Group
- CY7C964 Address Comparison Group
- Local Data Swap Buffer Logic

The focus of this application note is the CY7C964. Each of the interface functional sections are examined from this perspective. The CY7C964s are referred to as the LSB (Least Significant Byte), NMSB (Next Most Significant Byte), and MSB (Most Significant Byte) device depending on the segment of the VMEbus that they control. The LSB controls VMEbus address and data signals [15:8], NMSB [23:16], and MSB [31:24]. This interface uses the CY7C964s in the byte width mode I as address and data controllers. All of the information contained within this section pertains to this mode of operation. For additional information on the signals described within this section consult The *VMEbus Specification (IEEE 1014)* and/or the *Cypress Semiconductor VIC068A/VAC068A User's Guide*

VMEbus Signal Group

This group includes the VMEbus address and data signals.

D[7:0]. VMEbus compatible data signals that directly connect to VMEbus P1 and P2 connectors.

A[7:0]. VMEbus compatible address signals that directly connect to VMEbus P1 and P2 connectors.

Each CY7C964 provides support for 8 bits of VMEbus address and data. Three CY7C964s are necessary for 32-bit (D32/D64) interface applications. The A[7:0] and D[7:0] transceivers on the CY7C964 furnish a high drive strength allowing direct connection to the respective address and data signals on the VMEbus backplane. With the VIC068A or VIC64 controlling the CY7C964s, all VMEbus worst case timing and drive strength requirements are met for all types of data transfer operations.

VIC Buffer Control Signal Group

This group includes all of the VIC buffer control signals.

LADO. Latch Address Out, directly connects to VIC LADO on all CY7C964s.

LADI. Latch Address In, directly connects to VIC LADI on all CY7C964s.

LEDO. Latch Enable Data Out, directly connects to VIC LEDO on all CY7C964s.

LEDI. Latch Enable Data In, directly connects to VIC LEDI on all CY7C964s.

ABEN*. VMEbus Address Bus Enable, directly connects to VIC ABEN on all CY7C964s.

DENO. Data Enable Output, directly connects to VIC DENO on all CY7C964s.

D64. D64 Block Transfer Mode Enable, directly connects to VIC64 SCON/D64 pin on all CY7C964s. This input should be tied Low on all CY7C964s when using VIC068A.

BLT*. Block Transfer Enable, directly connects to VIC BLT on all CY7C964s.

LAEN. Local Address Enable, directly connects to VIC LAEN on all CY7C964s.

DENIN*. Primary Data Enable In, directly connects to VIC UWDENIN* on NMSB and MSB CY7C964s, and directly connects to VIC LWDENIN* on LSB CY7C964.

DENIN1*. Companion Data Enable In, directly connects to VIC LWDENIN* on NMSB and MSB CY7C964s, and directly connects to VIC UWDENIN* on LSB CY7C964.

A major design-time savings is realized when using the CY7C964s because all of these signals directly connect to the VIC or are hardwired to a steady state value. The buffer control interface is simple and straight forward, with the minor exception that the connection of UWDENIN* and LWDENIN* control signals from the VIC are swapped to the DENIN* and DENIN1* on the LSB CY7C964.

CY7C964 Local Signal Group

The CY7C964 local signal group consists of the VMEbus and local block transfer counter count-enable daisy-chains.

LCIN*. Local address counter Count enable IN. On the LSB CY7C964 tie this input Low. On the NMSB device directly connect this signal to the LCOUT* of LSB device. For the MSB CY7C964 connect this input to the LCOUT* of the NMSB device.

LCOUT*. Local address counter Count enable OUT. On the LSB CY7C964, connect this output to the NMSB LCIN* input. On the NMSB CY7C964, connect this output to the MSB LCIN* input. For the MSB device do not connect this output.

VCIN*. VMEbus address counter Count enable IN. On the LSB CY7C964 tie this input Low. On the NMSB device directly connect this input to the VCOUT* of LSB device. For the MSB CY7C964 connect this input to the VCOUT* of the NMSB device.

VCOUT*. VMEbus address counter Count enable OUT. On the LSB CY7C964, connect this output to the NMSB VCIN* input. On the NMSB CY7C964, connect this output to the MSB VCIN* input. For the MSB device, do not connect this output.

These signals enable the local and VMEbus higher order address counters, two local address counters, a master block transfer, a slave block transfer, and a single VMEbus address counter. The local address counters share the LCIN*/LCOUT* count enable daisy-chain. These signals are multiplexed within the CY7C964 and enable the proper counter depending on the current state of the interface. The VCIN*/VCOUT* daisy-chain is dedicated to the VMEbus address counter on the device. When the VCIN* or LCIN* inputs are held Low, counting is enabled for the appropriate counters within the device. The VCIN* and LCIN* signals do not advance the counters; these signals just enable counting. The counters increment when these signals are active and the proper increment count control logic sequence occurs. The VIC advances the address counters at the proper time during VMEbus and local DMA block transfer operations.

CY7C964 Address Comparison Signal Group

The CY7C964 address comparison signal groups consists of the local signals that are associated with the internal VMEbus address comparators. FC1, MWB*, and LDS also are used to control other functions on the CY7C964. Refer to the *VIC64/CY7C964 Design Notes* for more information on these signals.

FC1. Function Code 1 signal, directly connects on all CY7C964s to the local signal that drives the VIC FC1.

MWB*. Module Wants VMEbus, directly connects on all CY7C964s to the local signal that drives the VIC MWB*.

LDS. Load Register Select, directly connects to LA2 for systems with 32-bit local bus. Refer to the following text for additional information.

STROBE*. Latch Register Control. A chip select like signal that selects the CY7C964 internal comparator mask and comparison registers.

VCOMP*. VMEbus Address Comparator Output. The comparator output signal from the CY7C964 address comparator. This signal asserts Low if the a pattern on address signals A[7:0] matches the programmed values of the comparison logic.

The implementation of this group of CY7C964 signals is application specific. The MWB* and FC1 signals are included in this section because they are locally generated signals required by the VIC. These two signals differ slightly from their companions on the VIC. On the CY7C964 the MWB* and FC1 signals are inputs. On the VIC, MWB* is also an input, but FC1 is a bidirectional signal that can be driven by the VIC. These signals on the CY7C964 can be directly connected to their respective local signals on VIC.

The CY7C964s contain a high-performance programmable VMEbus address equality comparator. This comparator is controlled by two internal write-only registers: a mask and a compare register. The mask register enables and disables bits of the

comparator and the compare register stores the data pattern which inputs are compared against. VCOMP* is the active Low comparator match output signal. VCOMP* is driven Low by the CY7C964 if the bit pattern on A[7:0] matches the associated enabled bits of the compare register. Loading the mask register bits with 0's enables the corresponding bits of the compare register. Loading bits of the mask register with 1s places bits of the compare register in a don't care or match-on-anything state. If all bits of the mask register are loaded with 1s, the compare register matches everything, causing VCOMP* to always be driven Low. These registers are loaded by supplying the proper data on LD(7:0) and address on MWB* and LDS signals. The STROBE* input is used to qualify the address and latch the data into the proper internal register. *Figures 5*and *6* and *Table 1* show the waveforms and timing conditions needed to load the compare and mask registers. The mask register is cleared (all bits enabled), when the compare register is loaded.

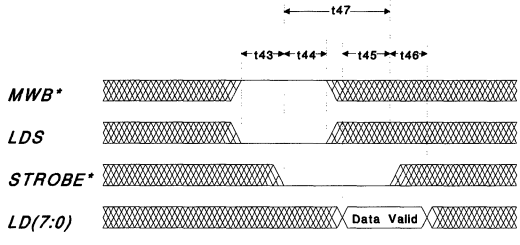


Figure 5. Mask Register Load Cycle

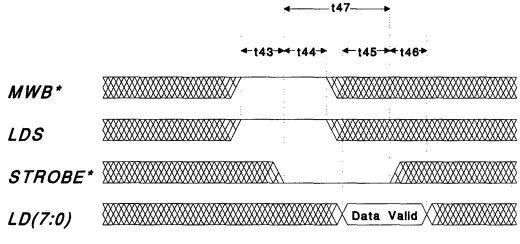


Figure 6. Compare Register Load Cycle

Table 1. Compare Register Load Cycle Times

Para	Description	Min.	Max.	Unit
t43	MWB*, LDS set-up time to STROBE* falling edge	0		ns
t44	MWB*, LDS hold time after STROBE* falling edge	5		ns
t45	LD(7:0) set-up time to STROBE* rising edge	5		ns
t46	LD(7:0) hold time after STROBE* rising edge	5		ns
t47	STROBE* minimum pulse width	10		ns

Therefore it is important to load the compare register first, then load the mask register as desired.

This load cycle operates as follows. The state of LDS and MWB* are latched on the falling edge of STROBE*. The data is loaded into the selected register on the rising edge of STROBE*. MWB* must be held inactive (High) during comparator register loading. The state of LDS selects the register to load. If LDS is High during the cycle the compare register is loaded; if LDS is Low the data is written to the mask register. This load cycle can be generated by decoding a separate address region or chip select signal for the CY7C964 comparator registers.

For applications with a 32-bit local data bus it is desirable to load all three CY7C964s in parallel by having the host processor perform a 32-bit write cycle to the address region that activates STROBE*. The select signal for the address region is connected to the STROBE* input on all three CY7C964s. The 8 bits of data on the lowest order section of the local data bus LD[7:0] do not matter to the VIC, as long as the VIC CS* signal remains inactive during this write cycle. Boards that use this style of interface should connect LDS to LA2, thereby decoding the mask register at the Base Address of the address region and the compare register at the Base Address + 4. LDS also controls the operation of the D64 block transfer data multiplexer/demultiplexer. Systems with 32-bit local

data buses should connect local address signal LA2 to LDS for proper operation of the D64 data multiplexer/demultiplexer logic.

The mask and compare registers can be set to select any contiguous address region on the VMEbus. These registers do not preload and can power up in any state. It is advisable to initialize these registers as soon as possible in the system boot sequence. The CY7C964 comparator output signal VCOMP*, supplies the result from the equality compare logic. VCOMP* drives Low when the input matches the loaded comparator conditions.

The CY7C964 VCOMP* signal is not directly compatible with the VIC SLSEL0* and SLSEL1* slave select signals. The short (10 ns) address setup time to AS* active for VMEbus slave boards does not meet the worst case compare out delay of the CY7C964 VCOMP* signal. Combining this with the potential output glitching that can occur with an asynchronous comparator can cause problems for the VIC. It is recommended that the VCOMP* signal be externally filtered prior to being used with the VIC SLSEL0* or SLSEL1* signals. Most applications will require some external comparison logic to combine the VCOMP* signals from the NMSB and MSB devices, furnishing finer grained VMEbus decoding.

The interface example in *Figure 4* uses a 12-ns 18G8 to perform these functions and to disable the VMEbus slave select signals to the VIC until the CY7C964 comparator control registers have been initialized. Using this PLD allows the interface to decode three different VMEbus addresses regions:

- VMEbus A32 for local access – VIC SLSEL0*
- VMEbus A24 for local access – VIC SLSEL1*
- VMEbus A16 for mailbox interrupt – VIC ICFSEL*

Figure 7 shows the PLD ToolKit design file for this device. The two VIC slave select signals (SLSEL0* and SLSEL1*) can be used to conveniently decode two VMEbus address regions. SLSEL0* selects if the NMSB CY7C964 becomes TRUE. SLSEL1* requires both NMSB and MSB comparators to evaluate TRUE.

A 50-MHz clock and the D registers within the 18G8 are used to build a simple digital filter that removes any glitches that may occur on the three CY7C964 VCOMP* signals.

As mentioned previously, the comparators within the CY7C964s are always active and they power up in an unknown state. The PLD includes an ENABLE signal which disables the SLSEL0*, SLSEL1*, and ICFSEL* signals to the VIC until the first access is made to one of the comparator control registers. Adding the ENABLE function to this PLD guarantees that the VIC slave select signals cannot become active until the one of the comparator control registers has been initialized.

There is a potential problem that can occur when loading the CY7C964 comparator control registers. The local data bus isolation buffer, which is necessary to allow data swapping, is normally disabled by the VIC. This causes a problem during CY7C964 register initialization cycles because the VIC only enables the local data bus isolation buffers during VIC or VMEbus accesses. The PLD solves this problem by providing conditioning logic for the ISOBE* signal. In the PLD design file in *Figure 7,* a signal named CISOBE* is generated. CISOBE* asserts (Low) enabling the isolation buffer if the VIC ISOBE* is asserted or the CY7C964 STROBE* input is asserted. SWDEN* was added to the equation for completeness, however, it may not be necessary in many designs. There are obviously many other implementations for control of the VIC isolation buffers. One implementation is shown here, but the best method for control of this signal is application specific and left to the designer.

Local Data Swap Buffer Logic

Local Data Swap Buffer logic is a requirement for all 32-bit local bus designs that want to be able to perform 8- or 16-bit transfers. The swap buffer moves data to and from the lower section of the VMEbus D[15:0] to the upper segments of the local bus D[31:16]. VMEbus requires that all 8- and 16-bit data transfers be performed on the D[15:0] section of the bus. The CY7C964s work properly with the VIC controlled swap buffer.

```
C18G8;

CONFIGURE;

CLK_50Mhz(node=1),
LSBCOMP(node=2),
NSBCOMP(node=3),
MSBCOMP(node=4),
STROBE(node=5),
BD_RESET(node=6),
ISOBE(node=7),
SWDEN(node=8),

ENABLE(node=12, noreg, iop),
SEL(node=13),
DSEL(node=14),
CISOBE(node=15, noreg, iop),
SLSEL1(node=17, noreg, iop),
SLSEL0(node=18, noreg, iop),
ICFSEL(node=19, noreg, iop),

EQUATIONS;

/CISOBE     =    <OE>
                 <SUM> /ISOBE
                 +     /STROBE *  SWDEN;

/ENABLE     =    <OE>
                 <SUM>  BD_RESET * /STROBE
                 +      BD_RESET * /ENABLE;

/SEL        =    <OE>
                 <SUM>  /LSBCOMP * /ENABLE *  BD_RESET
                 +      /NSBCOMP * /ENABLE *  BD_RESET
                 +      /MSBCOMP * /ENABLE *  BD_RESET;

/DSEL       =    <OE>
                 <SUM>  /SEL * /LSBCOMP * /ENABLE *  BD_RESET
                 +      /SEL * /NSBCOMP * /ENABLE *  BD_RESET
                 +      /SEL * /MSBCOMP * /ENABLE *  BD_RESET;

/ICFSEL     =    <OE>
                 <SUM>  /DSEL * /LSBCOMP * /ENABLE * BD_RESET;

/SLSEL0     =    <OE>
                 <SUM>  /DSEL * /NSBCOMP * /ENABLE * BD_RESET;

/SLSEL1     =    <OE>
                 <SUM>  /DSEL * /MSBCOMP * /NSBCOMP * /ENABLE *  BD_RESET;
```

Figure 7. PLD ToolKit™ Design File For 18G8 PLD

Summary

The CY7C964 is a high-performance byte or word width slice of VIC compatible VMEbus logic. Using the CY7C964 in conjunction with the VIC068A or VIC64 shortens design cycle time, reduces component count, reduces interface real estate requirements, and overall design risk. For further information on the local VIC interface and more detailed timing information in the CY7C964 refer to the *VIC068A/VAC068A User's Guide* and the *VIC64/CY7C964 Design Notes.*

Features of the VIC068A VMEbus Interface Controller

This application note describes some features of the Cypress VIC068A and provides information on how to use the device.

The VIC068A was designed by a consortium of VMEbus manufacturers in partnership with Cypress. The major goals of this consortium were to achieve a standardized, reasonably priced VMEbus interface that was not dominated by any board manufacturer. Manufacturing this specialized chip requires a high-speed process (125 MHz) and high-power I/O pins (64 and 48 mA).

The VIC068A adheres to the ANSI/IEEE Standard 1014, which minimizes the problems of interfacing among the VMEbus boards of various manufacturers. A block diagram detailing the device's functional blocks appears in *Figure 1*.

VIC068A Highlights

With very precise timing, based on a 64-MHz clock that is used internally to make decisions on 8-ns intervals, you can reach the theoretical limits of the VMEbus transfer rates—a block transfer rate of 40 Mbytes/s.

Because all logic resides in a single chip, the VIC068A greatly reduces the board space necessary to interface to the VMEbus. Even a highly sophisticated interface with an A32/D32 system controller and block transfer support requires no more than 60 square cm or 20 percent of a double eurocard (6U card).

Special care has been taken to speed up the VIC068A's VMEbus access. Although many of today's CPU boards use megabytes of high-speed local RAM to limit the number of VMEbus accesses, the accesses that do occur for I/O or data reads and writes must be done efficiently to avoid slowing the rest of the system.

For both types of data transfers, the VIC068A offers special support. For single-write cycles, you can program the VIC068A to operate in the so-called master or slave write-posting mode (*Figure 2*), the local VMEbus write cycle is terminated locally as soon as data is latched in the VMEbus latches. This allows the local CPU to continue with instruction fetches or other operations while the VIC068A transfers data over the VMEbus.

In slave write-posting mode (*Figure 3*), the same function happens with write cycles form the VMEbus to the local bus. As soon as the data is latched, the VMEbus cycle is terminated and the local cycle can finish independently of further VMEbus traffic. Both modes reduce CPU overhead and VMEbus utilization, providing higher bandwidth in single-cycle writes.

The VMEbus prohibits a similar function in single-cycle reads because every read cycle on the VMEbus could turn out to be a read-modify-write (RMW) cycle. This cannot be foreseen because the only difference is that address strobe is held low between the two cycles. Therefore, if the VMEbus address strobe were released during the two portions of the same RMW cycle, another VMEbus master could break into that cycle and modify the same data.

To move blocks of data over the VMEbus, the VIC068A uses the block transfer mode. In its standard form, this mode allows a processor to transfer up to 256 bytes with just one starting address sup-

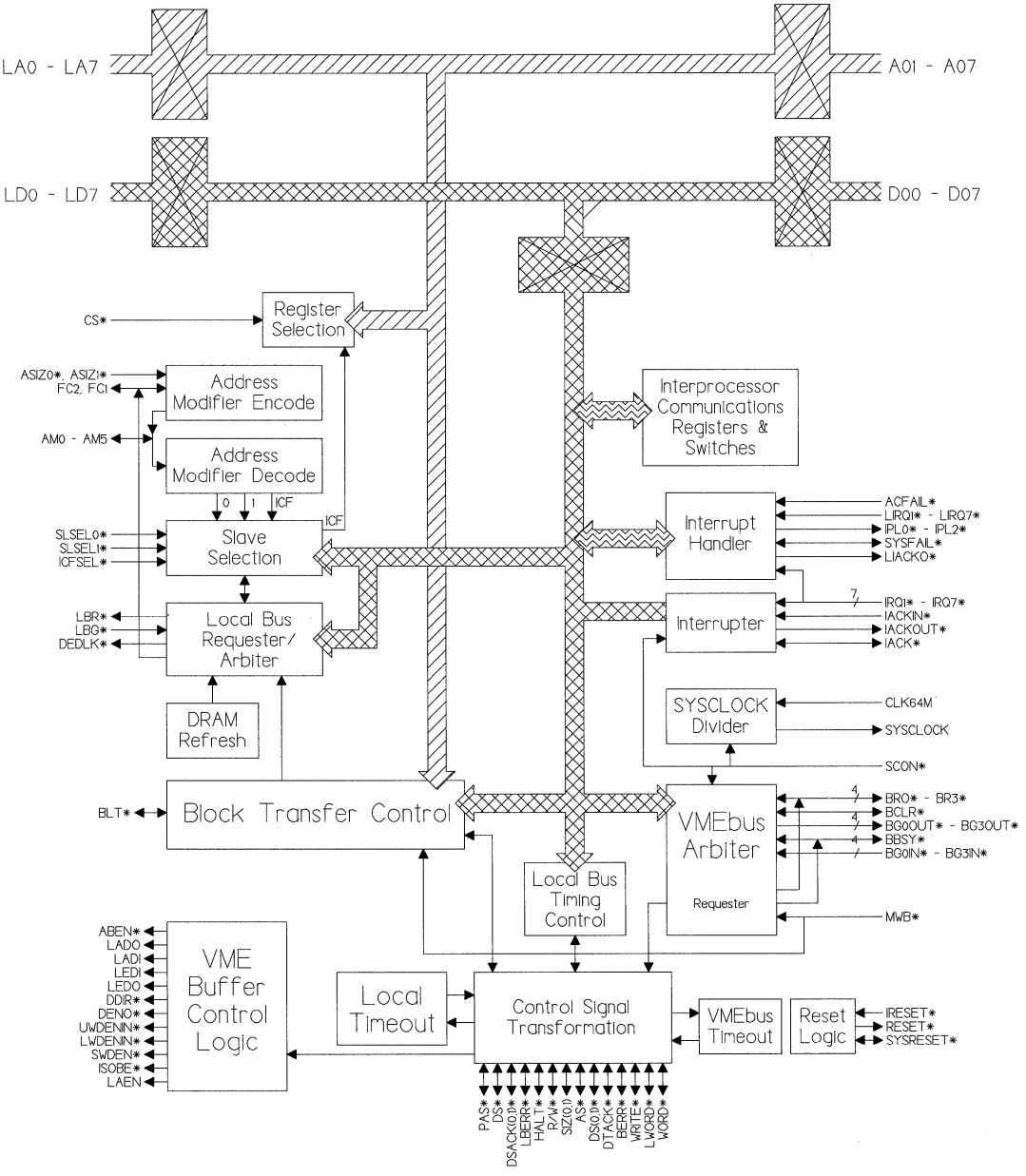


Figure 1. VIC068A Functional Block Diagram

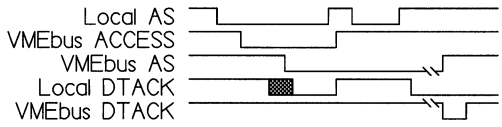


Figure 2. Master Write Posting

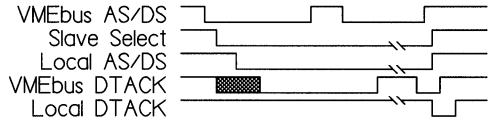


Figure 3. Slave Write Posting

plied to the VMEbus. Additionally, the VIC068A uses a type of pipelining to accelerate VMEbus throughput. On a block transfer read cycle, the slave VIC068A automatically prefetches the n + 1 data byte during the same read. The nth data byte is transferred across the VMEBus, and the n − 1 byte is latched in local RAM. As shown in *Figure 4*, this operation uses all three buses in overlapped and parallel operation to speed up the transfer. Write transfers use the same mechanism.

The limiting factor on the VMEbus transfer rate is either the VMEbus's many timing restrictions or the source or destination memories. If the memory consists of dynamic RAM, the restriction is probably the cycle time of the chips used, often as slow as 200 ns. To overcome this limitation, the VIC068A offers a programmable access mode so that attached DRAM can be used in page mode.

After a starting row address cycle (RAS), all subsequent cycles need only a column address (CAS) to reduce the access time, often by as much as half. For a slave interface, the VIC068A contains all the necessary counters and timing elements for local AS, DS, and address generation.

A master block transfer needs two or three additional latches for the higher address lines during the local DMA part of the block transfer. Thus, even with low-cost DRAMs, the VIC068's block transfer rate can reach 40 Mbytes/s, limited only by the VMEbus specification and the physical characteristics of the VMEbus.

This transfer rate decreases the time needed to load programs or move data to graphics boards, as well as increasing the VMEbus's bandwidth, thereby allowing more CPUs to work together in a multiprocessor system.

Mailbox Signaling

To add greater capability to multiprocessor systems, the VIC068A has four interprocessor communication global switches (ICGS) and four interprocessor communication module switches (ICMS). These are all byte-wide mailbox registers that generate a

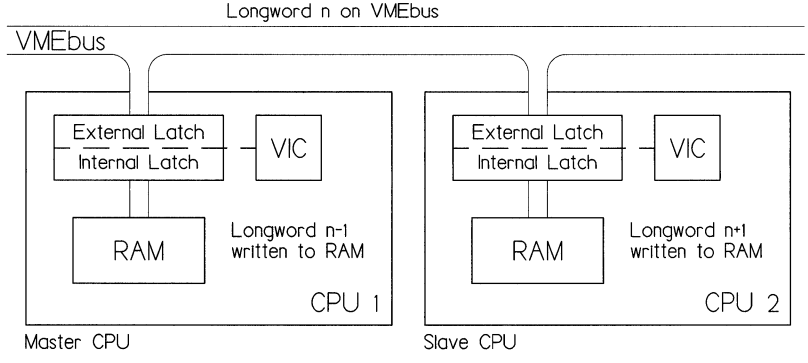


Figure 4. Block Transfer Read Cycle

local interrupt when accessed from the VMEbus. The ICGSs of one group reside at the same address and are accessed with a write cycle, which behaves as a broadcast to all members of the group. Because the ICMSs are at different addresses, one dedicated processor can be activated with a local interrupt request (LIRQ).

A processor can inform a logical group of processors about a new task via a broadcast using the ICGSs and can then communicate with single processors about the task using the ICMSs.

Eight-byte-wide interprocessor communication registers (ICR) are also available. Five of these registers serve as general-purpose read/write registers, and three are dedicated to control local activities (Halt, Reset, Mask ID, etc.). The ICRs can be read and written from the local side or the VMEbus without interfering with each other.

Interrupt Generator

The VIC068A handles up to seven simultaneous pending IRQs with separate vectors. The VIC068A also provides independent local IRQ vectors, if external IRQs are served.

Miscellaneous Features

The VIC068A furnishes several features for VMEbus support:

- SYSFAIL generation
- Software reset
- ACFAIL
- BERR register for detailed information

For local support, the VIC068A provides these features:

- Seven local IRQ sources, all level, polarity, edge and vector programmable
- Local bus timeout (2–512 ms)
- With /without VMEbus request time included
- 21 different local IRQ vectors
- VIC ID register

In addition to the VIC068, the following parts or equivalents are required for a minimum hardware interface:

- Three address latches and drivers (74xx543)
- Three data latches and drivers (74xx543)
- Four isolation buffers (74xx245)

You might also need the following:

- One to two PLDs for slave address decoding
- Two to three latches for master block transfer
- ½ PLD for block transfer glue logic

Interfacing

To connect a processor other than the 680x0 to the VIC068A, it is often easiest to map the processor control signals into the control signals available on a 680x0-type of processor. This type of transition interface offers the advantage of compatibility with a large family of 680x0-compatible peripheral parts, which you can then use elsewhere in the design.

Figure 5 shows a sample interface, whose four address latches store the multiplexed Mbus of the MC88000 processor. Four data latches store the data bytes after the acknowledge of the 680x0 bus and then start calculating parity the processor's MBus. The reason for this approach lies in some older peripheral I/O chips, which change their data lines when they should remain stable (i.e., transmit data buffer empty, etc.).

Two other data latches emulate the MC68020's dynamic bus sizing. The last buffer, between D0 – D7 of the 680x0 bus and AD16 – AD23 of the MBus, emulates the 680x0 bus's IRQ cycles with normal read cycles of the MC88000.

Acknowledgment

Cypress Semiconductor wishes to thank Jürgen Bullacher of Eltec GMbH and Eltec International S.A.R.L. for submitting this article.

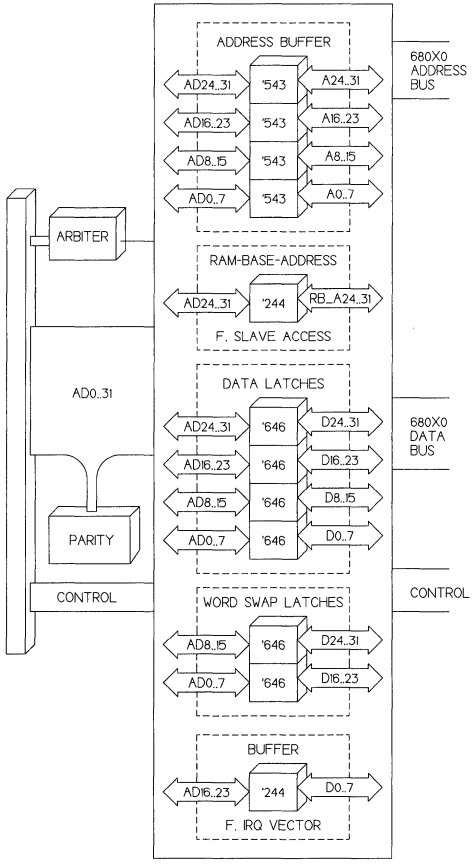


Figure 5. Sample Interface

Interfacing the VIC068A to the MC68020

This application note explains some of the features of the Cypress VIC068A and provides the first-time VIC068A user with simple implementations of these features. The VIC068A offers the most highly integrated VMEbus interface available today. It reduces the number of parts needed and saves board space. The emphasis in this application note is on interfacing the VIC068A as VMEbus A24/A16 D16/D08(E0) master/slave to the Motorola 68020.

Reset Operation

The VIC068A performs three distinct reset operations:

- Internal reset, activated by the IRESET pin, which initializes most of the internal registers
- System reset, essentially the same as IRESET, but is activated by writing ($F0) to the system reset register, or by asserting IRESET when the VIC068A is the VMEbus controller (SCON pin asserted)
- Global reset initializes all the VIC068A registers

After a reset, the 680X0 processor reads its initial stack pointer (SSP) and program counter (PC) from addresses $0 through $7. One way to handle this is to remap the boot-up ROMs to the low addresses for the first few cycles of the processor.

Figure 1 shows a circuit you can use to do this. The circuit uses a serial-in/parallel-out shift register (the 74HC164) to generate the MAP signal. This active-Low signal can be used with address-decode logic to force boot ROM access to the lower addresses during initial power up. Asserting the 74HC164 CLEAR pin drives all the parallel outputs Low, which asserts the selected MAP signal. With the two serial inputs tied High, each Low-to-High transition of the 68020 AS clocks the High through the shift register and out each of the parallel outputs. By picking the proper output for the MAP signal, you can decode from 1 to 8 of the initial processor cycles. You can use the MAP signals on memory configurations that are 8, 16, or 32 bits wide by using the QH, QD, or QB outputs, respectively.

Using the Processor RESET Instruction

The OR gate in *Figure 1* ensures that the 74HC164 is cleared only when HALT and RESET are both asserted. This allows the use of the 68020 RESET instruction without inadvertently reasserting MAP. An alternative to this approach is to use two small-signal diodes (1N4148) and a pull-down resistor in place of the OR gate. This change reduces the design's parts count by eliminating the 74HC32.

A ROM remapping circuit must be used whether the RESET instruction is issued or not because of the way the VIC068A arbitrates local bus contention between the 68020 and the VMEbus. Contention occurs when both master and slave operations are requested concurrently (MWB asserted and SLSEL0, SLSEL1, or IFCSEL asserted). The VIC068A indicates this contention by asserting DEDLK. You can deal with the condition by setting bit 4 of the VIC068's interface configuration register ($AF) to assert HALT along with LBERR when DEDLK occurs (68020 bus retry sequence). The VIC06 then waits for the 68020 to deassert the MWB input. Once this happens, the VIC068A releases LBERR but continues to assert HALT to keep the 68020 off the local bus. The VIC068A then allows the slave operation to complete and deasserts HALT. The 68020 can now retry the contested bus cycle.

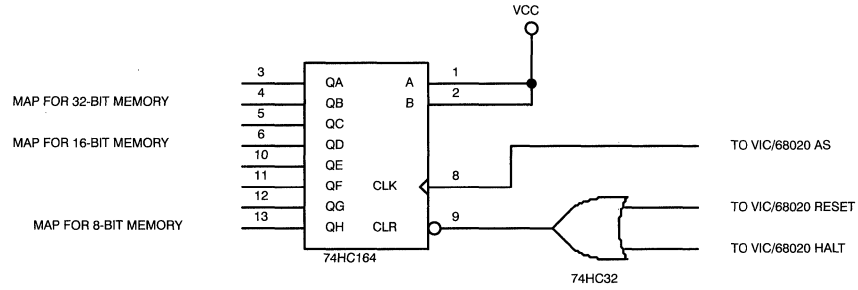


Figure 1. ROM Remapping Circuit

Internal Reset

At first glance, the IRESET might seem the logical choice for implementing the power-on reset. Because the IRESET input has some built-in hysteresis, a simple RC circuit would be appropriate for applying the power-on signal.

IRESET does not initialize the local bus timing register nor any of the slave select registers, however. Additionally, the VIC068A powers-up with the DRAM refresh option enabled (bit 4 of the arbiter/requester configuration register $B3 High). This condition is acceptable if you are using DRAM but adversely affects the external reset circuit in *Figure 1*. Specifically, for the first DRAM refresh cycle, the VIC068A deasserts RESET but maintains HALT in the active (Low) state and toggles AS. This action causes shift operations in the 74HC164. You can activate DRAM refresh after reset by writing a 1 to bit 4 of the arbiter/requester configuration register ($B3).

System Reset

The assertion of SYSRESET on the VMEbus typically activates system reset, but only when a global reset is not taking place. When the VIC068A is configured as the system controller (SCON pin asserted), it drives the SYSRESET pin for the required 200 ms during an internal or global reset.

Global Reset

The global reset is the most useful for power-up purposes because it places all the VIC068A registers in a known state. You initiate a global reset by asserting IPL(0) concurrent with or just after asserting IRESET. These reset signals should not be asserted until the V_{CC} power source has stabilized at 5 volts. Because IPL(0) is also one of the encoded interrupt lines for the 68020, you must assert this signal with an open-collector or three-state device.

In using global reset, bear in mind that when the VIC068A powers-up it ignores the VMEbus SYSRESET. The VIC068A releases HALT and RESET after the 200-ms time out even if the current VMEbus master asserts SYSRESET past this required minimum time. This automatic release is a useful feature because it eliminates reliance on the system controller to release SYSRESET to start the power-up sequence. Refer to the *VIC068A/VAC068A User's Guide* for more information on global reset.

The VIC068A generates a LBERR if you try to access the VMEbus or any of the VIC068A registers before SYSRESET is deasserted. One solution to this problem is to structure the software so that the VIC068A registers are set up as late as possible in the power-up sequence. You can also temporarily point the 68020 BERR exception vector to an address containing an RTE instruction and let the 68020 cycle in a BERR/RTE loop until SYSRESET is deasserted. The latter approach provides an op-

portunity to be the first board in a system to request VMEbus mastership.

Connecting the Bus Lines

Figure 2 shows the standard buffer configuration for an A24/D16 VMEbus connection. This design also supports A16 and D08(E0) operation.

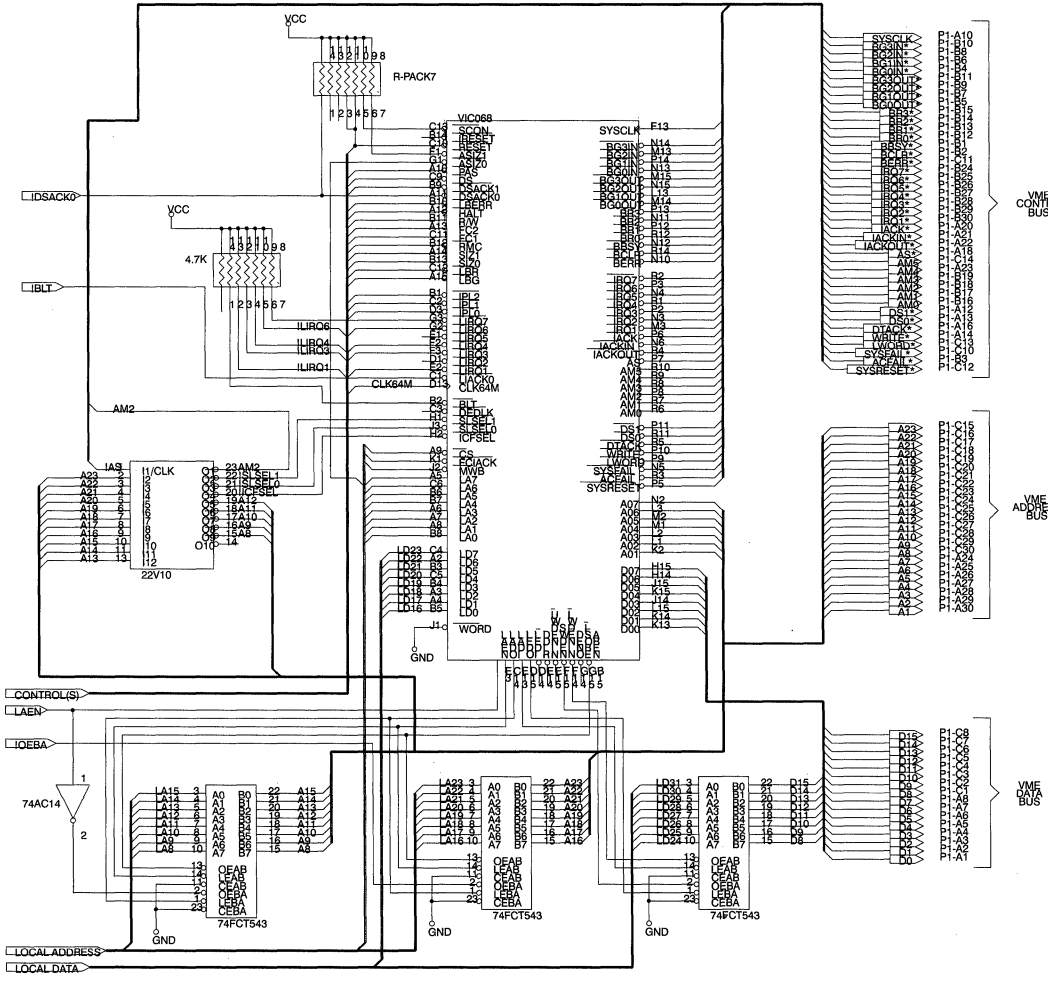


Figure 2. Address and Data Bus Connections

The D16/D08(E0) Data Bus

Connect the VIC068A to the 68020 as you would any 16-bit peripheral device. The 74FCT543 data buffer connects between the 68020 data bus's upper byte (D31 – 24) and the VMEbus D15 – 8 data lines. The lower byte (LD7 – LD0) is buffered through the VIC068A to the VMEbus low byte (D7 – D0). Several control signals connect directly from the VIC068A to the 74FCT543: DENO (data enable out) to OEAB (Output enable A-to-B), LWDENIN (lower word data enable) to OEBA (Output enable B-to-A), LEDO (latch enable data out) to LEAB (Latch enable A-to-B), and LEDI (latch enable data in) to LEBA (latch enable B-to-A).

The Address Bus

The A24/A16 configuration requires the use of two more 74FCT543 devices to buffer and control the VMEbus A23 through A8 signals. The 74FCT543 LEAB, LEBA, and OEBA inputs connect directly to the VIC068A LADO (latch address out control), LADI (latch address in control), and ABEN (enable address out control) outputs, respectively. The output of the VIC068A LAEN (local-address enable control) must be connected to the 74FCT543 OEBA input through an inverter because LAEN is an active-High output and OEBA is an active-Low input.

Connecting the DSACK Lines

During the normal local bus operation, the 68020's slave devices (i.e., memory, UART, parallel port) must tell the processor the size of their data bus. This is done by asserting the DSACK1 inputs, which tells the 68020 that the port is a 16-bit device. Asserting DSACK0 instead indicates that the port is an 8-bit device. Asserting both DSACK1 and DSACK0 indicates that the port is 32 bits wide. To configure the VIC068A as a 16-bit port, simply connect the 68020 DSACK1 to the VIC068A DSACK1.

So long as there you have no requirement for VMEbus access to 8-bit devices on the local bus, you do not need to do anything with the VIC068A DSACK0 pin except terminate it (pull it High).

When you do need to access 8-bit devices, a small problem arises with the way the VIC068A acknowledges register accesses and interrupt-acknowledge cycles. During these cycles, the VIC068A always asserts both DSACK1 and DSACK0, whether the WORD input is asserted or not. And in VMEbus master cycles, when talking to an 8-bit device on the VMEbus, the VIC068A responds with DASCK0 to acknowledge the 8-bit transfer completion.

The solution to the DSACK0 problem is simple but can be complicated to implement: You must break the DASCK0 connection between the VIC068A and the 68020 during interrupt acknowledge or VIC068A register access (CS) cycles. The circuit needed to do this is a bidirectional, open-collector buffer between the VIC068A and 68020. The buffer should be inactive in both directions only when the VIC068A FCIACK or CS inputs are asserted. In *Figure 3*'s PAL equations, the DSACK0_020 and VIC068A DSACK0 equation illustrates how to handle the DSACK0 connection.

Master Operation

VMEbus master operation with the VIC068A is easily accomplished with the use of the MWB (module-wants-bus) input. The VMEbus can be requested at any level (0 – 3). The VMEbus can also be dynamically changed via the arbiter/requester configuration register ($B3), which eliminates the need for hardware jumpers. All VMEbus release modes are supported through the release control register ($D3). Support for write posting means that the local processor can write to the VMEbus without having to wait for the current bus master to release the bus or for the arbitration logic to assert the correct BGIN 9 (bus grant in) line. The VIC068A takes cares of this overhead for the local processor, improving system throughput.

To request VMEbus mastership, the 68020 asserts the MWB input. You can think of MWB as a VMEbus chip select. When interfacing to the VMEbus as an A24 or A16 device, you can have access to the whole VMEbus address space by decoding a 32-Mbyte area of the 68020 address space for VMEbus operations. The ASIZ1-0 pins tell the VIC068A whether the current cycles represent an A32, A24,

or A16 operation. You can use the upper 16-Mbyte address space (A24 High) for VMEbus A23 operation and the lower half (A24 Low) for VMEbus A16 operation by following three steps: decode A31 through A25 to generate MWB, tie the ASIZ1 input High, and connect the 68020 A24 address line to the VIC068's ASIZ0 input. *Figure 3* demonstrates this way of decoding MWB.

When the VIC068A recognizes a valid slave access, the device asserts LBR (68020 BR input) and waits for LBG assertion (68020 BG output). Once the VIC068A receives LBG, the device becomes the local bus master at the conclusion of the current cycle and completes the requested VMEbus slave operation. If the VIC068A is the only DMA device on the local bus, there is no need to generate BGACK (bus grant acknowledge) for the 68020. But if any other devices are capable of local bus mastership, you have to provide the arbitration logic and the BGACK signal for the 68020. Keep in mind, too, that other DMA devices must be able to recognize and deal appropriately with the 68020 bus-cycle entry operation (BERR and HALT asserted).

Slave Operation

The VIC068A can provide full VMEbus slave operation by dual-porting local memory with little or no 68020 overhead. The normal slave access operation starts by providing SLSEL0 or SLSEL1 through VMEbus address decoding. The circuits in *Figures 2* and *3* use a 22V10 PAL for this purpose. Always qualify VMEbus address decoding with the AS and/or DS1-0.

Decoding SLSEL0, SLSEL1, and IFCSEL

Figure 3 illustrates a typical PAL specification that you can use to provide address decoding for SLSEL0, SLSEL1, and IFCSEL. The VIC068A uses all the address modifier lines (AM5 – 0) to qualify the access mode. Address decoding can ignore these inputs. The VIC068A then decides if the access mode is legal and completes the cycle or generates the VMEbus BERR signal, depending on the value programmed in the slave select registers. You can also qualify the select outputs with the address modifiers and let the initiating device time-out if the access is not legal.

The IFCSEL input gives the VMEbus access to some of the VIC068A control registers and the interprocessor communication registers. These registers are available only through an A16 privileged-mode access.

The PAL specification in *Figure 3* configures SLSEL0 to dual-port a 4-Kbyte (minus 256 bytes) space of local RAM as an A16 non-privileged access input and decodes IFCSEL in the SLSEL0 area's upper 256 bytes. You can use this 256-byte space for mailbox communication between boards in a multimaster system.

SLSEL1 is decoded as an A24 supervisory-only access and provides full dual-porting of the 68020 board's E2PROM program memory. This allows the VMEbus system controller to put the system in a reset and hold state by asserting bit 6 of the VIC068's interprocessor communications register 7. The VMEbus master can then reprogram the entire program memory space. Once that operation is complete, the controller can use the interprocessor communications register 7 to release the reset and hold state. The board comes up running the newly installed program.

Take care when decoding SLSEL0, SLSEL1, and IFCSEL. The VIC068's operation is undefined when more that one of these inputs is active simultaneously.

Decoding for Supervisor/User Mode

You can use the VMEbus AM2 signal to select user (AM2 Low) or supervisor (AM2 High) modes. The AM2 input is used as part of the slave-select decoding shown in *Figure 3*.

Dealing with A24 and A16 Slave Accesses

Regardless of the access address size, the 74FCT543 address buffer outputs are enabled. Typically, the backplane pulls unused VMEbus address lines High passively, but most masters drive these lines regardless of the bus-cycle-address size. If this is not desir-

```
module_CYCLE_DECODE;
   Cycle_decode device 'PV22V10';
   VCC,GND                              pin 24,12;

"inputs (15)
   A31,A30,A29,A28,A27,A26,A25,A19      pin 1,2,3,4,5,6,7,8;
   SLSEL1, SLSEL0                       pin 9,10;
   FC2,FC1,FC0,AS,LBG                   pin 13,14,15,16,17  "for FCIACK and VIC_Cycle output

"outputs (6)
   VIC_DSACK0,DSACK0_020    pin 18,19;    "To VIC DSACK0 and local system DASCK0
   VIC_CYCLE                pin 20;         "current bus cycle is VMEbus
   FCIACK                   pin 21;         "Interrupt Acknowledge Cycle
   PRE_MWB,MWB              pin 22,23;      "VIC module-wants-bus (with and without AS)

"output type declarations
   VIC_CYCLE,PRE_MWB,MWB          istype 'com';
   FCIACK,VIC_DSACK0,DSACK0_020   istype 'com';
   VIC_CYCLE.OE,FCIACK.OE         istype 'com';
   PRE_MWB.OE,MWB.OE              istype 'com';
   VIC_DSACK0.OE,DSACK0_020.OE    istype 'com';

equations in CYCLE_DECODE
"Enable ALL outputs except DSACK's
   VIC_CYCLE.OE     =1;
   PRE_MWB.OE       =1;
   MWB.OE           =1;
   FCIACK.OE        =1;

"This signal tells everybody that the VIC068A is controlling the current bus cycle
   !VIC_CYCLE<T>=!LBG & AS<T><T><T><T>"signal is asserted while AS is still high
         #!VIC_CYCLE & !LBG &!AS "maintain signal through entire cycle

"Interrupt acknowledge cycle (68020 to VIC).  Use VIC_CYCLE to insure this is not a VMEbus
master cycle
   !FCIACK = A31 & A30 & A29 & A28 & A27 & A26 & A25 & A19 & FC2 & FC1 & FC0 & !AS &
VIC_CYCLE;

"VME A24 access is at addresses $04000000 - $04FFFFFF.  A16 access is at addresses $0500000 -
$05FFFFFF (ASIZ0 is tied to LA24)
   !MWB = !A31 & !A30 & !A29 & !A28 & !A27 & A26 & !A25 & VIC_CYCLE &!(FC2 & FC1 & FC0);

"This is the same signal as MWB but the AS input is removed to provide an early VMEbus master
cycle indication input to other PLDS
   !PRE_MWB = !A31 & !A30 & !A29 & !A28 & !A27 & A26 & !A25 & VIC_CYCLE &!(FC2 & FC1 & FC0);

"This signal is connected directly to the VIC DSACK0.  It generates the VIC DSACK0 for VMEbus
slave accesses to 8 bit device
   !VIC_DSACK0 = !VIC_CYCLE & !DSACK0_020;

"This enables VIC_DSACK0 only when VIC is the local bus master (slave accesses)
   VIC_DSACK0.OE = !VIC_CYCLE & (!SLSEL0 # !SLSEL1);

"This signal is connected to the 68020 DSACK).  It generates the 68020 DSACK0 for VMEbus mas-
ter accesses to 8 bit devices
   !DSACK0_020 = !MWB & VIC_CYCLE & !VIC_DSACK0;

"This enables the 68020 DSACK0 only when the VIC is the VMEbus master
   DSACK0_020.020 = !MWB & VIC_CYCLE;
   end_CYCLE_DECODE
```

Figure 3. ABEL Equations for PALC22V10 Cycle Decoding

able, control the output-enable signals with the upper address line buffers using the VMEbus address modifiers. *Table 1* illustrates how to use AM5 and AM4 to determine the bus-cycle-address size.

You can derive individual enables for each of the VMEbus address latches by ANDing one or both of these address modifiers with the VIC068A LAEN (local-address enable) signal; modify both if operating in an A32 system.

Remember to provide a stable level for the local-address lines because nothing drives them during VMEbus accesses. You can ensure a stable level using 4.7Ω pull-up or pull-down resistors on the local-bus A31–A16 lines. The local-bus address buffers can be set to the desired address state and enabled with the same latch-enable signals.

Dual-Porting Local Memory

The PAL specification in *Figure 3* generates a signal called VIC_CYCLE than can serve as part of the local-address decoding to re-map local memory for dual-porting on the VMEbus. This approach allows memory placement at a VMEbus address independent of the local address.

Interrupts

The VIC068A interrupt structure is very versatile. One of the most useful features is the ability to redefine interrupt levels, and thus priorities, under normal program control. The VIC068A supports all seven levels of VMEbus interrupt as well as the seven local-interrupt levels. Interrupts are also available to notify the 68020 of VMEbus status and error conditions.

Figure 3 shows how to decode the 68020 interrupt acknowledge bus cycle to generate the VIC068A FCIACK input. You can omit A19–A16 from the equation if you do not use breakpoints, a memory management unit (MMU), or a coprocessor (68881/68882).

Using LIACK0

The LIACK0 output is typically connected to the 68020 AVEC input to initiate autovectoring of interrupts to which the VIC068A has not been programmed to respond. You can also use LIACK0 with the IPL(2–0) outputs to generate interrupt-acknowledge signals to other 680x0-compatible interrupting devices.

LIRQ7–1 Inputs

The LIRQ7–1 inputs are the interrupt request inputs to the VIC068. The control register for each input allows you to determine the input's polarity (high/low) and sensitivity (level or edge). The control register also allows you to define whether the VIC068A supplies the vector during interrupt acknowledge cycles or asserts LIACKO (local-interrupt acknowledge out), sets the level of interrupt the 68020 sees on IPL2–0, and enables or disables the interrupt. You do not need to terminate these inputs if you leave them unconnected, but you must pull them up externally if they are used.

The local interrupts (IPL2–0) are grouped and have a common vector base register ($57). This vector base is added to the encoded interrupt level programmed in each of the interrupt control registers to supply a unique vector to the 68020 for each interrupt input.

LIRQ2 is a special case because it can be used as an interrupt clock tick timer. You enable the timer through bits 2 and 3 of slave-select control register 0($C3). When enabled, LIRQ2 becomes the timer output, and the local-interrupt control register 2 ($2B) becomes the timer's interrupt-control register. The timer's periodic interrupt can be set to 50, 100, or 1000 Hz. If you plan on using the tick timer, do not connect the external interrupts to LIRQ2 because this pin becomes an output.

Table 1. Determining Bus-Cycle-Address Size

AM5	AM4	Cycle
H	H	A24 Access
H	L	A16 Access
L	L	A32 Access

Connecting the Cypress VIC068/VAC068 to the TI TMS320C40: A Prototype Design

Introduction

The Cypress Semiconductor VIC068 VMEbus Interface Controller and its companion VAC068 VMEbus Address Controller provide a complete VMEbus interface including master and slave capability (Reference 2). As these components can be used in a wide variety of applications, it is natural to utilize the VIC068/VAC068 in a single- or multiple-TMS320C40 VMEbus card design.

This application note provides high-level as well as low-level details of interfacing VIC/VAC to TMS320C40. This allows for techniques to be implemented to minimize design time for subsequent efforts since this design has not been optimized for either size or speed. The Design Requisites section provides the design goals established prior to design as well as relevant background regarding devices involved. Hardware details, including schematics and programmable logic source code, represent the central focus of the paper. In addition, software initialization of the chip, set by the TMS320C40, is covered. Throughout this note, it is assumed that the reader is familiar with the TMS320C40 architecture (Reference 1), the basics of the VIC068A/VAC068A (Reference 2), and the VMEbus and its protocol(s) (References 5, 6).

Design Requisites

Design Goals

This project began with the development of a set of design goals for the VME interface based on our particular needs. The main focus was on a TMS320C40 card providing both (VMEbus) master and slave capability for reads, writes, read-modify-writes, write posting, and slave block transfers. In terms of the address/data capability, a design was made to the most prevalent configuration (for other cards available commercially): 24-bit address and 32-bit data (i.e., A24/D32). However, the design presented here does not preclude 32-bit addressing with minor modifications. Via the VIC068, this design also provides system controller capability. VMEbus interrupt support is not provided. The VAC068 is utilized to provide address control/mapping, two Universal Asynchronous Receiver/Transmitters (UARTs) (required for our application), and a general purpose parallel I/O. In addition to the VMEbus functionality, interface compatibility is required with both the existing TMS320C40–40, which has 50-nanosecond cycle time, and the faster TMS320C40, 40-nanosecond part.

Design Considerations

The 68020 User's Manual (Reference 7) was referenced extensively for this design, which covers a complete examination of the VIC068 and VAC068 and extends to review the 680x0 family bus signals and cycles. An examination of the VIC068 and VAC068 reveals a direct interface to the Motorola 680x0 family data, address, and control signals. The VIC068 and VAC068 are both driven with the familiar 680x0 address and data strobes (PAS*, DS*), which have an asynchronous transfer protocol implemented with the data transfer and size acknowledgment signals DSACK0* and DSACK1*. In addition to these signals, the transfer size signals, SIZ0 and SIZ1, are an integral part of the 680x0's dynamic bus sizing capability and, with the the lower ad-

dress lines, encode the size of the transfer in progress. During transfer, the function code signals (FC0–FC2) provide additional information of importance in multi-user environments. Bus arbitration is an integral part of the 680x0 via the bus request (BR*), bus grant (BG*), and bus grant acknowledge (BGACK*) signals and are used directly by the VIC068. Finally, like many other general purpose microprocessors, bus cycles for the 680x0 are several clock cycles long.

Although the VIC068 and VAC068 are driven by, and can drive, the familiar 680x0 bus signals, a quick examination of the TMS320C40 bus signals shows little similarity to the 680x0 family. The TMS320C40 provides a bus protocol common to the TMS320 floating-point DSP product line. An external ready signal allows for wait-state generation and controls the rate of transfer in a synchronous fashion (i.e., cycles can be extended an integer number of clock cycles). As described in Reference 1, the TMS320C40 provides a 32-bit address range divided into two identical 31-bit address, 32-bit data buses termed local and global. The TMS320C40 executes single cycle instructions and relies on a multistage pipeline for execution speed. Detailed bus status is provided for each cycle via the STAT lines, which provide information regarding the type of instruction and access. Individual control lines are provided for three-stating the address, data, and control bus(es). A read-modify-write signal is not provided (as it is on the 680x0). However, an instruction-driven LOCK* signal is available. Each cycle is controlled by a strobe (STRBx*) signal in conjunction with the corresponding read/write (R/Wx*) strobe. One of the TMS320C40's many features is the range of configuration options for this external interface. The TMS320C40 has evolved from its earlier floating-point counterparts and provides a truly flexible interface via the local and global bus configuration control registers.

High-Level Architecture

The high-level architecture for the card places 20-nanosecond (ns), high-density, 4-megabit (128Kx32) Cypress CMOS SRAM modules on both local and global buses of the TMS320C40. (The size of the memory array should not impact the TMS320C40-to-VIC068/VAC068 interface design.) The global side is designated as program memory and the local side as data memory for the application. It is anticipated that the local memory will be fully occupied by DMA coprocessor activity coupled with data fetches during communications-oriented DSP operations. Given this, the A24 VME spectrum is placed on the global (program) side, segmenting the local side I/O activity, the critical path for the application, from all VMEbus activity. (Note that the interface documented herein can be used on either side due to the symmetry of the global and local buses.) In addition to programming SRAM on the global side, two 128Kx16 EPROMs for embedded program store are also placed, with the boot load feature of the TMS320C40 applied.

Because the design is limited to VMEbus A24 addressing, its spectrum fits nicely anywhere in the TMS320C40 global side address spectrum, from 08000 0000h to 0FFFF FFFFh. Therefore, VMEbus master access is memory mapped into the TMS320C40 global side address range. When an access occurs in this predefined A24 range, the TMS320C40 bus signals are transformed into 680x0 bus signals. These drive the VIC068/VAC068 pair and initiate a VMEbus transfer. Global side accesses outside of this range do not generate such signals and occur at full speed (i.e., the speed appropriate for that memory or peripheral). Regarding the "endianess" (References 8, 9) of the interface, it is known that the 680x0 family maintains big-endian byte ordering (byte addressable memory organization) with little-endian bit ordering in each addressable unit. In contrast, the TMS320C40 is flat in its byte endianess (32-bit word addressable only) and little-endian bit ordered. Therefore, no swapping is done on the interface as 32-bit word transfers (between processors) maintain D0 as the least significant bit. This forces the designer to tradeoff transfer speed with a wider range of transfers (byte, word, and three byte) than inherent to the TMS320C40 (longword). Transfers are limited to D32. In order to provide transfers of all sizes, additional set-up and/or decoding would have to be done prior to/during the transfer in progress.

Hardware Description

After examining the VIC068/VAC068 interface and capabilities and comparing them with the TMS320C40, a prototype design was initiated. Based on the above discussion, the strategy is to map from the given set of TMS320C40 bus signals to a set of 680x0-like signals driving their counterparts on the VIC068 and VAC068 for master cycles. (Note: the TMS320C40 is reading from or writing to the VMEbus.) Not only can the TMS320C40 initiate VMEbus cycles as a bus master, the card can also respond to slave cycles. Most often, slave access is used in terms of access to shared memory on the (slave) card. To accomplish this on the TMS320C40-based VME card, a set of signals is required to respond to bus requests from the VIC068/VAC068 and an additional set is required to "hold off" the TMS320C40 global side during such transfers.

To accomplish this transformation of signals, programmable logic is applied. It is desirable to keep the design time to a minimum while maintaining the most flexible or programmable design. Based on this, Cypress's CY7C335 universal synchronous EPLDs (Reference 3) are used. These devices are field programmable and optimized for state machines. The 335 has 12 input macrocells, 12 output macrocells, 256 product terms, 4 buried registers, and operates to a maximum frequency of 100 megahertz (MHz). Development tools for these EPLDs include *Warp2*™, a VHDL compiler from Cypress, and Data I/O's ABEL™ version 4.0 or better, using fitters available on the Cypress Bulletin Board (408−943−2954).

Reset Circuitry

The VIC068 *must* receive a global reset in order to function correctly. The global reset should occur after the power supply and the 64-MHz (U4) oscillator have stabilized and before any interaction with the chip is attempted. The VIC requires both the leading and trailing edges of IRESET*/IPL0, as shown on page 14−32 of the Users Guide and as discussed in Chapter 12.

To implement this function, an R/C network along with a pair of Schmitt Trigger inverters are used to create a power-up signal. The R/C values are not given since they will be a function of the system power supply and oscillator power-up delay time. The power-up signal is supplied to U12, a 22V10 that is programmed as a state machine to create the required waveforms. (Part of U12 is also used for address decoding.) After the power up delay is complete, the power-up signal goes High, which causes U12 to drive IRESET Low. The state machine waits for the VIC to respond by driving RESET Low. It then drives IPL0 Low for two clock cycles indicating a global reset is to take place. IPL0 is returned to a High state followed by the IRESET signal. If the VIC is configured as a VMEbus system controller, a global reset will cause the VIC to drive the SYSRESET line for 200 ms, as required by the VMEbus specification. The RESET output on the VIC068 is used to reset both the TMS320C40 as well as the VAC068 and all programmable logic onboard.

Address Bus Decoding

The VIC068/VAC068 interface, and consequently the VMEbus itself, is mapped into the TMS320C40 global side at 0D000 0000h. In this application, the global side is divided into two halves via the STRB ACTIVE field in the TMS320C40 (global) memory control register. Zero-wait state devices (fast SRAM) are placed in the lower half and slower memory (EPROM) and peripherals (the VIC068/VAC068 pair) are placed in the upper half. Therefore, the TMS320C40 addresses program memory via STRB0 and addresses the VMEbus via STRB1. As shown in the accompanying schematics, U12 is a 22V10 PAL used for $\overline{\text{STRB1}}$ address bus decoding. In particular, a Cypress PAL22V10D−7 was used. This PAL is used to decode each (global) TMS320C40 $\overline{\text{STRB1}}$ bus cycle using the TMS320C40 H1 clock. Cycle type decoding is accomplished fully via the STAT lines (versus simply using the R/W strobe) and allows for future expansion/reconfiguration if required. As shown, the $\overline{\text{STRB1}}$ range is divided into 8 distinct segments via use of GA28−GA30. (GA31 is implicitly a logic 1.) Outputs of the decoding operation are (VMEbus) master write (MWR*), master read (MRD*),

VIC068/VAC068 register write (RWR*), register read (RRD*) and EPROM read (GPROM*). As found in the VIC068/VAC068 documentation, the VAC068 is hard-wired, starting at address 0FFFD 0000h. It also provides for VIC068 selection, starting at address 0FFFC 0000h. A memory map for the global side, as decoded by the 22V10 PAL, is shown in *Table 1*. The ABEL source code is provided in the appendix.

Table 1. Global Side Memory Map

Address	Unit Addressed
08000 0000h	SRAM
0C000 0000h	EPROM
0D000 0000h	VMEbus A24
Address 00 0000h	
0FFFC 0000h	VIC068 Register Set
0FFFD 0000h	VAC068 Register Set

Bus Control

Once a cycle in the VMEbus address range is detected by the address decoding PAL, the sequencers shown provide the signals required for both master and slave cycles. U13 is the first of three sequencers and accomplishes overall bus control providing enable signals for global bus access by the TMS320C40 (GBE*), master cycle sequencer output (MOE*), slave cycle sequencer output (SOE*), VMEbus slave local bus grant (LBG*) and a ready signal for the TMS320C40 (GRDY1*). Notice that a full complement of inputs are presented to the bus control sequencer. This is done to accommodate all possible cycles and to allow reconfiguration without hardware changes. While the TMS320C40 H3 clock (20 MHz) is used here, this is not an absolute requirement as the array of sequencers operate asynchronously once a master or slave cycle begins. The use of H3 here, however, does simplify the sequencer code because the H3 clock serves as a convenient reference to the TMS320C40 cycle in progress.

A master cycle begins with U12 generating a master read or write signal or either register read or write. This enables the output of the master bus cycle signal generation sequencer U14. In fact, this signal is asserted during all bus activity other than slave cycles. Termination of a master cycle ends with the assertion of the acknowledge signals DSACK0* and DSACK1* and/or the local bus error signal (LBERR*). All are generated by the VIC068 in response to acknowledge signals provided over the VMEbus. The sequencer responds to these signals by providing the ready signal for this TMS320C40 $\overline{\text{STRB1}}$ access (RDY1*) by asserting GRDY1*. In this design, external ready signals are used exclusively, versus ANDing or ORing with internal ready, and the generation of the ready signal conforms to the second of two methods called out in Reference 1: ready is High between accesses.

Slave cycles are initiated by the assertion of local bus request (LBR*) by the VIC068. With this asserted, U13 provides bus control by first disabling the TMS320C40 global bus (deasserting GBE*), then disabling the master bus cycle generation sequencer outputs (U14 MOE*), followed by enabling the outputs on the slave bus cycle signal generation sequencer (SOE*), U15. Given that the bus has been successfully "seized", the local bus grant signal (LBG*) is asserted. Slave cycles are terminated by deassertion of the local bus request input.

Master Bus Cycle Generation

The master bus cycle generation sequencer U14 runs in tandem with the bus control seqencer U13. The sequencer code found in U13 and U14 results from one common state diagram. It is necessary to split these functions up due to limitations of the number of outputs per sequencer. Therefore, the inputs to U14 are identical to those found on U13. Master bus cycles proceed according to the appropriate cycle (read or write) definition found in Reference 7. The function code lines are driven to a supervisory state, giving the widest possible audience, supervisory data. Termination of a master cycle ends with the assertion of the acknowledge signals DSACK0* and DSACK1* and/or the local bus error signal (LBERR*) as described above. Note that VIC068/VAC068 register accesses are also master accesses in the address range(s) specified above. While the sequencer code does not initiate read-modify-write cycles, it is easy to see how the

use of the TMS320C40 GLOCK* input could be used to accomplish this.

Slave Bus Cycle Generation

Slave cycles are initiated by the VAC068 in response to a request over the VMEbus in the selected range as determined in the appropriate VAC068 register (covered in the next section). As shown, inputs to the sequencer are the common 680x0 bus signals driven by the VIC068 for slave cycles and alternately driven by the master sequencers for master cycles. Assertion of the local bus grant signal (LBG*) by U13 indicates the absence of the TMS320C40 on the global bus, thereby allowing access of (shared) global SRAM by the VIC068/VAC068 pair. Assuming the correct transfer size, the memory strobe signals GSTRB0* and GR/W0 are driven to provide access to the shared global SRAM. Following this, acknowledgement is provided via DSACK0* and DSACK1*, ending the slave cycle. Note that while VAC068 documentation states that its DSACK signals can be three-stated on the assertion of LAEN, this was not the case with this configuration. Therefore, U8A was required to artificially three-state those signals so that the slave sequencer could control the data acknowledgement.

VIC068/VAC068 Software Initialization

The VIC068/VAC068 pair register set can be overwhelming at first glance, but very few registers require attention prior to using the pair for either master or slave operations. The VAC068 should be initialized first as this controls both master and slave address mapping. With that complete, the VIC068 is programmed. Fine tuning of the interface can be performed using the programmable delay registers for the interface after initial capability is verified. As the VIC068/VAC068 was programmed using C, vic.h and vac.h header files were developed which provide base and offset definitions for the complete register set of each device.

Before programming the VIC068/VAC068 pair, the VAC068 must be brought out of its initial "Force EPROM" mode which asserts EPROMCS* for all accesses. This is accomplished by reading from the EPROM space beginning at 0FF00 0000h. The address decoding PAL U12 does not provide for access to this range. However, a dummy access to this region can be initiated by manipulating the TMS320C40 (global) memory interface control register. The register is set to provide zero-wait state, internal ready dependent (only) accesses to the appropriate strobe (STRB1 for this case). With this set in the SWW and WTCNT fields, a read from address 0FF00 0000h is performed. Immediately following this read, the SWW field to external ready accesses is reset and a second read to the VAC068 is performed, this time at the VAC068 register base, 0FFFD 0000h. This read provides the required access to "snap" the sequencers back to their default states.

After the "Force EPROMCS" mode is exited, it is verified that the VAC068 can be addressed by reading the device ID via the VAC068 ID register. With that established, the (slave) SLSEL0 Base Address register and the SLSEL0 Address Mask register followed by the (master) A24 Base Address register are programmed. To enable the VAC068 decode and compare functions, the last step is to write to the VAC068 ID register. The VIC068 ID register is similarly polled and. following the successful read of that register, the Address Modifier Source register and the Slave Select 0 Control 0 register are set. This completes the initial programming of the pair. At this time, the SYSFAIL LED, if applicable, may be extinguished by writing to the Interprocessor Communication 7 register. The initial register settings for this application are provided in *Table 2*.

Table 2. VIC068/VAC068 Initial Register Settings

Address	Register	Size (Bits)	Setting
0FFFD 0200h	VAC SLSEL0 Base	16	0010h
0FFFD 0300h	VAC SLSEL0 Mask	16	00F0h
0FFFD 0800h	VAC A24 Base	13	0D10h
0FFFD 2900h	VAC ID	16	Write to EnableVAC
0FFFC 00B4h	VIC Address Modifier	8	03Dh
0FFFC 00C0h	VIC Slave Select 0 Control 0	8	014h

Conclusion

The development of a prototype interface between the TMS320C40 DSP and the Cypress VIC068/VAC068 was accomplished with a minimum amount of programmable logic in the form of simple PALs and sequencers. The result is a reconfigurable, programmable interface for A24/D32 VMEbus master/slave capability. While the initial transfer speed is low, speed gains can be made by increasing the sequencer's clock speed and eliminating unnecessary states in the prototype sequencer code. Read-modify-write cycles can be accomplished with the existing hardware via the use of the TMS320C40 LOCK instruction group. Ultimately, the design documented herein could be encapsulated in an FPGA for both speed and size gains.

References

1. *TMS320C4x Users Guide*. Texas Instruments, 1991.
2. *VIC068A/VAC068A Users Guide*. Cypress Semiconductor, 1992.
3. *Cypress Semiconductor CMOS/BiCMOS Data Book*. Cypress Semiconductor, 1992.
4. Siy, P. F., and W. T. Ralston, "Application of the TI TMS320C40 in Satellite Modem Technology," to be published, to be presented at the Third Annual International Conference on Signal Processing Applications and Technology, November, 1992, Boston, MA.
5. *IEEE Standard for a Versatile Backplane Bus: VMEbus*. IEEE, 1987, New York, NY: Wiley-Interscience.
6. Peterson, W. D., *The VMEbus Handbook*, Scottsdale, AZ: VFEA International Trade Association.
7. *MC68020 32-Bit Microprocessor User's Manual*. Motorola, Inc., 1984
8. Henessey, J. L., and D. A. Patterson, 1990, *Computer Architecture A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc.
9. Dewar, R. B. K, and M. Smosna, 1990, *Microprocessors A Programmers View*, New York, NY: McGraw-Hill, Inc.

Appendix A. Address Bus Decoder – ABEL Source

```
module          U12
title           'Global Bus Decode
Revision        1.0
Part            Cypress PAL22V10D-7
Abel Version    4.3
Project         C40 I/O Board'

     U12 device 'P22V10';

"Inputs"
clk, reset                  pin 1,2;     "clock, reset"
gstat0,gstat1               pin 3,4 ;    "C40 status"
gstat2,gstat3               pin 5,6;     "C40 status"
ga28,ga29,ga30              pin 7,8,9;   "C40 address"
gstrb1                      pin 10;      "C40 strobe 1"
oute                        pin 11;      "output enable
pureset                     pin 13;      "output enable"

"Outputs"
ipl0,ireset,tmp1            pin 17,18,16 istype 'reg,invert';
mrd,mwr                     pin 19,20;   "master read & write"
rrd,rwr                     pin 21,22;   "register read & write"
gprom                       pin 23;      "PROM select"

"Misc"
ga31 = 1;                               "dummy var"

"Sets"
stat = [gstat3,gstat2,gstat1,gstat0];    "status"
addr = [ga31,ga30,ga29,ga28];            "ms nibble"
output = [gprom,rwr,rrd,mwr,mrd];        "output"
power_up = [ireset,ipl0,tmp1];           "reset

" state definations for power up reset sequence.
"  [ireset,ipl0,tmp1]
s0 = [1,1,1] ; "initial state
s1 = [0,1,1] ; "ireset asserted
s2 = [0,0,1] ; "ireset/ipl0 asserted
s3 = [0,0,0] ; "hold for extra clock
s4 = [0,1,0] ; "de-assert ipl0
s5 = [1,1,0] ; "de-assert ireset
s6 = [1,0,0] ; "should never get here
s7 = [1,0,1] ; "should never get here

H,L,X,C,Z = 1,0,.X.,.C.,.Z.;
```

Appendix A. Address Bus Decoder – ABEL Source (continued)

```
equations
output.c = clk;
power_up.c = clk;
output.oe = !oute;
power_up.oe = !oute;

"Master Read"
!mrd := reset & (addr == ^hd) & (stat == [1,0,X,X]) & !gstrb1;
"Master Write"
!mwr := reset & (addr == ^hd) & ((stat == [1,1,0,1]) #
(stat == [1,1,1,0])) & !gstrb1;

"Register Read"
!rrd := reset & (addr == ^hf) & (stat == [1,0,X,X]) & !gstrb1;
"Register Write"
!rwr := reset & (addr == ^hf) & ((stat == [1,1,0,1]) #
(stat == [1,1,1,0])) & !gstrb1;

"PROM Read"
!gprom := reset & (addr == ^hc) & (stat == [1,0,X,X]) & !gstrb1;

" power up reset state equations
state_diagram power_up

" Initial power up state
state s0:
if (pureset) then s1
else s0;

" assert ireset
state s1:
if (!pureset) then s0
else if (!reset) then s2
else s1;

"assert ipl0
state s2:
if (!pureset) then s0
else s3;

"keep both asserted for extra clock
state s3:
if (!pureset) then s0
else s4;
```

Appendix A. Address Bus Decoder – ABEL Source (continued)

```
"de-assert ipl0
state s4:
if (!pureset) then s0
else s5;

"de-assert ireset and stop
state s5:
if (!pureset) then s0
else s5;

"check on indeterminate states
state s6:
if (!pureset) then s0
else s6;

"check on indeterminate states
state s7:
if (!pureset) then s0
else s7;

test_vectors
([clk,reset,gstat3,gstat2,gstat1,gstat0,ga30,ga29,ga28,
gstrb1,oute] -> output)

[C,X,X,X,X,X,X,X,X,X,1] ->     Z;      "1 test for high-z"
[C,0,X,X,X,X,X,X,X,X,0] ->    ^b11111;"2 test for reset"
[C,1,1,0,X,X,1,0,1,0,0] ->    ^b11110;"3 test for master read"
[C,1,1,1,0,1,1,0,1,0,0] ->    ^b11101;"4 test for master write"
[C,1,1,1,1,0,1,0,1,0,0] ->    ^b11101;"5 test for master write"
[C,1,1,0,X,X,1,1,1,0,0] ->    ^b11011;"6 test for register read"
[C,1,1,1,0,1,1,1,1,0,0] ->    ^b10111;"7 test for register write"
[C,1,1,1,1,0,1,1,1,0,0] ->    ^b10111;"8 test for register write"
[C,1,1,0,X,X,1,0,0,0,0] ->    ^b01111;"9 test for PROM read"
[C,1,1,0,X,X,0,0,0,0,0] ->    ^b11111;"10 test bad address"
[C,1,0,0,0,0,1,1,1,0,0] ->    ^b11111;"11 test bad status"
end U12
```

Appendix B. Bus Control Sequencer – ABEL Source

```
module          U13C
title           'C40 Bus Control
Revision        1.0
Part            CY7C335
Abel Version    4.3 using Cypress fitter
Project         TMS320C40 I/O Card '

U13C device 'p335';

"Inputs"
clk, reset                  pin 1,13;          "clock, reset"
!mrd,mwr,rrd,rwr,gprom       pin 24,11,10,9,12; "decoded cycle"
mwb,lbr                     pin 7,6;           "master/slave requests"
dedlk                       pin 5;             "m/s deadlock"
dsack0,!dsack1,!lberr       pin 4,28,15;       "cycle responses"
!glock                      pin 26;            "C40 lock"
oe                          pin 14;            "output enable"

"Outputs"
lbg       pin 27      istype 'reg_RS,invert' ;"slave grant"
gbe       pin 16      istype 'reg_RS,invert' ;"C40 g bus enable"
soe,moe   pin 17,18   istype 'reg_RS,invert' ;"pls oe(s)"
grdy1     pin 19      istype 'reg_RS,invert' ;"C40 ready 1"

"Sets"
cycle  = [gprom,rwr,rrd,mwr,mrd];  "cycle request"
ack    = [dsack1,dsack0];          "acknowledge"
output = [grdy1,soe,moe,gbe,lbg];  "output"

"State Description"
P4,P3,P2,P1 node 34,33,32,31;
P0   pin  25 istype 'reg,invert';
P4,P3,P2,P1  istype 'reg';

sreg = [P4,P3,P2,P1,!P0];
 S0 = [0,0,0,0,0];
 S1 = [0,0,0,0,1];
 S2 = [0,0,0,1,0];
 S3 = [0,0,0,1,1];
 S4 = [0,0,1,0,0];
 S5 = [0,0,1,0,1];
 S6 = [0,0,1,1,0];
 S7 = [0,0,1,1,1];
 S8 = [0,1,0,0,0];
 S9 = [0,1,0,0,1];
 S10 = [0,1,0,1,0];
 S11 = [0,1,0,1,1];
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
 S12 = [0,1,1,0,0];
 S13 = [0,1,1,0,1];
 S14 = [0,1,1,1,0];
 S15 = [0,1,1,1,1];
 S16 = [1,0,0,0,0];
 S17 = [1,0,0,0,1];
 S18 = [1,0,0,1,0];
 S19 = [1,0,0,1,1];
 S20 = [1,0,1,0,0];
 S21 = [1,0,1,0,1];
 S22 = [1,0,1,1,0];
 S23 = [1,0,1,1,1];
 S24 = [1,1,0,0,0];
 S25 = [1,1,0,0,1];
 S26 = [1,1,0,1,0];
 S27 = [1,1,0,1,1];
 S28 = [1,1,1,0,0];
 S29 = [1,1,1,0,1];
 S30 = [1,1,1,1,0];
 S31 = [1,1,1,1,1];

"Misc"
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe;          "set output enable"
output.CLK = clk;         "clock the output regs"
sreg.CLK = clk;           "and state regs"

@page
state_diagram sreg
state S0:

if !reset then S0 WITH
 lbg.S = 1; "slave disable"
 gbe.R = 1; "enable C40 global side"
 soe.S = 1; "slave disable"
 moe.R = 1; "enable master pls"
 grdy1.S = 1; "not ready"
 ENDWITH;

else if (!mrd # !mwr & lbr) then S1;     "master read/write"
else if (!rrd # !rwr & lbr) then S4;     "reg read/write"
else if (!gprom & lbr) then S8;          "EPROM read"
else if (!lbr # !dedlk) then S16 WITH    "slave request"
gbe.S = 1;                               "disable global side"
moe.S = 1;                               "and master pls"
 ENDWITH;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
else S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "slave disable"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

@page
"Master Read/Write"

state S1:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if !dedlk & ((!mwb) # (mwb)) then S16 WITH
 moe.S = 1;
 gbe.S = 1;
 ENDWITH;

else if !mwb then S2;"wait for !mwb"

else S1;

state S2:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if !dedlk & ((!mwb) # (mwb)) then S16 WITH;
 moe.S = 1;
 gbe.S = 1;
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S3 WITH
 grdy1.R = 1;
 ENDWITH;

else S2;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
state S3:
goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
"Register Read/Write"

state S4:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if !dsack1 then S5 WITH
 grdy1.R = 1;
 ENDWITH;

else S4;

state S5:
goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
"EPROM Read, 150ns EPROMs"

state S8:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S9;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
state S9:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S10;

state S10:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S11;

state S11:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S12 WITH
 grdy1.R = 1;
 ENDWITH;

state S12:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S0 WITH
 grdy1.S = 1;
 ENDWITH;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
@page
"Local Bus Request"

state S16:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S17 WITH
 soe.R = 1;          "enable slave PLS"
 ENDWITH;

state S17:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S18 WITH
 lbg.R = 1;     "finally allow slave access"
 ENDWITH;

state S18:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if lbr then goto S19 WITH
 lbg.S = 1;     "slave disable"
 ENDWITH;

else S18;

state S19:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S20 WITH
 soe.S =1;          "disable slave pls"
 ENDWITH;

state S20:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S0 WITH
 moe.R = 1;
 gbe.R = 1;
 ENDWITH;

@page
"make sure there are no undefined states"
state S21: goto S0;
state S22: goto S0;
state S23: goto S0;
state S24: goto S0;
state S25: goto S0;
state S26: goto S0;
state S27: goto S0;
state S28: goto S0;
state S29: goto S0;
state S30: goto S0;

"Power-Up"

 state S31:
goto S0 WITH
 lbg.S = 1;     "slave disable"
 lbg.R = 0;     "dummy err 6099"
 gbe.R = 1;     "enable C40 global side"
 gbe.S = 0;     "dummy err 6099"
 soe.S = 1;     "disable slave PLS"
 soe.R = 0;     "dummy err 6099"
 moe.R = 1;     "enable master pls"
 moe.S = 0;     "dummy err 6099"
 grdy1.S = 1;   "not ready"
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
 grdy1.R = 0;   "dummy err 6099"
 ENDWITH;

@page
" Test vectors will not work with beta version of
" Abel 4.3
test_vectors

([clk,reset,gprom,rwr,rrd,mwr,mrd,lbr,mwb,
 dsack1,dsack0,dedlk,lberr,glock,oe] ->
 [!sreg,grdy1,soe,moe,gbe,lbg])

[1,X,X,X,X,X,X,X,X,X,X,X,X,X,0]->[S31,X,X,X,X,X];"1 power up"
[0,X,X,X,X,X,X,X,X,X,X,X,X,X,0]->[S31,X,X,X,X,X];"2 power up"
[C,0,X,X,X,X,X,X,X,X,X,X,X,X,0]->[S0,1,1,0,0,1]; "3 resetstate"
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0]->[S1,1,1,0,0,1];"4 master read"
[C,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0]->[S2,1,1,0,0,1];"5 mwb asserted"
[C,1,1,1,1,1,1,0,1,0,0,0,1,1,1,0]->[S3,0,1,0,0,1];"6 data acked"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0,1,1,0,0,1];"7 ready for nxt"
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0]->[S1,1,1,0,0,1];"8 master write"
[C,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0]->[S2,1,1,0,0,1];"9 mwb asserted"
[C,1,1,1,1,1,1,0,1,0,0,0,1,1,1,0]->[S3,0,1,0,0,1];"10 data acked"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0,1,1,0,0,1];"11 rdy for nxt"
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0]->[S4,1,1,0,0,1];"12 reg read"
[C,1,1,1,1,0,1,1,1,1,1,0,1,1,1,0]->[S5,0,1,0,0,1];"13 data ackd"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0,1,1,0,0,1];"14 rdy for nxt"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S8,1,1,0,0,1];"15 prom read"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S9,1,1,0,0,1];"16 prom read"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S10,1,1,0,0,1];"17 wait"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S11,1,1,0,0,1];"18 wait"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S12,0,1,0,0,1];"19 wait"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0, 1,1,0,0,1];"20 rdy for nxt"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S16,1,1,1,1,1];"21slaverequest'
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S17,1,0,1,1,1];"22 en slve pls"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"23 slave grant"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"24 slave aces"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S19,1,0,1,1,1];"25rescnd grant"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S20,1,1,1,1,1];"26disbl sl pls"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0, 1,1,0,0,1];"27end sl acces"
[C,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0]->[S16,1,1,1,1,1];"29 deadlock"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S17,1,0,1,1,1];"30 en slve pls"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"31 slave grant"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"32 slave aces"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S19,1,0,1,1,1];"33rescnd grant"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S20,1,1,1,1,1];"34disbl sl pls"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0, 1,1,0,0,1];"35end sl acces"

end U13C
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source

```
module          u14a
title           'C40 Bus Control
Revision        1.0
Part            CY7C335
Abel Version    4.3 using Cypress fitter
Project         TMS320C40 I/O Card '

U14a device 'p335';

"Inputs"
clk, reset                  pin 1,13;        "clock, reset"
mrd,mwr,rrd,rwr             pin 12,11,10,9; "decoded cycle"
mwb,lbr,gprom               pin 7,6,5 ;      "master/slave requests"
dedlk                       pin 4;          "m/s deadlock"
!dsack0,!dsack1,lberr       pin 28,27,2;    "cycle responses"
glock                       pin 3;          "C40 lock"
oe                          pin 14;          "output enable"

"Outputs"
pas  pin 18     istype 'invert,reg_RS';"68K address strobe"
ds   pin 16     istype 'invert,reg_RS';"68K data strobe"
rw   pin 17     istype 'invert,reg_RS';"68K read/write bar"
rmc  pin 15     istype 'invert,reg_RS';"68K read-mod-write"
siz0 pin 19     istype 'invert,reg_RS';"68K size 0"
siz1 pin 20     istype 'invert,reg_RS';"68K size 1"
fc1  pin 23     istype 'invert,reg_RS';"68K function 1"
fc2  pin 24     istype 'invert,reg_RS';"68K function 0"

"Sets"
cycle  = [gprom,rwr,rrd,mwr,mrd];  "cycle request"
ack    = [dsack1,dsack0];          "acknowledge"
output = [pas,ds,rw,rmc,siz0,siz1,fc1,fc2]; "68K ouputs"

"State Description"
P4,P3,P2,P1  node 34,33,32,31 istype 'reg';
P0           pin  25          istype 'reg,invert';
sreg = [P4,P3,P2,P1,!P0];
 S0 = [0,0,0,0,0];
 S1 = [0,0,0,0,1];
 S2 = [0,0,0,1,0];
 S3 = [0,0,0,1,1];
 S4 = [0,0,1,0,0];
 S5 = [0,0,1,0,1];
 S6 = [0,0,1,1,0];
 S7 = [0,0,1,1,1];
 S8 = [0,1,0,0,0];
 S9 = [0,1,0,0,1];
 S10 = [0,1,0,1,0];
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
 S11 = [0,1,0,1,1];
 S12 = [0,1,1,0,0];
 S13 = [0,1,1,0,1];
 S14 = [0,1,1,1,0];
 S15 = [0,1,1,1,1];
 S16 = [1,0,0,0,0];
 S17 = [1,0,0,0,1];
 S18 = [1,0,0,1,0];
 S19 = [1,0,0,1,1];
 S20 = [1,0,1,0,0];
 S21 = [1,0,1,0,1];
 S22 = [1,0,1,1,0];
 S23 = [1,0,1,1,1];
 S24 = [1,1,0,0,0];
 S25 = [1,1,0,0,1];
 S26 = [1,1,0,1,0];
 S27 = [1,1,0,1,1];
 S28 = [1,1,1,0,0];
 S29 = [1,1,1,0,1];
 S30 = [1,1,1,1,0];
 S31 = [1,1,1,1,1];

"Misc"
rwmem  pin 26 istype 'reg_RS,invert'; "r/w flag"

H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe;          "set output enable"
output.CLK = clk;         "clock the output regs"
sreg.CLK = clk;           "and state regs"
rwmem.CLK = clk;          "and r/w store"

@page

state_diagram sreg
state S0:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc
 siz0.R = 1     "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
else if (!mrd & !rwmem & lbr) then S1 WITH    "master read"
 rw.S = 1;           "assert read/write"
 rwmem.S =1;
 ENDWITH;

else if (!mrd & rwmem & lbr) then S2 WITH     "master read"
 pas.R = 1;          "assert pas"
 ds.R = 1;           "and ds"
 ENDWITH;

else if (!mwr & rwmem & lbr) then S8 WITH     "master write"
  rw.R = 1;          "assert r/w"
 rwmem.R = 1;
 ENDWITH;

else if (!mwr & !rwmem & lbr) then S9 WITH    "master write"
pas.R = 1;           "assert pas only"
 ENDWITH;

else if (!rrd & !rwmem & lbr) then S16 WITH   "reg read"
 rw.S = 1;           "assert r/w"
 rwmem.S = 1;
 ENDWITH;

else if (!rrd & rwmem & lbr) then S17 WITH    "reg read"
 pas.R = 1;          "assert pas"
 ds.R = 1;           "and ds"
 ENDWITH;

else if (!rwr & rwmem & lbr) then S24 WITH    "reg write"
rw.R = 1;
 rwmem.R = 1;
 ENDWITH;

else if (!rwr & !rwmem & lbr) then S25 WITH
 pas.R = 1;          "assert pas only"
 ENDWITH;

else S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
@page
"Master Read"

state S1:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else S2 WITH
 pas.R = 1;
 ds.R = 1;
 ENDWITH;

state S2:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !mwb then S3;          "wait for !mwb"

else S2;

state S3:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
else if ((!dsack1 & !dsack0) # !lberr) then S4
" WITH
" grdy1.R = 1"
ENDWITH;

else S3;

state S4:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
"Master Write"

state S8:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1; "  no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else S9 WITH
 pas.R = 1;
 ENDWITH;

state S9:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else S10 WITH
 ds.r = 1;
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
state S10:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !mwb then S11;
else S10;

state S11:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S12;

else S11;

state S12:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
"Register Read"

state S16:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else S17 WITH
 pas.R = 1;
 ds.R =1;
 ENDWITH;

state S17:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !dsack1 then S18
"WITH
" grdy1.R = 1
" ENDWITH;

else S17;

state S18:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
"Register Write"

state S24:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
ENDWITH;

else S25 WITH
 pas.R = 1;
 ENDWITH;

state S25:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else S26 WITH
 ds.r = 1;
 ENDWITH;

state S26:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !dsack1 then S27;

else S26;

state S27:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
@page
"Power-Up"

state S31:
goto S0 WITH
 pas.S = 1;     "no strobe"
 pas.R = 0;     "error 6099 fix"
 ds.S= 1;       "no strobe"
 ds.R= 0;       "error 6099 fix"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rw.R = 0;      "error 6099 fix"
 rmc.S = 1;     "no rmc"
 rmc.R = 0;     "error 6099 fix"
 siz0.R = 1;    "set for"
 siz0.S = 0;    "error 6099 fix"
 siz1.R = 1;    "32-bit xfers"
 siz1.S = 0;    "error 6099 fix"
 fc1.R = 1;     "set for supervisory"
 fc1.S = 0;     "error 6099 fix"
 fc2.S = 1;     "data access"
 fc2.R = 0;     "error 6099 fix"
ENDWITH;

@page
test_vectors

([clk,reset,gprom,rwr,rrd,mwr,mrd,lbr,mwb,
 dsack1,dsack0,dedlk,lberr,glock,oe] ->
[!sreg,rwmem,fc2,fc1,siz1,siz0,rmc,rw,ds,pas])

"1 power up"
[1,X,X,X,X,X,X,X,X,X,X,X,X,0] -> [S31,X,X,X,X,X,X,X,X,X];
"2 power up"
[0,X,X,X,X,X,X,X,X,X,X,X,X,0] -> [S31,X,X,X,X,X,X,X,X,X];
"3 reset state"
[C,0,X,X,X,X,X,X,X,X,X,X,X,0] -> [S0, 1,1,0,0,0,1,1,1,1];
"4 master read"
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,0] -> [S2, 1,1,0,0,0,1,1,0,0];
"5 mwb  asserted"
[C,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0] -> [S3, 1,1,0,0,0,1,1,0,0];
"6 data acked"
[C,1,1,1,1,1,1,0,1,0,0,0,1,1,1,0] -> [S4, 1,1,0,0,0,1,1,0,0];
"7 ready for  nxt"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 1,1,0,0,0,1,1,1,1];
"8 master  write"
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0] -> [S8, 0,1,0,0,0,1,0,1,1];
"9 assert pas"
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0] -> [S9, 0,1,0,0,0,1,0,1,0];
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
"10 assert ds"
[C,1,1,1,1,0,1,1,1,1,1,1,1,0] -> [S10,0,1,0,0,0,1,0,0,0];
"11 mwb"
[C,1,1,1,1,0,1,1,0,1,1,1,1,0] -> [S11,0,1,0,0,0,1,0,0,0];
"12 data ackd"
[C,1,1,1,1,0,1,1,0,0,0,1,1,0] -> [S12,0,1,0,0,0,1,0,0,0];
"13 ready for next"
[C,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 0,1,0,0,0,1,0,1,1];
"14 reg read"
[C,1,1,1,1,0,1,1,1,1,1,1,1,0] -> [S16,1,1,0,0,0,1,1,1,1];
"15 assert strobes"
[C,1,1,1,1,0,1,1,1,1,1,1,1,0] -> [S17,1,1,0,0,0,1,1,0,0];
"16 data ackd"
[C,1,1,1,1,0,1,1,1,1,1,0,1,0] -> [S18,1,1,0,0,0,1,1,0,0];
"17 ready for nxt"
[C,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 1,1,0,0,0,1,1,1,1];
"18 reg write"
[C,1,1,0,1,1,1,1,1,1,1,1,1,0] -> [S24,0,1,0,0,0,1,0,1,1];
"19 assert pas"
[C,1,1,0,1,1,1,1,1,1,1,1,1,0] -> [S25,0,1,0,0,0,1,0,1,0];
"20 assert ds"
[C,1,1,0,1,1,1,1,1,1,1,1,1,0] -> [S26,0,1,0,0,0,1,0,0,0];
"21 data ackd"
[C,1,1,0,1,1,1,1,1,1,0,1,1,0] -> [S27,0,1,0,0,0,1,0,0,0];
"22 ready for next"
[C,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 0,1,0,0,0,1,0,1,1];

end u14a
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source

```
module          u15a
title           'C40 Bus Control
Revision        1.0
Part            CY7C335
Abel Version    4.30
Project         TMS320C40 I/O Card '

U15a device 'p335';

"Inputs"
clk, reset      pin 1,13;  "clock, reset"
pas,ds          pin 12,11;  "address,data strobe"
rw,rmc          pin 10,9;  "read/write strobes"
siz0,siz1       pin 7,6;  "bus sizing"
fc0,fc1,fc2     pin 5,4,3;"function codes"
lbg             pin 2;   "local bus grant"
oe              pin 14;   "output enable"

"Outputs"
dsack0    pin 15    istype 'invert,reg_RS'; "data ack 0"
dsack1    pin 17    istype 'invert,reg_RS'; "data ack 1"
lberr     pin 19    istype 'invert,reg_RS'; "bus error"
gstrb0    pin 23    istype 'invert,reg_RS'; "C40 mem strobe"
grw0      pin 25    istype 'invert,reg_RS'; "C40 read/write"

"Sets"
size   = [siz1,siz0];    "size"
func   = [fc2,fc1,fc0];  "function"
output = [grw0,gstrb0,lberr,dsack1,dsack0];

"State Description"
P3,P2,P1,P0    node 34,33,32,31    istype 'reg';
sreg = [P3,P2,P1,P0];
 S0 = [0,0,0,0];
 S1 = [0,0,0,1];
 S2 = [0,0,1,0];
 S3 = [0,0,1,1];
 S4 = [0,1,0,0];
 S5 = [0,1,0,1];
 S6 = [0,1,1,0];
 S7 = [0,1,1,1];
 S8 = [1,0,0,0];
 S9 = [1,0,0,1];
 S10 = [1,0,1,0];
 S11 = [1,0,1,1];
 S12 = [1,1,0,0];
 S13 = [1,1,0,1];
 S14 = [1,1,1,0];
 S15 = [1,1,1,1];
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
"Misc"
!rwmem pin  27 istype 'reg_RS,invert'; "r/w flag"
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe; "set output enable"
output.CLK = clk;"clock the output regs"
sreg.CLK = clk;  "and state regs"
rwmem.CLK = clk; "and r/w store"

@page
state_diagram sreg
state S0:

if (!reset) then S0 WITH
 dsack0.S = 1; "deassert"
 dsack1.S = 1; "all"
 lberr.S = 1;  "strobes"
 gstrb0.S = 1; "deassert C40"
 grw0.R = 1;   "strobe, read"
 rwmem.S = 1;  "set to read"
 ENDWITH;

else if (!lbg) then S1;

else S0 WITH
 dsack0.S = 1; "deassert"
 dsack1.S = 1; "all"
 lberr.S = 1;  "strobes"
 gstrb0.S = 1; "deassert C40"
 grw0.R = 1;   "strobe, read"
 rwmem.S = 1;  "set to read"
 ENDWITH;

@page
"Sort Slave Request"
state S1:

"Reset"
if (!reset) then S0 WITH
 dsack0.S = 1; "deassert"
 dsack1.S = 1; "all"
 lberr.S = 1;  "strobes"
 gstrb0.S = 1; "deassert C40"
 grw0.R = 1;   "strobe, read"
 rwmem.S = 1;  "set to read"
 ENDWITH;
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
"32-Bit Read"
else if (!pas & !ds & rw & !rwmem & !siz0 & !siz1) then S2 WITH
grw0.S = 1;
 rwmem.S = 1;
 ENDWITH;

else if (!pas & !ds & rw & rwmem & !siz0 & !siz1) then S3 WITH
gstrb0.R = 1;
 ENDWITH;

"32-Bit Write"
else if (!pas & !ds & !rw & rwmem & !siz0 & !siz1) then S2 WITH
grw0.R = 1;
 rwmem.R = 1;
 ENDWITH;

else if (!pas & !ds & !rw & !rwmem & !siz0 & !siz1) then S3 WITH
gstrb0.R = 1;
 ENDWITH;

"Illegal Access (non-32 bit access)"
 else if (!pas & !ds & (rw # !rw) & (siz0 # siz1)) then S9 WITH
lberr.R = 1;
 ENDWITH;

else S1;

@page
"32-Bit Read/Write"

state S2:
goto S3 WITH
 gstrb0.R = 1;
 ENDWITH;

state S3:
goto S4 WITH
 dsack0.R = 1;
 dsack1.R = 1;
 ENDWITH;

state S4:
if pas then S0 WITH
 dsack0.S = 1;
 dsack1.S = 1;
 gstrb0.S = 1;
 ENDWITH;
```

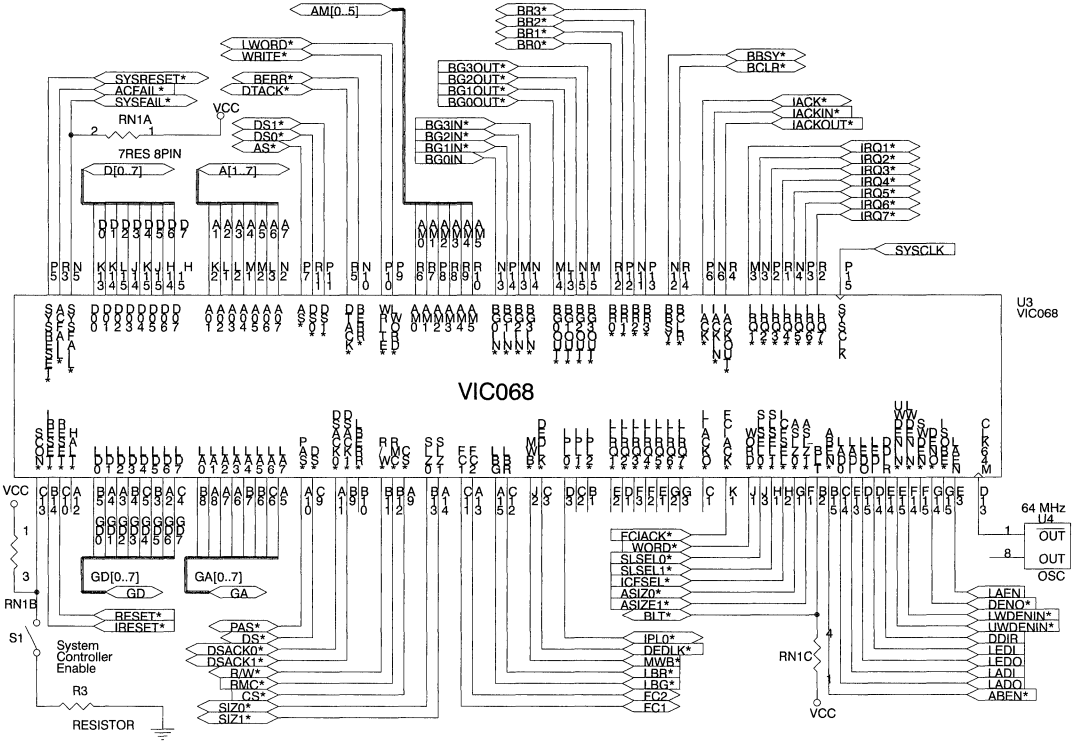
Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
else S4;

@page
"Illegal Access"

state S9:
if pas then S0 WITH
 lberr.S = 1;
 ENDWITH

@page
"Power-Up"

state S15:
goto S0 WITH
 dsack0.S = 1; "no ack"
 dsack0.R = 0; "error 6099 fix"
 dsack1.S = 1; "no ack"
 dsack1.R = 0; "error 6099 fix"
 rwmem.S = 1;  "r/w mem"
 rwmem.R = 0;  "error 6099 fix"
 lberr.S = 1;  "no bus error"
 lberr.R = 0;  "error 6099 fix"
 gstrb0.S = 1; "no strobe"
 grw0.S = 1;   "read"
 ENDWITH;

@page
test_vectors

([clk,reset,pas,ds,rw,rmc,siz0,siz1,fc0,fc1,fc2,lbg,oe] ->
[!sreg,rwmem,dsack0,dsack1,lberr,gstrb0,grw0])

[1,X,X,X,X,X,X,X,X,X,X,X,0]->[S15,X,X,X,X,X,X];"1 power up"
[0,X,X,X,X,X,X,X,X,X,X,X,0]->[S15,X,X,X,X,X,X];"2 power up"
[C,0,X,X,X,X,X,X,X,X,X,X,0]->[S0, 1,1,1,1,1,1];"3 reset state"
[C,1,1,1,1,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"4 slave read,lbg"
[C,1,0,1,1,1,0,0,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"5 pas asserted"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S3, 1,1,1,1,0,1];"6 and ds,strobe"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"7 ack"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"8 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 1,1,1,1,1,1];"9 dne, rel gstrb"
[C,1,1,1,0,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"10 slav wrte,lbg"
[C,1,0,1,0,1,0,0,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"11 pas assert"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S2, 0,1,1,1,1,0];"12 and ds"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S3, 0,1,1,1,0,0];"13 asert strob"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"14 ack"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"15 wtfor pas rel"
```

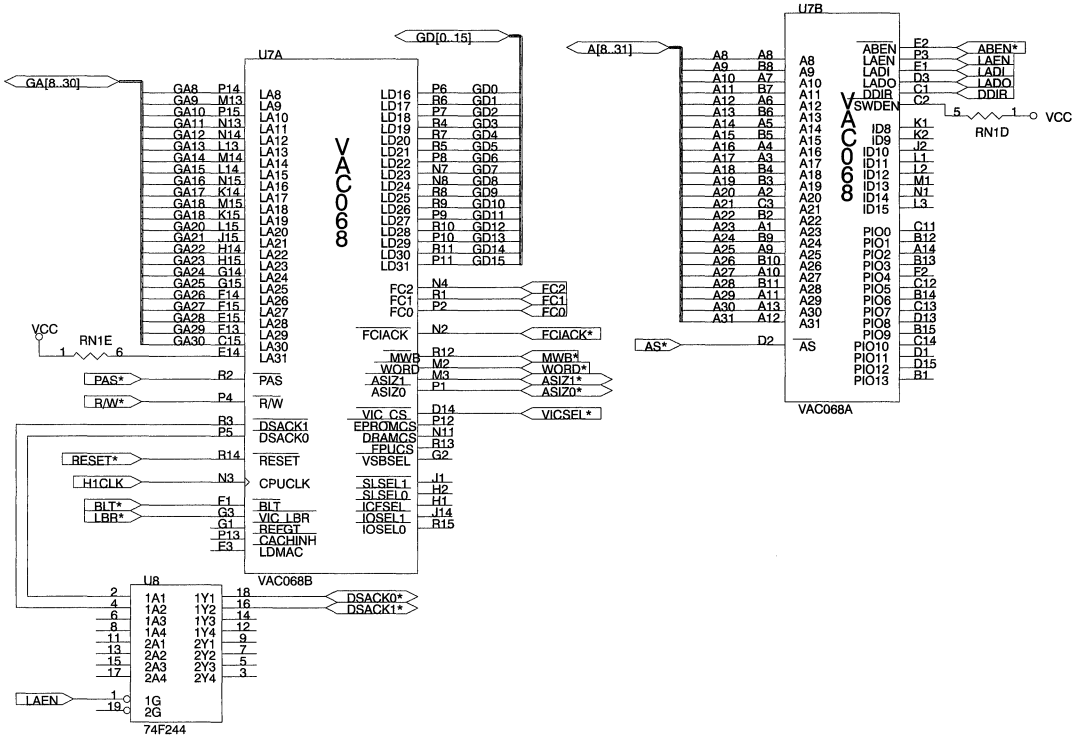
Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 0,1,1,1,1,0];"16done,rel gstrb"
[C,1,1,1,1,1,1,1,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"17 slav read,lbg"
[C,1,0,1,1,1,0,0,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"18 pas asserted"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S2, 1,1,1,1,1,1];"19 & ds,r/w asrt"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S3, 1,1,1,1,0,1];"20 and strobe"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"21 ack"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"22 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 1,1,1,1,1,1];"23done,rel gstrb"
[C,1,1,1,1,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"24 bad acess,lbg"
[C,1,0,1,1,1,0,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"25 pas asserted"
[C,1,0,0,1,1,0,1,X,X,X,0,0]->[S9, 1,1,1,0,1,1];"26 & ds, error"
[C,1,0,0,1,1,0,1,X,X,X,0,0]->[S9, 1,1,1,0,1,1];"27 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 1,1,1,1,1,1];"28done,rel lberr"
[C,1,1,1,0,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"29 slv write,lbg"
[C,1,0,1,0,1,0,0,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"30 pas asserted"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S2, 0,1,1,1,1,0];"31 and ds"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S3, 0,1,1,1,0,0];"32 assert strobe"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"33 ack"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"34 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 0,1,1,1,1,0];"35done,rel gstrb"
[C,1,1,1,0,1,1,1,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"36 slav wrte,lbg"
[C,1,0,1,0,1,0,0,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"37 pas asserted"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S3, 0,1,1,1,0,0];"38 & ds,asrt str"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"39 ack"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"40 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 0,1,1,1,1,0];"41done,rel gstrb"

end u15a
```

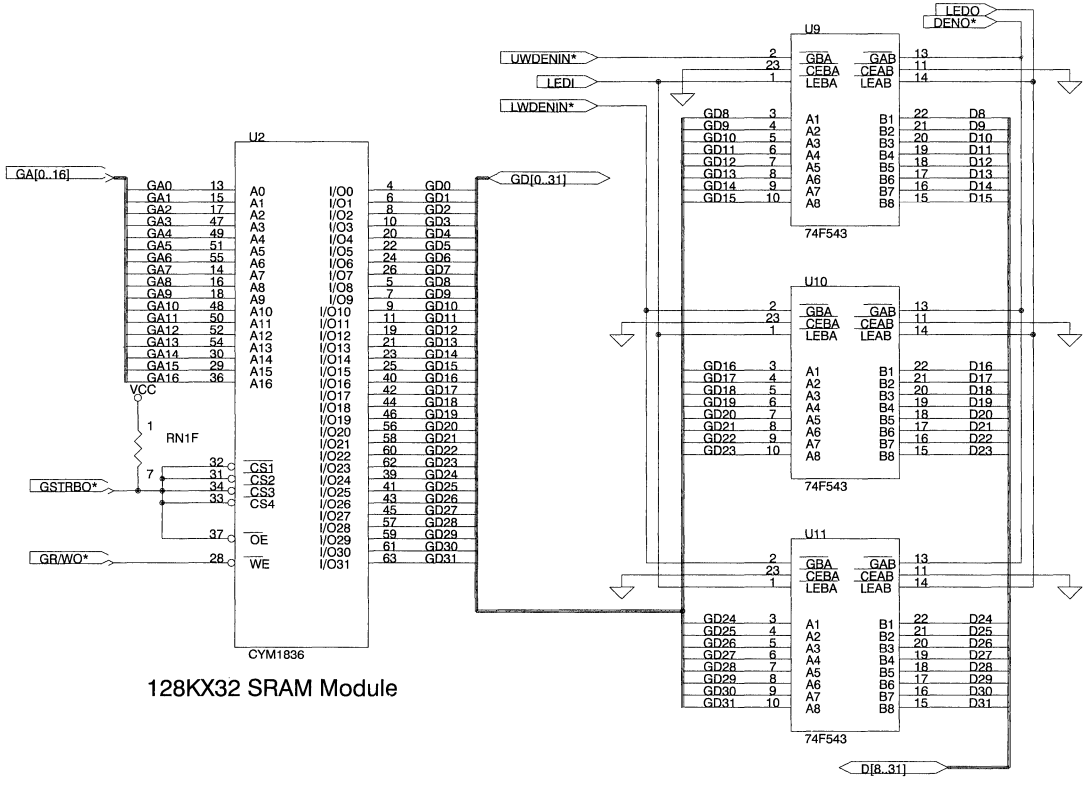
Appendix E. Schematics



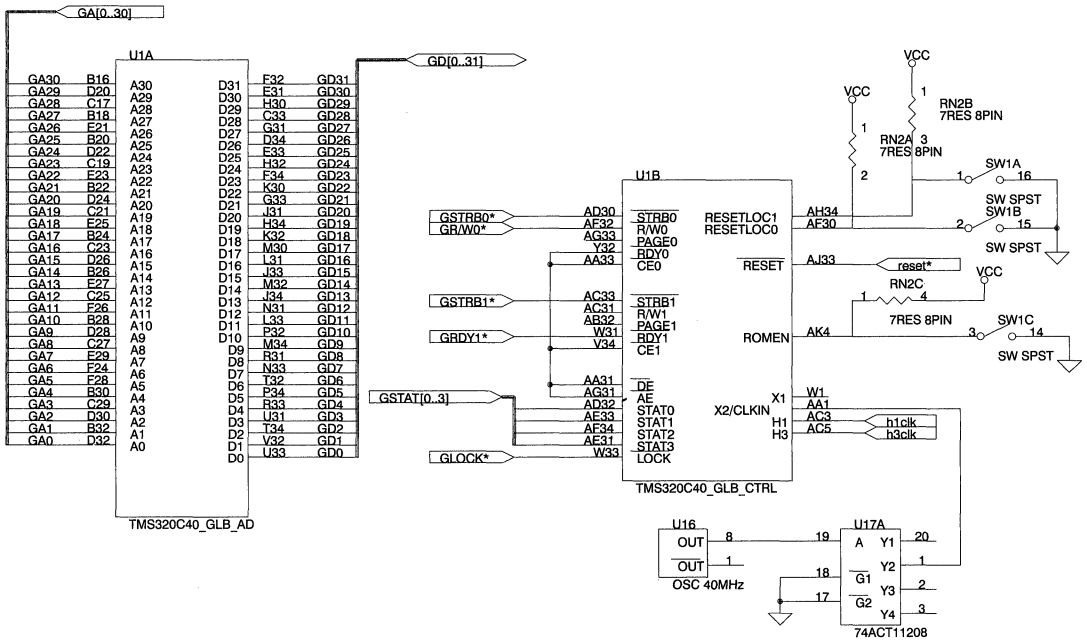
Appendix E. Schematics (continued)



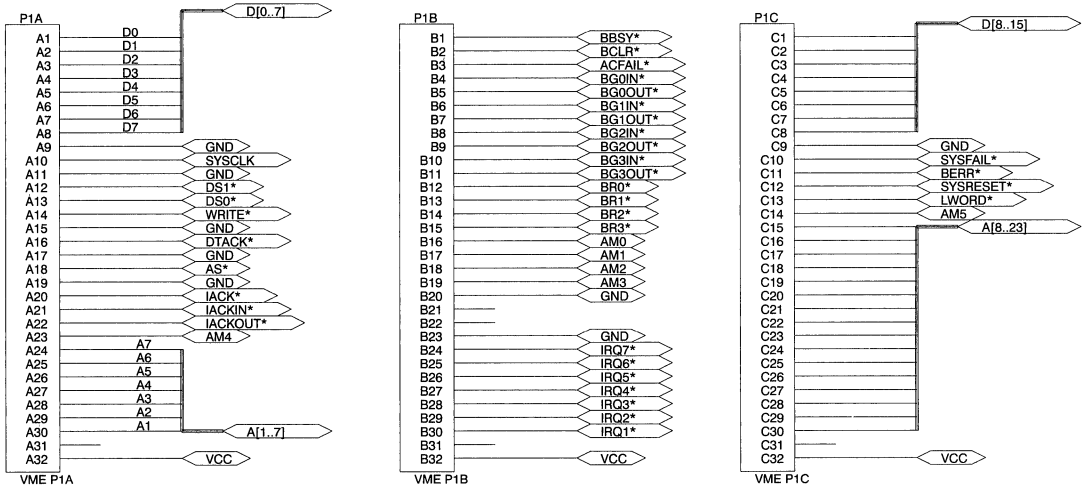
Appendix E. Schematics (continued)



Appendix E. Schematics (continued)

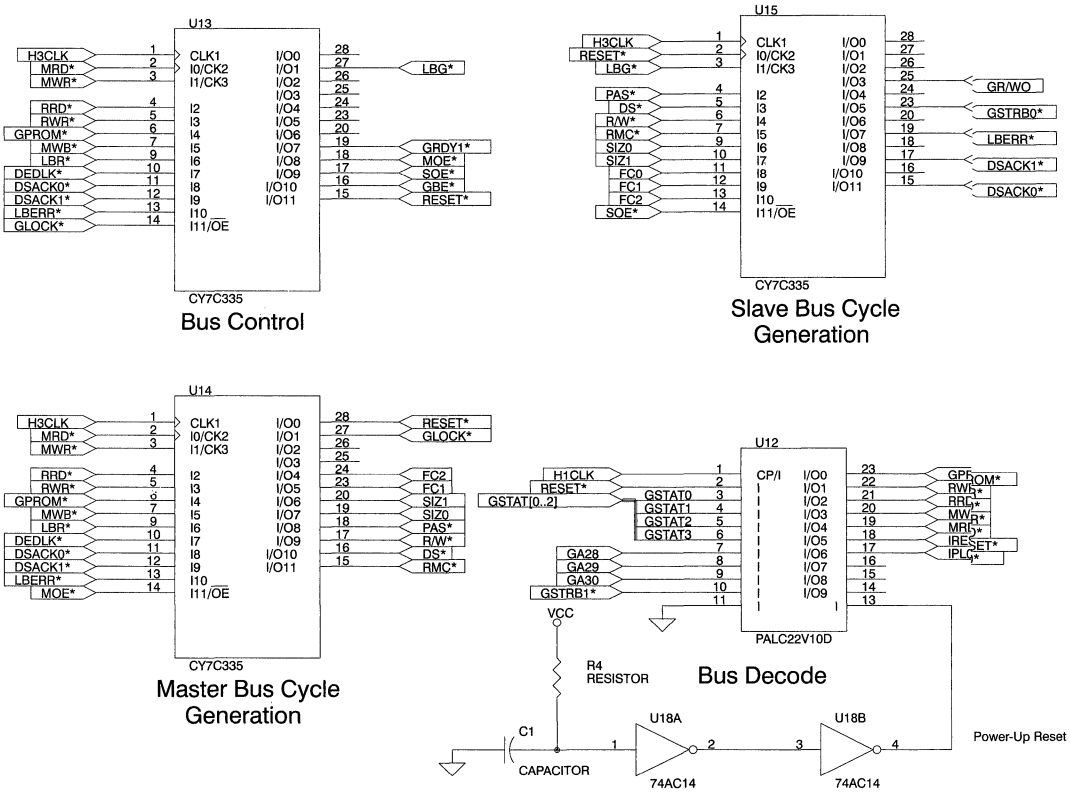


Appendix E. Schematics (continued)



VME P1 CONNECTOR

Appendix E. Schematics (continued)



Software Considerations for the VIC64

Introduction

This application note provides the VIC64 software developer with proven tips and examples for both configuring and operating the VIC64. The software described here is based on a SPARC-based VMEbus card utilizing a VIC64. This board was developed within Cypress Semiconductor as a test/evaluation vehicle for the VIC64 and the CY7C964.

This application note also discusses the configuration of the CY7C964 VMEbus address compare functions.

Although this application note specifically addresses the VIC64, virtually everything in this application note could also be applied to the VIC068A. VIC64-only features are flagged to notify the reader of items that are not applicable to the VIC068A.

The source files vic.h, eval_bd.h, and blt_cmd.c, which are described in this application note, are available through the Cypress Semiconductor BBS (Bulletin Board System). These files are contained within a file named "SAMPCODE.EXE."

Related Documents

The reader may also wish to consult the following documents for additional information:

- *VIC068A/VAC068A User's Guide*
- *VIC64/CY7C964 Design Notes*

These documents are available through your local Cypress Semiconductor field sales office.

Hardware Overview

The examples in this application note are based on an actual design of a SPARC-based VIC64 evaluation VMEbus board developed by Cypress Semiconductor. The following paragraphs provide background for this hardware platform. Contact your local field applications engineer regarding specific hardware information on this board.

The Evaluation Board

This evaluation board includes the following features:

- Cypress's CY7C611 embedded SPARC microprocessor
- Floating-point support
- 64 Kbytes to 4 Mbytes of private SRAM
- 64 Kbytes to 2 Mbytes of shared SRAM
- 512 Kbytes of EPROM for the embedded monitor program
- Performs D64 VMEbus transfers utilizing VIC64 and CY7C964 devices
- MC68681 DUART
- 2 Kbytes of non-volatile storage
- Real-time clock

Evaluation Board Local Control Register (LCR)

The evaluation board contains a single 32-bit, dual-purpose control register. When read, this register provides the memory size of the SIMM sockets as shown in *Table 1.*

Table 1. LCR Read Fields

Bits	Socket
bits 0,1	SIMM socket 1 size (private)
bits 2,3	SIMM socket 2 size (private)
bits 4,5	SIMM socket 3 size (private)
bits 6,7	SIMM socket 4 size (private)
bits 8,9	SIMM socket 5 size (shared)
bits 10,11	SIMM socket 6 size (shared)

The two bits for each SIMM contain one of the codes shown in *Table 2.*

Table 2. SIMM Size Codes

Code	Size
00	1M-byte SIMM
01	256K-byte SIMM
10	64K-bytes SIMM
11	Socket empty

When written, this register provides control over the the resources shown in *Table 3.*

Table 3. LCR Write Fields

Bits	Function
bits 0–15:	LEDs (lit when bit is clear)
bits 16,17:	VIC64 reset
bits 18–28:	VMEbus address (A31:21)
bits 29,30:	VMEbus address size (ASIZ0–1)
bit 31:	VMEbus data port size (WORD*)

Bits 0–15 provide control of 16 LEDs located on the edge of the board. When a bit is cleared, the corresponding LED is lit.

Bits 16 and 17 provide control over the reset operations of the VIC64. When bit 16 is cleared, the board's state logic asserts the IRESET* signal of the VIC64. When bit 17 is cleared, state logic asserts the IPL0* signal of the VIC64, issuing a global reset to the VIC64.

Bits 18–28 provide control over the most-significant eleven VMEbus address lines. Prior to a VMEbus access, this bit field is loaded with the most-significant eleven address bits. An access is then made to a predefined address (VME_BASE_ADRS) with the least-significant 21 VMEbus address lines obtained from the physical address of the transaction.

Bits 29 and 30 control the VIC64 ASIZ0/1 signals respectively. These signals tell the VIC64 what address size to use.

Bit 31 controls the WORD* signal line. When clear, the VIC64 performs D16 VMEbus accesses; when set, D32.

Software Considerations

SPARCmon™

The embedded monitor program used on the evaluation board is *SPARCmon. SPARCmon* is a commercial product available from Sun Microsystems. *SPARCmon* consists of source code modules for initialization, trap handling, floating-point support, process control, remote debugging, I/O, and a main command interpreter. Board-specific code such as board initialization, test, and additional commands are incorporated into *SPARCmon* separately. This application note does not address the specifics of *SPARCmon*, only board-specific details as it relates to the VIC64.

Boot-Up

The flow of initialization for booting the evaluation board is described in the following sections.

Disable Traps

Traps are disabled until resources exist to service them.

Initialize 7C611 Window Invalid Mask (WIM) and Trap Base Register (TBR)

Reset the VIC64

This is discussed in detail later in this application note.

Test First 64 Kbytes of Private Memory

This provides us with tested memory for temporary storage to perform subsequent boot tasks.

Set Up Initial Stack Frame Pointer and Enable Traps

With the first 64 Kbytes of memory tested, we may now service traps. The trap vector table is located initially in EPROM at address $0.

Initialize I/O

This consists of setting up I/O tables, structures and the DUART itself.

Perform Board Diagnostics

The remainder of the board is checked, including

- EPROM checksum
- NVRAM checksum
- Determining amount of SRAM installed
- Testing remaining private SRAM
- Testing the shared SRAM
- Testing the NVRAM
- Testing the VIC64 (discussed later)
- Testing the DUART
- Configuring board local memory map

Local memory map is created with regions for

- Monitor variables (DATA)
- Uninitialized monitor variables (BSS)
- The relocated trap table
- User memory area
- User stack (STACK) area

Clear User Memory Areas

The user areas are "cleared" to a predefined value.

Relocate the Trap Table in EPROM to SRAM

This speeds up trap table accesses and makes the table modifiable. The TBR is adjusted after the table is moved.

Configure VIC64

This is discussed in detail later in this application note.

VIC64 Initialization and Test

VIC64 Register Accesses

All of the VIC64's internal registers are 8 bits wide but occupy 32 bits of address space. Specific address and size information must be presented to the VIC64 in order for the VIC64 to accept the register access.

When the VIC64 has been selected for a register access (CS*, PAS*, and DS* are asserted to the VIC64), the VIC64 checks the SIZ1/0 and LA[1:0] signals to insure proper byte orientation. This is because the VIC64 is only connected to the lower 8 data lines of the local data bus and the data must be aligned as such.

Table 4 shows the valid combinations of SIZ1/0 and LA[1:0] that must be present for the VIC64 to accept the register access. The VIC64 mimics the Motorola CISC processors in that the SIZ and LA combinations for it are the same as for the VIC64. The SIZ codes for the CY7C611 are not the same and translation circuitry is required.

Table 4. VIC64/068 D(7:0) Data Alignment

SIZ1	SIZ0	LA1	LA0	Size
0	1	1	1	Byte
1	0	1	0	Word
0	0	0	0	Longword
1	1	1	1	3-Byte

If *Table 4* is not satisfied, the VIC64 ignores the attempted cycle by not reading or writing the information and not acknowledging the cycle (does not assert DSACKi*).

VIC64 Reset

The evaluation board issues a power-on reset to the VIC64 via the LCR. The LCR contains two bits for VIC64 reset. Bit 16 controls the assertion of IRESET* for the purposes of performing a internal reset. Bit 17 controls the assertion of IPL0*, which is used in conjunction with IRESET*, to perform a global reset. The VIC64 requires that a global reset be issued at power-up. The SPARC assembler code in *Figure 1* performs a VIC64 global reset.

This routine is written in assembler language because it must be a "leaf" routine. That is, it must not use the stack in any way since no stack exists yet. Calls from a high-level language or calling an additional routine would almost certainly use the stack.

Notice that the VIC64 is reset in stages. First the IRESET* signal is asserted to the VIC64 by clearing bit 16 of the LCR. The next instruction clears bit 17 to assert IPL0*. The reason that these are performed in separate instructions is that sufficient time must be allowed for the assertion of IRESET* to switch the IPL0* from an output to an input. Next, the IPL0* signal is removed, then the IRESET* signal is removed, in separate instructions. This is done to insure that the VIC64 200-ms reset timeout is observed. If they were removed simultaneously, this timeout may not be observed and the reset would complete immediately. Refer to section 12.1 of the *VIC068/VAC068 User's Guide* for more details on VIC reset.

VIC64 Test

To determine if the VIC64 is present and has been reset properly, the VIC64 test routine performs write-read-verify cycles to the VIC64 ICR0–5 registers. At this time, the VIC64 version register is read to determine the mask revision. The mask register reads $00, and any VIC64 values above $F0 indicate a VIC068 is installed. This may or may not be acceptable for specific applications.

VIC64 Configuration

The configuration of the VIC64 is accomplished by writing the VIC64 registers to desired values. The board stores these predetermined values as a structure located in the NVRAM at boot-up. The VIC64 configuration routine reads these values and stores them into the appropriate VIC64 registers. This way, the configuration of the VIC64 is not hard-coded and may be modified by simply changing the values in NVRAM and calling a VIC64 configuration routine.

VIC64 Address Spaces

In VMEbus systems, each VMEbus board typically has its own unique address spaces within the total 4-Gbyte VMEbus addressing range. These regions may consist of various sub-regions including:

- A32, A24, and/or A16 regions
- D32 and/or D16 regions
- Interprocessor communication regions

In addition to the VMEbus address spaces, the local processor within each board works with a local address space that may include:

- Private memory
- Shared memory (shared with the VMEbus)
- UARTs
- Interrupt acknowledge
- Board control registers

```
#include <eval_bd.h>                    ; Needed for LCR pointer

set LOCAL_CONTROL_BASE_ADRS, %l6        ; This symbol points to the LCR
set 0xffffffff, %l2
st %l2, [%l6]                           ; "clear" LCR
set 0xfff7ffff, %l2
st %l2, [%l6]                           ; Assert IRESET*
set 0xfffcffff, %l2
st %l2, [%l6]                           ; Assert IPL0*
set 0xfff7ffff, %l2
st %l2, [%l6]                           ; Remove IPL0*
set 0xffffffff, %l2
st %l2, [%l6]                           ; Remove IRESET*
```

Figure 1. VIC64 Reset

- The VMEbus
- Control registers (including VIC64)

These local areas may or may not be visible to other VMEbus modules. It is not uncommon for shared memory to be the only local resource available to other VMEbus modules. This is the case for this board. The local addresses and the VMEbus addresses to this shared memory would almost certainly be different. Some type of secondary decode or address translation is necessary in these instances. In the examples given in this application note, the header file eval_bd.h defines the local address map used for the board.

The VIC64 does not directly support VMEbus accesses to their internal registers with the exception of the Interprocessor Communication registers. It is possible via external hardware to make all VIC64 registers visible to the VMEbus (see Cypress's application note titled "Using the VIC068A Without a Processor"). If a VAC068A is used, the local VIC64 (the VAC068A is not compatible with the D64 operations of the VIC64, but can be used if D64 operation are not performed) register region is fixed at addresses FFFCxxxx to FFFDxxxx. As a minimum, sufficient space must always be allotted for the 58 longwords of VIC64 registers.

VMEbus addressing through the LCR

As noted earlier, bits 18–28 of the LCR provide control over the most significant eleven VMEbus address lines. Therefore, a VMEbus access may consist of two parts: loading the LCR with the proper value, and performing the actual transfer to the VMEbus address location. This location consists of a fixed address in combination with the lower 21 bits of the VMEbus address.

As an example, assume a VMEbus A32, D32 read access is desired from the VMEbus address 0x38004000 and that the LEDs should remain clear (see *Figure 2*). A value of 0xA703FFFF should be written into the LCR. If the VMEbus address space on the local address map is 0xE00000 (VME_BASE_ADRS), the local address should be 0xE04000 (0xE00000 + least significant 21 bits of VMEbus address).

An addressing scheme of this sort makes the entire 4-Gbyte range of the VMEbus addressable by the board. A disadvantage is that the LCR must be written for any VMEbus transaction is in a different 2-Mbyte address spaces from the previous VMEbus transaction. An example of a function that would return the proper address is shown in *Figure 3*.

An example function that returns the proper LCR value could be as shown in *Figure 4*.

CY7C964 Address Comparator Configuration

The evaluation board uses the CY7C964 as the VMEbus slave address comparator. The address comparator consists of two registers: the mask register and the compare register. The compare register is loaded with the base address of the slave address. The mask register is loaded with a value that determines which bits of the address should be compared with the value in the compare register. This defines the size of the address region. A zero in a bit enables the comparison of the corresponding bit in the compare register to the VMEbus address bit.

For example, if there are 4 Mbytes of shared memory and the VMEbus slave range is to start at address 0xC00000, the following values should be loaded into the CY7C964 registers:

Compare Register:	0x00C00000
Mask Register:	0x003FFFFF

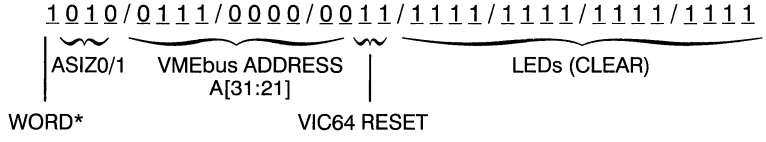


Figure 2. VMEbus A32, D32 Read Access

```
/* eval_bd.h includes the following:
   typedef unsigned int WORD
   #define VME_BASE_ADRS 0xE00000 */

#include <eval_bd.h>

#define VMEADRSMASK 0x001FFFFF

WORD *CalcVMEadrs (adrs)
WORD *adrs
{
WORD VMEadrs;

VMEadrs = (WORD) adrs;
VMEadrs &= VMEADRSMASK;       /* mask off upper 11 bits of address */
VMEadrs |= VME_BASE_ADRS;     /* overlay VMEbus address for evaluation board */

return ((WORD *) VMEadrs);
}
```

Figure 3. VMEbus Address Calculation

```
/* eval_bd.h includes the following:
   typedef unsigned int WORD */

#include <eval_bd.h>

#define LCRADRSMASK 0xFFE00000
#define LCRMASK 0xE003FFFF
#define LCRSHIFT 3

WORD *CalcLCR (adrs, LCReg)
WORD *adrs, LCReg;
{
WORD TempAdrs;

TempAdrs = (WORD) adrs;     /* convert WORD pointer to WORD */
TempAdrs &= LCRADRSMASK;    /* mask off lower 21 address bits */
TempAdrs >>= LCRSHIFT;      /* shift over by 3 */
LCReg &= LCRAMSK;           /* clear out existing address in LCReg */
LCReg |= TempAdrs;          /* overlay new address onto LCReg */

return (LCReg);
}
```

Figure 4. LCR VMEbus Address Calculation

```
/* NO!!! */
WORD *VMEadrs = (WORD *) 0x400000;

/* Yes!!! */
WORD *VMEadrs;
VMEadrs = (WORD *) 0x400000;
```

Figure 5. Proper Variable Initialization

Compiling Considerations

Because the monitor used for the evaluation board is EPROM-based, certain considerations are noted, namely:

1. All monitor sections that can be read-only are linked such that they occupy a contiguous section of EPROM. This may be done with the −R option of a UNIX *cc* compiler. The −R option merges the code segment TEXT with the initialized data segment DATA.

2. Because the DATA segment is now located in EPROM, any initialized data is now read-only and is not modifiable. This suggests that variable declarations do not initialize the variable, as shown in *Figure 5*.

3. The uninitialized data segment BSS and the stack segment STACK must be located in RAM.

Example VIC64 Software Building Blocks

The following are examples of code that were used for the VIC64-specific routines on the board.

vic.h

vic.h is a header file that defines useful macros and VIC64-register-related constants. First, the macro VIC is defined, which returns an address to a VIC64 register. The argument to this macro is the number of the register. These numbers start from 0 (VIICR) and end with 57 (BTLR2) for the VIC64 (56 for the VIC068). These numbers are not the address of the register. Next, constants are defined that assign these numbers to the register names themselves. And lastly, a unique VIC64 register identifier is given to each register so that its address and contents can be obtained directly. A similar macro is defined for setting and clearing the Interprocessor Communication (IPC) switches. This IPC macro needs, as an argument, the starting address of the VMEbus IPC areas of interest.

As examples, consider the code fragment shown in *Figure 6*, which illustrates the VIC_xxx macros.

In addition, numerous other constants are defined that aid in manipulating the various bit fields within the registers themselves. These constants are separated by register. Also, the last character of the constant name may consist of a underscore (_) or lower case letters that indicate something about the constant or the bits. *Table 5* summarizes these characters.

```
#include <vic.h>              /* VIC macros located here */
#include <eval_bd.h>          /* typedef for BYTE (unsigned char) */

BYTE      TempStorage;
BYTE      *TempStoragePtr;

TempStorage = *VIC_BTCR;      /* read contents of BTCR */
*VIC_SS0CR0 = TempStorage;    /* store contents of SS0CR0 */
TempStoragePtr = VIC_TTR;     /* read pointer to TTR */
ICF_ICGS0_SET (ICF_BASE);     /* set ICGS0 */
```

Figure 6. Using the "VIC" Macros

Table 5. *vic.h* Constant Preceders

Suffix	Meaning
_	Implies a bit field which is cleared
r	Implies read-only bit(s)
m	Implies a masking value for bit(s)

eval_bd.h

eval_bd.h is a header file that contains board-specific constants. These constants also include the local address map of the board, including those resources described in *Table 6*.

In addition, other types and constants are defined, including individual DUART registers, power-of-2 constants, byte-extraction macros, and some NVRAM macros.

A Generic Block Transfer Utility

blt_cmd is a generic, command-line driven program that enables the user to perform almost every conceivable block transfer operation using the VIC64 or the VIC068. One notable exception is allowing the VIC64 to interrupt when the block transfer is complete. blt_cmd is meant mainly to be used as a vehicle for board and code testing.

Configuration is provided by the command-line arguments outlined in *Table 7*.

Table 6. Local Address Symbols

Memory Area	Privilege	Symbol
EPROM	Read/Write	ROM_BASE_ADDRESS
Status Register (LCR)	Read-Only	STATUS1_BASE_ADRS
Control Register (LCR)	Write-Only	LOCAL_CONTROL_BASE_ADRS
DUART	Read/Write	M68681_BASE_ADRS
NVRAM	Read/Write	NVRAM_BASE_ADRS
7C964 Mask Register	Write-Only	BILC_M_BASE_ADRS
7C964 Compare Register	Write-Only	BILC_C_BASE_ADRS
Interrupt Acknowledge	Read-Only	INT_ACK_BASE_ADRS
VIC64	Read/Write	VIC_BASE_ADRS
VMEbus	Read/Write	VME_BASE_ADRS
Private SRAM	Read/Write	BANK1_BASE_ADRS
Shared SRAM	Read/Write	BANK2_BASE_ADRS

Table 7. Command-Line Arguments

Argument	Default[1]	Function
−6		Performs D64 transfers (requires VIC64 device).
−3	√	Performs D32 transfers.
−a[address]	0xC00000	Sets local starting address for which data will be read, for VMEbus write block transfers or written for VMEbus read block transfers to *address*.
−A[value]	Disabled	Sets user-defined AM code that is to used for block transfers to *value*.
−b[value]	0x200	Sets minimum value for byte count to *value*. If the −ib value is 0 (increment byte count) the fixed byte will be set to *value*.
−B[value]	0xFFFC	Sets maximum value for byte count to *value*. Not used if −ib value is set to 0.
−cl		Enables local boundary crossing.
−cL	√	Disables local boundary crossing.
−ct		Enables 2-kbyte VMEbus boundary crossing (implies −cv).
−cT	√	Disables 2-kbyte VMEbus boundary crossing.
−cv	√	Enables VMEbus boundary crossing.
−cV		Disables VMEbus boundary crossing.
−d		Enables the dual-path option but does not perform interleave master cycles (see −p).
−D	√	Disables the dual-path option.
−e		Sets the release mode to RWD.
−E	√	Sets the release mode to ROR.
−f		Enables DRAM refresh.
−F	√	Disables DRAM refresh.
−ib[value]	0	Set the byte count increment value to: *value* * size of the operand.
−ii[value]	0	Sets the interleave increment value to *value.*
−iu[value]	0	Sets the burst count increment value to *value.*
−i[value]	0	Sets minimum value for interleave to *value*. If the −ii value is 0 (increment increment count) the fixed interleave value will be set to *value*.
−I	0xF	Sets maximum value for interleave to *value*. Not used if −ii value is set to 0.
−k	√	Enables data set-up before every block transfer and data checking after every block transfer.
−K		Disables data set-up before every block transfer and data checking after every block transfer.

Note:

1. The check mark indicates the default of two preceding arguments.

Table 7. Command-Line Arguments (continued)

Argument	Default[1]	Function
−l[value]	1	Sets the number of block transfers to perform to *value.* If *value* is set to 0, program will loop forever.
−m		Enables the clearing of the BLT enable bit (BTCR[4]) during the first interleave (VIC64 only).
−M	√	Enables the clearing of the BLT enable bit (BTCR[4]) after the block transfer is completely finished.
−p		Enables the dual-path feature and performs VMEbus master cycles during the interleave period.
−P	√	Disables the performing of interleave master VMEbus cycles. Leaves the dual-path feature enabled.
−r	√	Enables BLT reads.
−R		Disables BLT reads.
−s[value]	3	Sets the VMEbus request level to *value.*
−t	√	Enables the "enhanced" BLT turbo mode (VIC64 only).
−T		Disables the "enhanced" BLT turbo mode.
−u[value]	0	Sets minimum value for the burst count to *value.* If the −iu value is 0 (increment burst count) the fixed burst count will be set to *value.*
−U[value]	0x3F	Sets maximum value for burst count to *value.* Not used if −iu value is set to 0.
−v[value]	0xDEADC0DE	Sets the value to which destination memory will be set to *value.*
−w	√	Enables BLT writes.
−W		Disables BLT writes.
−x		Restores all options to their default states.
[address(es)]	0x200000	VMEbus starting address(es) for block transfer. Up to five may be specified.

All mutually exclusive options are shown without a divider between the options. If two mutually exclusive options are defined, the last one in the command line will take precedence. The state of these options are saved in static variables such that once a configuration is entered, the whole command string will not have to be retyped. Only those options that need to be changed will have a new option. Using the −x option will restore all options to their default state.

Unsupplied Functions

blt_cmd.c contains one function, lib_atohex(), that is not supplied. It is a library routine supplied with the SPARCmon source. Any ASCII-to-hex converter could be used with small modifications to blt_cmd.c. lib_atohex() is outlined in *Figure 7.*

Program Flow

Figures 8, 9, and *10* illustrate the flow of blt_cmd.c.

Example Operations

The following examples show how blt_cmd can be used to initiate a variety of block transfers.

```
blt_cmd -l0 -6 -ii1 -aC800000 D800000
```

This command line would perform D64 read and write block transfers indefinitely using local address

```
#include <atohex.h>
/* needed for lib_atohex return
   values */

lib_atohex (string, hexvalue)
char *string;
unsigned long *hexvalue;

/*
inputs:
string    character to be converted
Outputs:
hexvalue  pointer to the hex result

Return value:
SUCCEEDED      valid number
(otherwise)    illegal hex number
*/
```

Figure 7. atohex() prototype

0xC800000 and VMEbus address 0xD800000. After each read/write block transfer, the interleave period is incremented by 1. All other options would remain at their default values.

```
blt_cmd -3 -W -iu1 D800000 E800000
```

This command line would perform D32 read block transfers indefinitely (−l0 still in effect) using local address 0xC800000 (defined last time) and VMEbus addresses 0xD800000 and 0xE800000. After each read block transfer, the burst count and the interleave period (still defined from last time) is incremented by 1. All other options would remain at their default values.

```
blt_cmd -6 -w -ib1 -K -p D800000
```

This command line would perform D64 read and write (writes are re-enabled with −w) block transfers indefinitely using local address 0xC800000 and VMEbus address 0xD800000. After each read/write block transfer, the byte count would be incremented by 8 (1 * 8 bytes/transfer). Data checking is suppressed. Master cycles are performed in the interleave period. All other options would remain at their default values.

```
blt_cmd
```

Performs block transfers using the same parameters as the last time invoked.

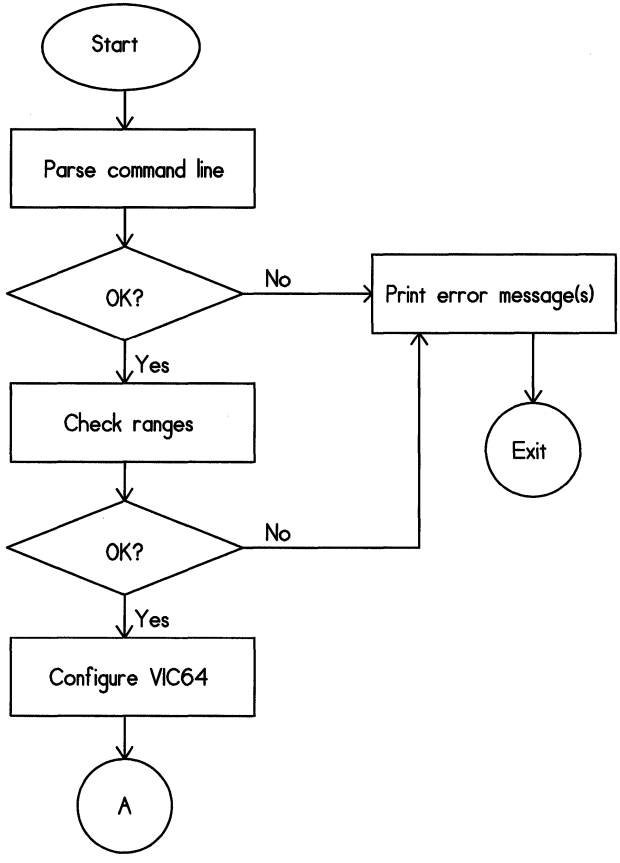


Figure 8. blt_cmd Flow

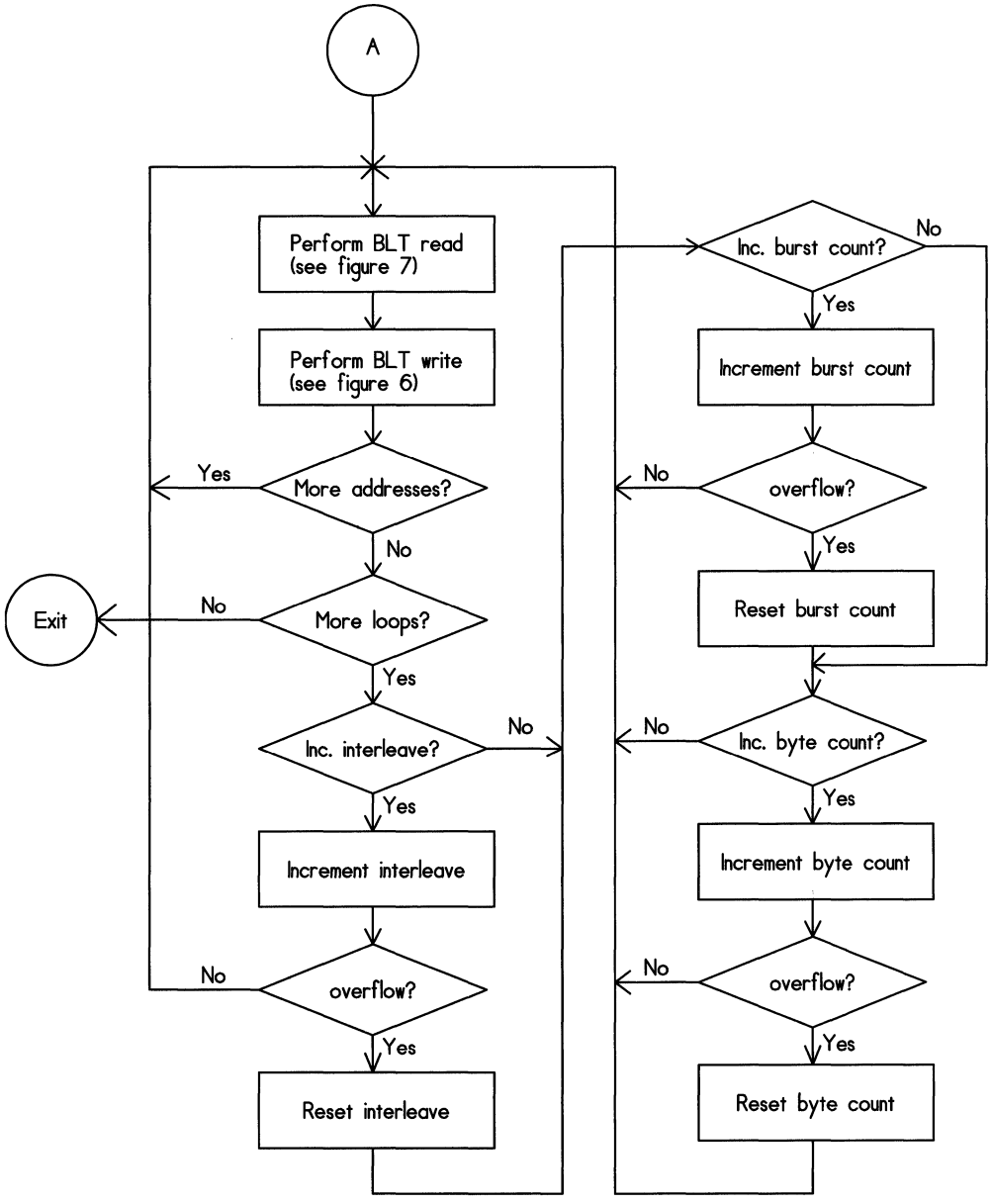


Figure 8. blt_cmd Flow (continued)

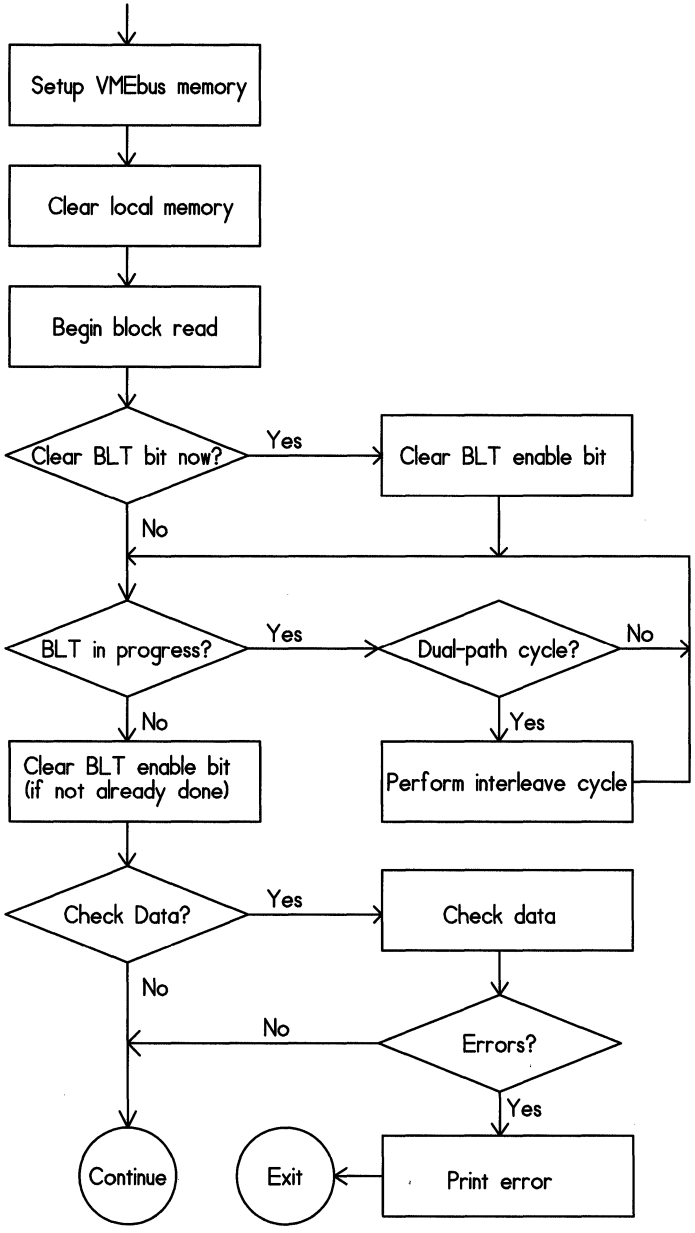


Figure 9. blt_cmd Read Flow

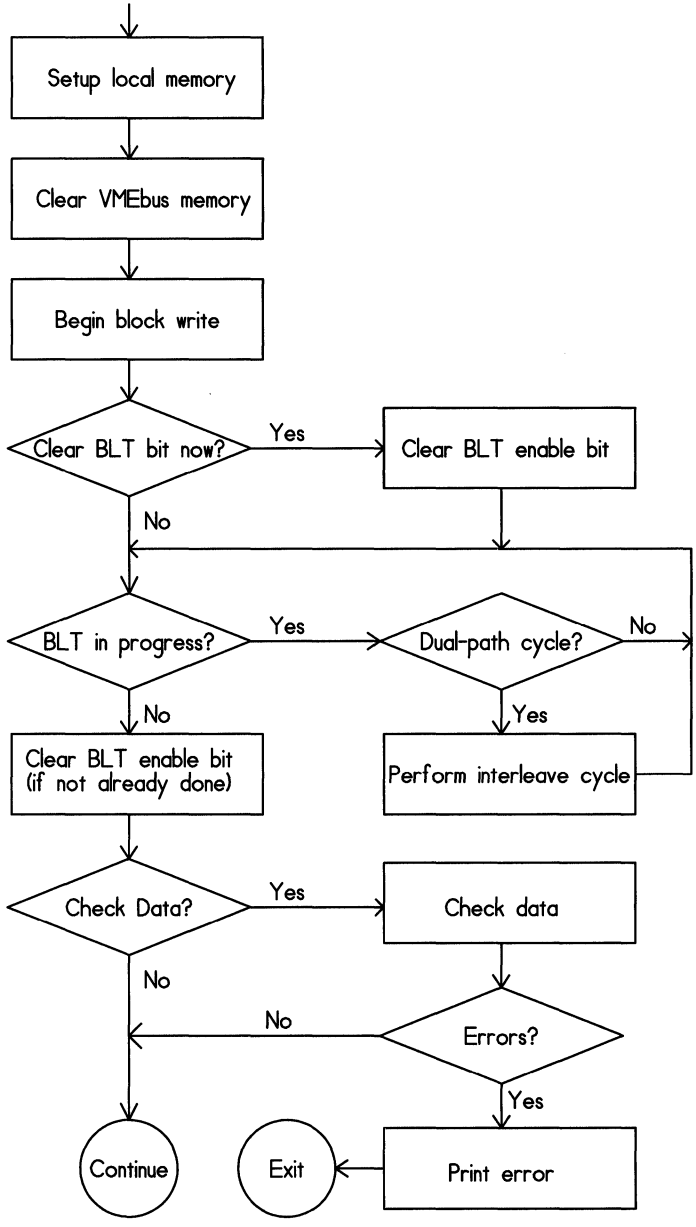


Figure 10. blt_cmd Write Flow

VIC64 to Motorola 68040 Interface

Purpose

This application note shows how the VIC64 can be interfaced to a Motorola 68040 microprocessor operating at 40 MHz. The issues and assumptions that go into designing such an interface are considerable and complex; thus, this application note will not attempt to design a complete VME board that can do everything. It will cover some of the issues that are pertinent when designing a 68040-based VMEbus board and will focus on the circuitry required for VIC64 to 68040 interfacing.

Design Issues

Asynchronous Bus (VIC64) to Synchronous Bus (68040) Interfacing

With the 68040 microprocessor, Motorola radically changed its bus architecture. With the 68030 and prior processors, Motorola used an asynchronous bus protocol. The 68040, on the other hand, uses a synchronous bus protocol. The VIC64, being an extension of the VIC068A architecture, retains the asynchronous bus protocol that is compatible with the 68030 and prior microprocessors. This makes the VIC64 and 68040 bus protocols incompatible.

For the most part, the VIC64 is a peripheral to the 68040. The 68040 generates read and write cycles to the VIC64 and the VIC64 responds. There is only one case where the 68040 would act as a peripheral to the VIC64 and that is if the 68040's snooping capability were turned on and the 68040 was required to supply data from its internal cache for a VIC64 cycle. To simplify the snooping interface, there are memory design strategies described later in this application note that can isolate memory accessed by the VIC64 from the 68040 internal cache. Thus, whenever the VIC64 were to act as master on the bus, the 68040 would never need to respond to a VIC64 cycle. No memory area that the VIC64 can access would be cached by the 68040.

To allow the 68040 and VIC64 to communicate, the VIC64 must be synchronized to the 68040. The primary signals that undergo this synchronization are the handshaking signals, DSACK0* and DSACK1*, that the VIC64 sends to the 68040 to indicate the completion of a register transfer or a VMEbus transfer.

Putting a "Slow" VIC64 on the 68040's Bus

The 68040 synchronous bus can transfer data at a rate of 1 transfer per 2 cycles of the 40-MHz bus clock when running in single-cycle mode. The transfer can either be a byte, word, or longword in length. This translates to 1 transfer every 50 ns. The VIC64 responds to a request for a data transfer to its internal registers no quicker than 67.5 ns. When the 68040 accesses the VMEbus via the VIC64, the transfer can be considerably slower since the VMEbus slave controls the progress of the transfer. To pace the transfer without losing data, the 68040 allows a slow peripheral to hold off on asserting $\overline{TA}$ until it has its data available on a read, or can accept data on a write. The interface designed in this application note synchronizes the DSACK1* and DSACK0* signals from the VIC64 and uses them to generate $\overline{TA}$ to the 68040.

Bus Contention – Peripheral Write after Read

When designing with a high-speed processor and a slow peripheral, bus contention is always a concern. Bus contention comes into play when a slow peripheral is being read by the processor in the current bus

cycle and in the next cycle, the processor executes a write. Typically, the slow peripheral cannot be disabled off of the bus before the processor begins driving the bus. The VIC64 to 68040 interface is no exception.

Figure 1 shows the timing of the contention. The VMEbus interface used in this application note is the full functional D64 VMEbus interface using the VIC64 and 3 CY7C964s as shown in *Figure 4* of the Cypress application note titled, "Using the CY7C964 with VIC." At the end of the 68040 cycle where data is read, it takes up to 5 ns for PAS*, DS*, and CS* to deassert (using a PALC22V10D−7), up to 23 ns from DS* deasserted to ISOBE* deasserted, up to 12 ns from ISOBE* deasserted to CISOBE deasserted, and then 7.5 ns for the '245 to disable assuming a 74FCT16245T is used. The next cycle can begin and write data can be presented to the bus as early as 5.25 ns after the BCLK following the cycle when PAS*, DS*, and CS* were deasserted. This creates over 15 ns of contention!

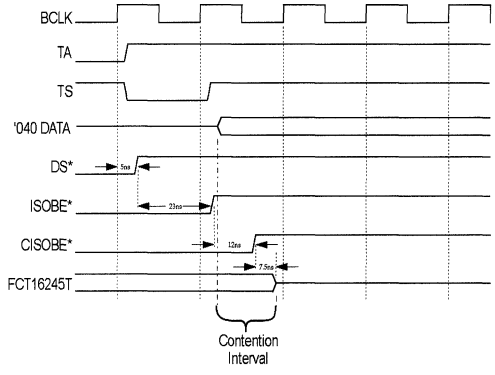


Figure 1. Contention for a Read Followed by a Write

Solving Bus Contention with Arbitration

The solution to the contention is easy considering the bus arbitration scheme of the 68040. In prior members of the 68k family, the processor also contained a bus arbiter on the same chip. Any peripheral that wanted to get access to the bus was required to request the bus from the processor. The 68040 relies on the designer to implement an external bus arbiter. All devices that can be masters on the bus must request the bus from the arbiter and the 68040 is no exception.

A way to eliminate the contention is to not allow the processor to begin a write cycle immediately after it has read the VMEbus or the VIC64 registers. The arbitration states of the 68040 make this possible. The timing of the arbitration is shown in *Figure 2*. At the beginning of the read cycle, the 68040 asserts $\overline{TS}$ along with an address that indicates either a VMEbus cycle or a VIC64 register access. The progression then is as follows:

1. The address is decoded and CS*, STROBE*, or MWB* is asserted along with PAS*.
2. The arbiter deasserts $\overline{BG}$ in response to CS*, STROBE*, or MWB* assertion. The 68040 will complete its current cycle. It is assumed that the 68040 does not want to relinquish the bus and will continue to drive $\overline{BR}$ asserted.
3. After receiving $\overline{TA}$, the 68040 is forced off the bus since the $\overline{BG}$ signal had been previously deasserted. However, when the arbiter sees that

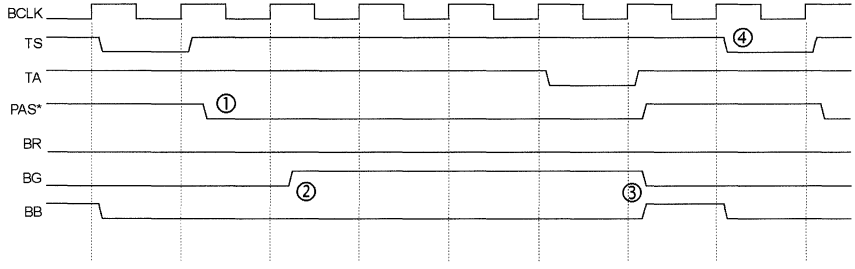


Figure 2. Arbitration Used to Eliminate Contention

$\overline{TA}$ has been asserted it grants the bus back to the 68040.

4. The 68040 on the next clock rising edge can assert a $\overline{TS}$ to begin the new cycle since $\overline{BG}$ is seen asserted.

With this method, there is no possibility of contention since 25 ns has been added to the contention resolution time.

For this method to be effective, good board layout and decoupling must be used. Taking the bus away from the 68040 causes its bus buffers to go high-impedance and then low-impedance in a single bus cycle. This can cause significant ground bounce and noise if the proper design practices are not used. Also, the signals that go high-impedance must be pulled up to V_{CC} to prevent them from floating.

Slave Access Implementation

Regardless of the memory map of the board, there are common issues pertaining to slave access of the board from the VMEbus. The slave interface is highly dependent on the function of the board. If the board is a memory array, chances are the board will primarily be accessed as a slave. However, if the board is a general purpose microprocessor, it will probably spend most of its time as a master on the VMEbus.

Since the slave interface is variable from board to board, the details of a slave interface to onboard circuitry will not be covered. The information in the *VIC068A/VAC068A User's Guide* and the *VIC64/CY7C964 Design Notes* contain ample information on using the VIC64 and CY7C964s for slave accesses. The next three sections address issues necessary for designing the slave circuitry on the board.

Bus Snooping

The 68040 can be configured to snoop cycles on its bus when it is not a master. Snooping is only a concern if the 68040 and the VIC64 share a common memory subsystem. If the VIC64 has its own dedicated memory which is gated off from the 68040's memory, snooping is not an issue (unless, of course, multiple bus masters reside on the bus with the 68040).

When the 68040 finds a cycle that requires data to be supplied from its internal cache, it will inhibit the memory subsystem and provide the requested data. The timing of this operation is synchronous to the BCLK and thus, if snooping were configured, the VIC64, when acting as a bus master, must have its signals synchronized to properly meet the 68040 timing.

Inhibiting Cache Transfers From Shared Memory

To avoid the timing difficulties that arise when snooping is enabled with a common memory subsystem, snooping can be disabled! This would also require that data areas on the board accessed by the VIC64 cannot be cached internally by the 68040. To disable caching of VIC64 register data or read VMEbus data, and disable snooping, accesses to the VIC64 and CY7C964 circuitry that generate STROBE*, CS*, and/or MWB* would cause $\overline{TCI}$, SC0, and SC1 signals to go to the 68040 in their inhibiting states. This would disallow the current cycle from being cached internal to the 68040.

To prevent the caching of data written to the board when the VIC64 is acting as a slave or a block transfer controller, the 68040's memory map decoder must assert $\overline{TCI}$ when any location the VIC64 can access when in that mode is requested by the 68040.

Memory Map Decoding and Remapping

Another design issue when implementing slave access logic is that of memory map decoding and remapping. When an address is provided from the VMEbus, it may not correspond to the same physical address on the board. Through the use of PLDs for decoding and shifting addresses, the VMEbus address can map to an on-board address.

Design Assumptions

Other than the design issues covered above, there are two assumptions that have been made in the design of the circuits herein. The first pertains to the memory system design and the second pertains to the buffer-type selection of the 68040.

Memory System Design

The goal in any memory system design is to match the performance of the memory to the masters that access it. This presents a problem in the design since the 68040 and VIC64 have vastly different bus structures. The 68040 is based on a synchronous bus and supports high-speed burst transfers as well as single-cycle transfers with all data and control signals synchronized to a common bus clock (BCLK). However, the VIC64 relies on asynchronous bus transfers that are paced by asynchronous data accesses and acknowledgements. There must be an assumption made by the board designer of one of the following memory strategies. Based on the typical application of the board, the designer can select a memory strategy to maximize data throughput. Two designs are presented here but many more are possible. In each case, the block labeled "VME Interface" contains the circuit shown in *Figure 4* of the Cypress application note titled, "Using the CY7C964 with VIC."

Two Memory Banks Architecture with No Caching of Shared Bank

Figure 3 shows a memory system design that is split into two separate banks. The first bank of memory is dedicated to the 68040 and runs synchronously. The second bank of memory is dedicated to the VIC64 and runs asynchronously. By having two separate memory banks, each can be designed to run optimally with its corresponding bus master. This would offer the best performance for the 68040 for its burst mode, and for the VIC64 for its burst mode. The gate between the two memory buses allows the 68040 access to the VIC's memory and to the VME-bus for single-cycle transfers. Access to the VIC64's memory bus is controlled by the arbiter and is granted to the 68040 when the VIC64 is not active on its bus.

Under normal operation, the gate opens when requested by the 68040, allowing the 68040 free access onto the VIC64's memory bus and onto the VME-bus. Only when the VIC64 is accessed as a slave or it is controlling burst transfers would the gate be closed. The VIC64 would request access to its bus via its LBR* signal. A memory configuration like this would allow both the VIC64 and the 68040 the most bandwidth on their respective busses. Both could be operating as bus masters at the same time. The application note titled "Interfacing the CY7C611A with the VIC64" uses this type of

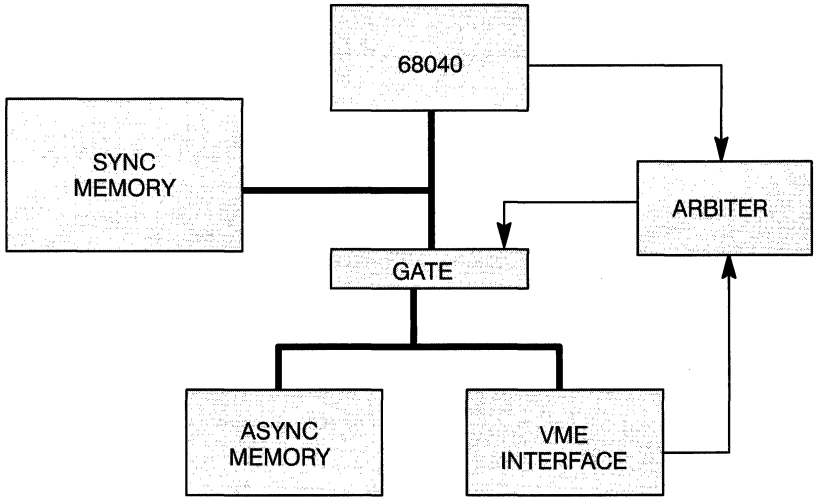


Figure 3. Two Memory Banks Architecture

memory scheme. The only caveat with this type of memory scheme is that when the 68040 is accessing the VIC64's memory, the data and acknowledgements from the VIC64 or its memory must be synchronized to the 68040's bus requirements.

Shared Memory with No Caching of VME Area

The other type of memory subsystem would be one that can act synchronously or asynchronously depending on whether the 68040 or the VIC64 was on the bus. This is illustrated in *Figure 4*. Both the 68040 and the VIC64 would share the same address and data buses and the arbiter would be used to grant access to one or the other. The arbiter could also indicate to the memory subsystem who has access to the bus. Although this simplifies the bus structure, it could complicate the memory design. It could also limit the bandwidth of the 68040 and the VIC64 to unacceptable levels.

However, this memory design might be perfectly acceptable for certain applications. The VIC068A and earlier 68K-family processors were able to share the same bus due to their compatible bus structures. Many designs allowed both the VIC64 and, for example, a 68020 to share bus bandwidth without detrimental effects. It is assumed that since this design revolves around a 68040 at 40 MHz, bus bandwidth for the processor is important! For this application note, it will be assumed that the separate memories strategy is used.

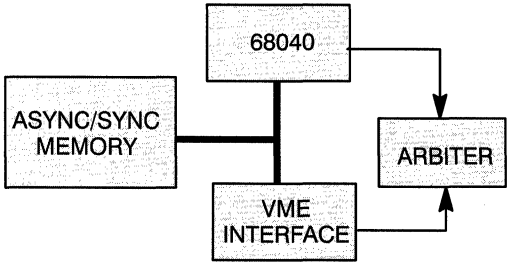


Figure 4. Single Memory Bank Architecture

68040 Configured for Large Buffer Timing Mode

To simplify the timing analysis and insure the peak performance from the design, the 68040 will be used in its Large Buffer Timing mode. This will require the careful layout of the board and the use of signal terminations to prevent adverse results from transmission line effects. Large Buffer mode is entered into during processor reset by pulling the $\overline{IPL2}$, $\overline{IPL1}$, and $\overline{IPL0}$ signals to a logic-one state.

Reset Circuitry

The reset circuitry and its routing is shown in *Figure 5*. There are three possible sources for a reset in this design. The first is a power-up or front panel pushbutton reset. The second is a reset initiated by the VIC64. The third is a 68040-initiated reset. The Reset PLD, a CY7C335-83, controls the sequencing for each of these reset types. The VHDL code describing this PLD is given in Appendix A.

Power-Up or Pushbutton Reset

The timing for the power-up or pushbutton reset is shown in *Figure 6*. While PWRUP_RST_N is LOW from either the pushbutton being depressed or the capacitor in *Figure 5* charging at power-up, BRD_RST_N_OUT is LOW. The capacitor and resistor values are chosen to guarantee that the clock and the board V_{CC} are stable when the rising edge of PWR_RST_N occurs. This insures that the VIC64 will be reset properly with a global reset. When the rising edge of PWRUP_RST_N occurs, the IRESET* signal is pulled LOW to the VIC64. The VIC64 responds with RESET* LOW, which in turn causes IPL0* to be pulled LOW, thus beginning a global reset. The IPL0* signal is then returned HIGH and, after a delay, IRESET* and BRD_RST_N_OUT are brought HIGH, ending the reset.

68040 Mode Selection

The 68040 is reset via the $\overline{RSTI}$ signal. For a valid reset to occur, the $\overline{RSTI}$ signal must be held LOW for a minimum of 10 BCLK cycles. The operation of the Reset PLD guarantees that $\overline{RSTI}$ will be held LOW for greater than this minimum amount of time. On the rising edge of $\overline{RSTI}$, the 68040 reads

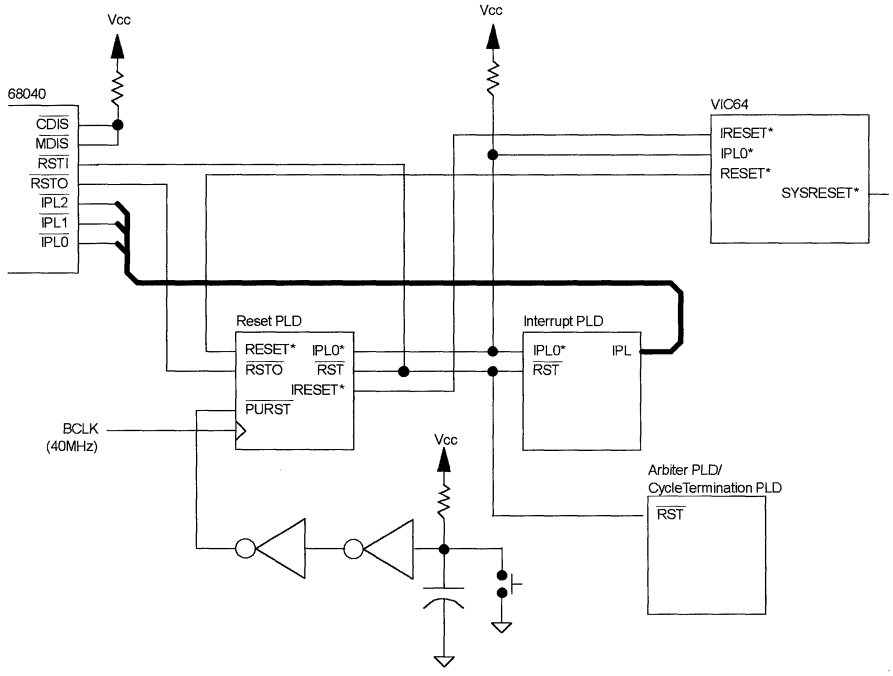


Figure 5. Reset Circuitry

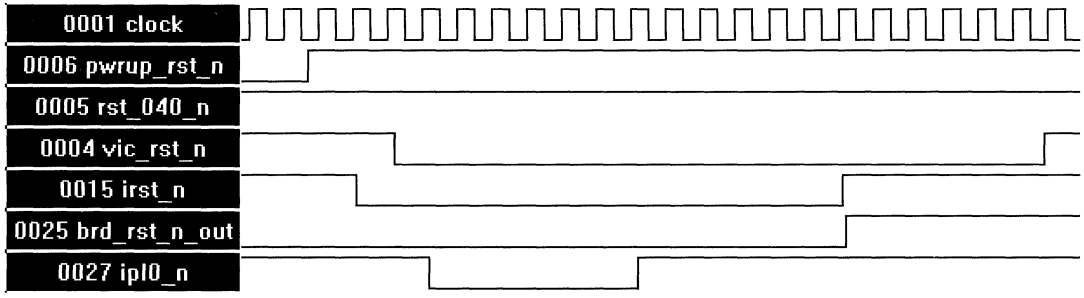


Figure 6. Power-Up or Pushbutton Reset

the current state of the $\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$, $\overline{MDIS}$, and $\overline{CDIS}$ and sets the mode of operation of the 68040.

The $\overline{CDIS}$ and $\overline{MDIS}$ signals are both pulled HIGH through a resistor that, at reset, disables Multiplexed Bus mode and Data Latch Enable mode. During normal operation, pulling $\overline{CDIS}$ and $\overline{MDIS}$ HIGH enables the internal cache of the 68040 and enables its internal MMU. The $\overline{IPLx}$ signals are all pulled HIGH at reset also via the Interrupt PLD. This enables the Large Buffer Timing mode for the data, address, and control signals.

VIC-Initiated Reset (SYSRESET* Active or SRCR Written)

The timing for a VIC-initiated reset is shown in *Figure 7.* A reset from the VIC is caused by one of two events. If the SYSRESET* signal on the VMEbus is driven active, the VIC64 will respond with the RESET* driven active. The VIC64 will also issue a RESET* if the SRR ($E3) is written with a value of $F0. This will cause the Reset PLD to force a full board reset via the BRD_RST_N_OUT signal and a global reset to the VIC64 with the IPL0* and IRESET* signals.

Support for 68040 RESET Instruction

The 68040 has an instruction, RESET, that forces its $\overline{\text{RSTO}}$ signal LOW for 512 BCLK cycles. The internal state of the 68040 is unaffected during this interval, which makes this instruction good for resetting board peripherals during normal processor operation. The implementation in this design, however, forces the board to be reset when the $\overline{\text{RSTO}}$ signal is activated. When the 68040 sees the $\overline{\text{RSTI}}$ signal active during an $\overline{\text{RSTO}}$ LOW interval, it immediately negates $\overline{\text{RSTO}}$ and forces a processor reset. The timing of this reset is shown in *Figure 8.*

Bus Arbitration

Bus Arbitration State Machine

The state machine for the bus arbitration is shown in *Figure 9.* There are essentially three arbitration states in the machine with a fourth being the reset state. The task of the arbiter is to grant access to the VIC64's private bus (*Figure 3*). Thus, it will normally allow the 68040's $\overline{\text{BG}}$ signal to remain active at all times. In fact, the arbiter does not even consider the state of the $\overline{\text{BR}}$ signal from the 68040 in the arbitra-

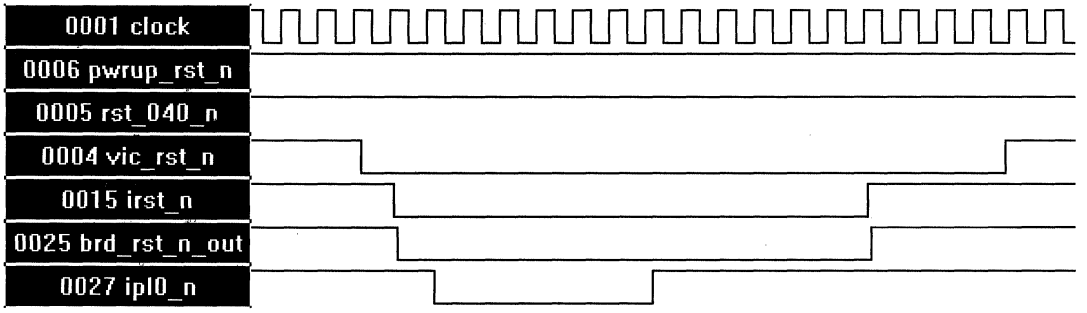


Figure 7. VIC64-Initiated Reset

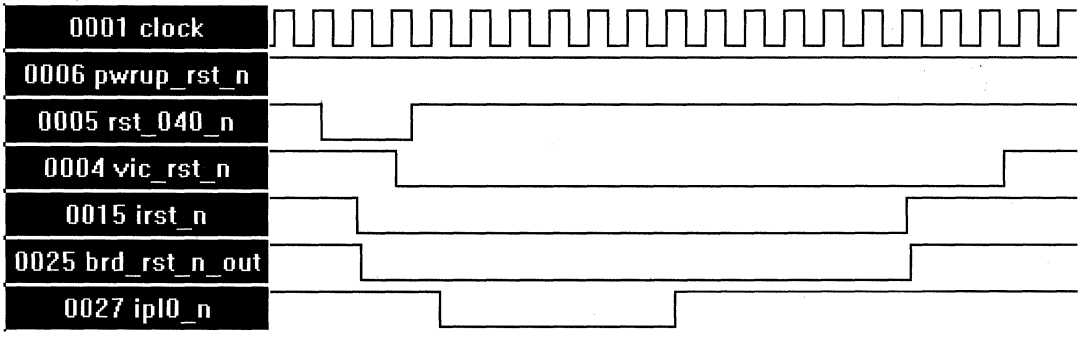


Figure 8. 68040-Initiated Reset

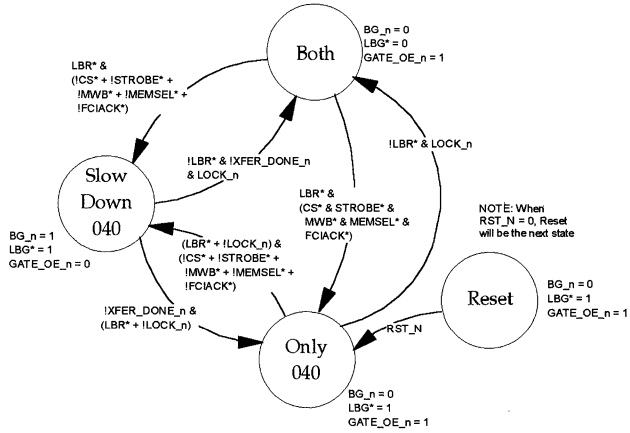


Figure 9. Bus Arbitration State Machine

tion. Rather, the state of the FCIACK*, MEMSEL*, CS*, STROBE*, or MWB* signals determine if the 68040 requires access to the VIC64's bus for either memory access, VIC64 or CY7C964 register access, or VMEbus access.

After a board reset has completed, the state machine transitions from the *Reset* state to the *Only_040* state. In this state, the 68040's $\overline{BG}$ signal is active, granting the 68040 access to its private bus. The LBG* to the VIC64 is inactive and the GATE_OE_N signal is also inactive. The GATE_OE_N signal is used to open the gate between the 68040's bus and the VIC64's bus. This "gate" consists of '245-type bidirectional drivers between the data busses and '244-type drivers for the 68040's address and control signals. It is suggested that FCT-C speed gates be used to insure that the data and address signals from the 68040 are driven to the VIC64 and/or the VMEbus with adequate setup time to DS* and PAS*.

The bus arbitration state machine is implemented in a CY7C335–83 and is named the Bus Arbitration PLD. The VHDL code describing this PLD is in Appendix D. The PLD and its connections within the circuit are shown in the schematic in *Figure 18*.

68040 Request for VIC64 Bus Access

There are two states from which the 68040 can gain access to the VIC64's bus, the *Only_040* state and the *Both* state. From the *Only_040* state, the 68040 would attempt access to the VIC64 bus with either the FCIACK*, CS*, STROBE*, MWB*, or the MEMSEL* going active. Only one of the signals would go active in a given access cycle. If the LBR* from the VIC64 is not active, the 68040 is granted access to the VIC64's bus by transitioning to the *Slow_Down_040* state. Another possible transition into the *Slow_Down_040* state from the *Only_040* state is if the 68040 is currently in the middle of a read-modify-write cycle, indicated by the $\overline{LOCK}$ signal being active. Regardless of the state of LBR*, if $\overline{LOCK}$ is active, the read-modify-write cycle is allowed to continue before the VIC64 can gain control of its bus.

Once in the *Slow_Down_040* state, the BG to the 68040 is driven inactive (to cause the 1-cycle delay in the 68040 bus cycle as described above) and the GATE_OE_N is driven active. When the current cycle completes as indicated by the XFER_DONE_N signal going active, the state machine transitions to either the *Both* state or the *Only_040* state depending on the state of the LBR* and $\overline{LOCK}$ signals.

If the VIC64 currently has ownership of its bus and the 68040 requests the VIC64's bus, the 68040 will not be granted access until the LBR* from the VIC64 has gone inactive. This could pose a problem if the VIC64 were in the midst of a block transfer. The 68040 might not receive ownership of the VIC64 in a timely fashion. Although not implemented in this application note, a bus timeout could be implemented to cancel the 68040's attempt to access the VIC64's data bus, or a method of testing the BLT* signal before initiating a cycle could be used.

VIC64 Bus Requests

The VIC64 requests ownership of its bus via the LBR* going active. If the active state is *Only_040* and the 68040 is currently not in the middle of a read-modify-write cycle (LOCK inactive), the state machine will transition to the *Both* state. In this state, the 68040 will have access to its bus, the VIC64 will be granted access to its bus, and the gate between the two buses will be closed. If the active state is *Slow_Down_040*, indicating that the 68040 currently owns the VIC64's bus, the VIC64 will not be granted its bus until the 68040 finishes its current cycle (assuming that the cycle is not the first half of a read-modify-write cycle). When the cycle completes, the state machine will transition from the *Slow_Down_040* state to the *Both* state.

Once in the *Both* state, the state machine will not transition until the VIC64 finishes its current cycle and releases its bus by driving the LBR* signal inactive. If the 68040 is attempting access to the VIC64's bus via the CS*, STROBE*, MWB*, or MEMSEL* signals, the state machine will transition to the *Slow_Down_040* state; otherwise, it will transition to the *Only_040* state.

Sample Arbitration Timing Diagrams

Figure 10 is a sample arbitration timing diagram. As the state machine exits the *Reset* state, $\overline{BG}$ is active, and GATE_OE_N and LBG* are inactive. When the 68040 attempts access to the VIC64's bus with the MEMSEL* signal, it is granted access and the $\overline{BG}$ signal goes inactive and the GATE_OE_N goes active. During the access, the LBR* signal goes active signifying that the VIC64 wants access to its bus. It is granted access (LBG* goes LOW) after the 68040's cycle completes with the XFER_DONE_N signal pulsing active.

During the VIC64's active time on its bus, the 68040 attempts access to the VIC64's bus via the MWB* signal. The 68040's cycle does not begin until the

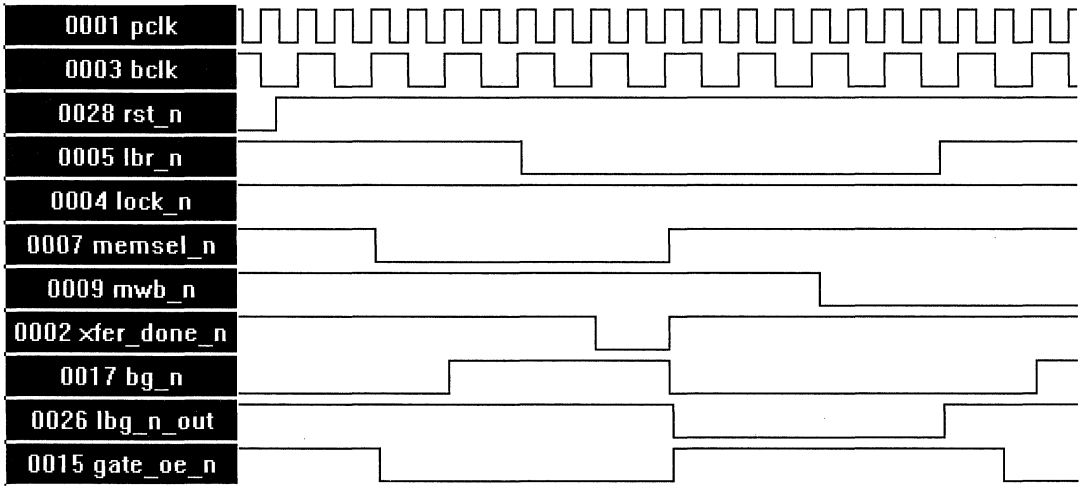


Figure 10. Arbitration Timing Diagram 1

LBR* signal is driven inactive, at which time the LBG* signal is driven inactive along with $\overline{BG}$. The *gate_oe_n* signal is driven active, allowing the 68040 onto the VIC64's bus.

Figure 11 is a continuation of *Figure 10*. The 68040 is completing its access with the MWB* signal and begins a read-modify-write cycle via the MEMSEL* signal. At the same time, the VIC64 requests access to its bus with the LBR* signal. In this case, the 68040 wins the arbitration and is allowed to complete the two cycles of the read-modify-write sequence. Once the sequence completes, the VIC64 is granted access to its bus until it deasserts the LBR* signal.

VIC64 and CY7C964 Register Access Cycles

VIC64 and CY7C964 register access cycles, as well as all other access cycles, are controlled by three PLDs. Two of the PLDs, the Address and Cycle Decode PLDs, control the initiation of a transfer. They are PALC22V10D-7s. The remaining PLD, the Cycle Termination PLD, controls the normal or abnormal completion of a cycle. This PLD is a CY7C335-83. The VHDL code for these PLDs can be found in Appendix B and Appendix C respectively. The PLDs and their connections within the circuit are shown in the schematic in *Figure 18*.

Selection of the PALC22V10D and CY7C335 Devices

The PALC22V10D and CY7C335 were chosen for a single, key reason. Both have a guarantee on their output data stability. The CY7C335 has a parameter, t_{OH}, that guarantees 2 ns of output data stability from the clock supplied to the part. The PALC22V10D-7 also guarantees a minimum on the t_{CO} specification of 2 ns. This is vital to the design because the 68040 running at 40 MHz requires that signals such as $\overline{TA}$, $\overline{TEA}$, $\overline{TBI}$, $\overline{TCI}$, etc., have a hold time of 2 ns from the rising edge of BCLK.

Selecting the VIC Registers vs. the CY7C964 Registers

Both the VIC64 registers and the CY7C964 registers are mapped into the same base address of A31–A28 = "0001." To make the determination between register sets, the lowest-order address bits, A00 and A01, are used in conjunction with the size signals, SIZ0 and SIZ1. When a byte transfer is requested and the lowest address bits are both HIGH, this is decoded as a VIC64 register access. When a longword transfer is requested and the lowest ad-

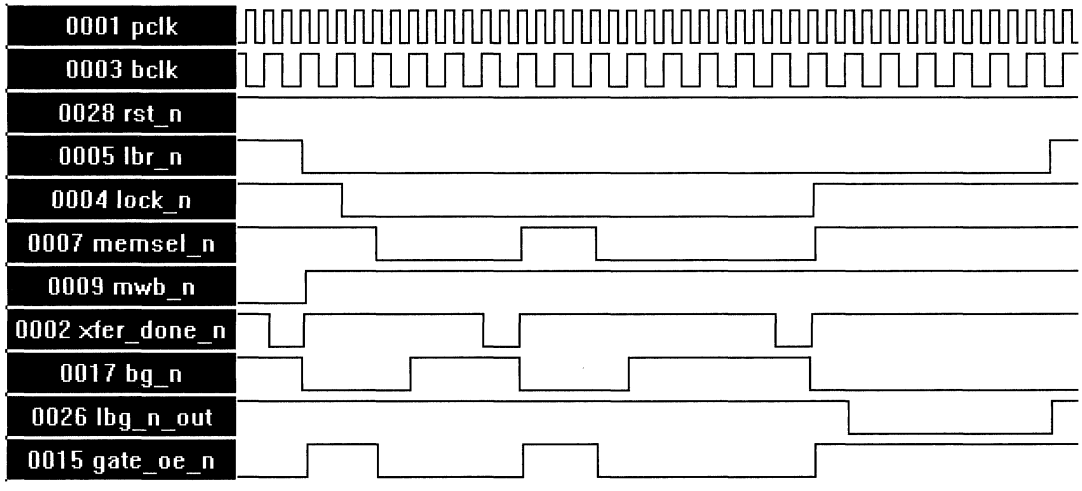


Figure 11. Arbitration Timing Diagram 2

dress bits are both LOW, this is decoded as a CY7C964 register access.

Register Access Cycle Initiation

If an address of $1xxxxxxx_{16}$ is detected when $\overline{TS}$ from the 68040 is active, a register access cycle is initiated. The address is qualified by the lowest address bits, the SIZ signals, and the transfer type from the 68040. For the CY7C964 register access, the PAS* and DS* signals are both kept inactive and only the STROBE* signal is allowed to be driven active. Data on the D31–D08 signals will be written into the CY7C964's while the data on D07–D00 is ignored. For the VIC64 register access, the PAS*, DS*, and CS* are all driven active. Data to be written to the VIC64 would be presented on D07–D00. Data read from the VIC64 would also appear on D07–D00.

To assure the proper timing on the D07–D00 signals with respect to the DS* signal, DS* is driven active on the cycle following PAS* driven active. This guarantees that, during a write cycle, data is present at the VIC64 prior to DS* becoming active.

Register Access Cycle Termination

The end of a register access cycle is indicated differently depending on whether the VIC64 registers were accessed or the CY7C964 registers were accessed. Also, the read or write status of the transfer has a bearing on how the cycle is terminated. For the VIC64 register transfers, the Cycle Termination PLD waits for either DSACK0* or DSACK1* to occur to indicate the end of the transfer. The DSACK1* and DSACK0* signals are registered as they enter the Cycle Termination PLD in order to synchronize them to the BCLK before they are used in output equations.

For the CY7C964 register transfers, the PLD counts three BCLK cycles before ending the cycle. This is because there are no external signals that indicate that the CY7C964s have received data.

In order to allow proper data hold times to the CY7C964 or VIC64, the termination of a write cycle is handled differently from the termination of the read cycle. In a read cycle, all active signals (PAS*, DS*, and CS*) are driven inactive at the same time in response to the XFER_DONE_N signal from the Cycle Termination PLD. However, for writes, an additional signal, XFER_DONE_W_N, is activated a full cycle before XFER_DONE_N. The STROBE* signal for CY7C964 register writes and the DS* for VIC64 register writes are driven inactive in response to this signal. On the subsequent BCLK cycle, the 68040 is given a $\overline{TA}$ signal and the PAS* and CS* signals are driven inactive. This insures that the rising edge of STROBE* or DS* is a cycle before the 68040 can remove data from the bus, thus guaranteeing the necessary data hold time into the CY7C964's and VIC64.

Performance of Register Access Cycles

An example of VIC64 register access is shown in *Figure 12*. An example of CY7C964 register access is shown in *Figure 13*. From these diagrams, the following performance figures are guaranteed for the different types of register access cycles.

VIC64 register write: 11 BCLK cycles assuming the slowest DSACK1/0* response time from the VIC64.

VIC64 register read: 10 BCLK cycles assuming the slowest DSACK1/0* response time from the VIC64.

CY7C964 register write: 10 BCLK cycles.

Master Read Cycles

Master Read cycles are also controlled by three PLDs, two Address and Cycle Decode PLDs and a Cycle Termination PLD. These cycles are very similar to a VIC64 Register read; however, instead of the VIC64 providing data and terminating the cycle, an addressed slave board would provide data and indicate that the data is available with the VMEbus signal, DTACK*. The VIC64 would issue DSACK1* and/or DSACK0* in response to the DTACK* signal.

Master Read Cycle Initiation

If an address of $2xxxxxxx_{16}$, $3xxxxxxx_{16}$, or $Fxxxxxxx_{16}$ is detected, along with R/$\overline{W}$ being in the HIGH state, when $\overline{TS}$ from the 68040 is active, a VMEbus master read access cycle is initiated. The address is qualified by the transfer type from the 68040.

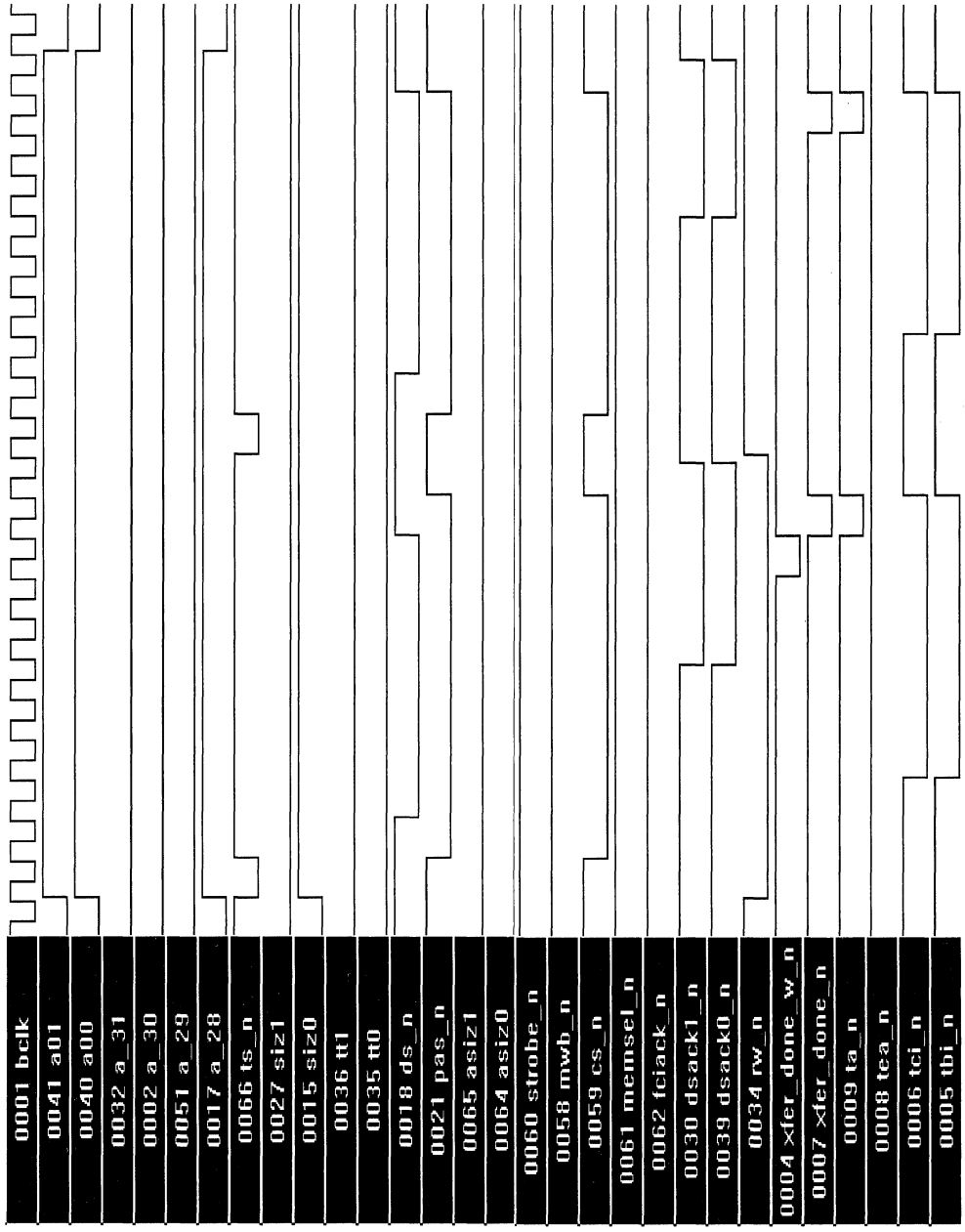


Figure 12. VIC64 Register Access

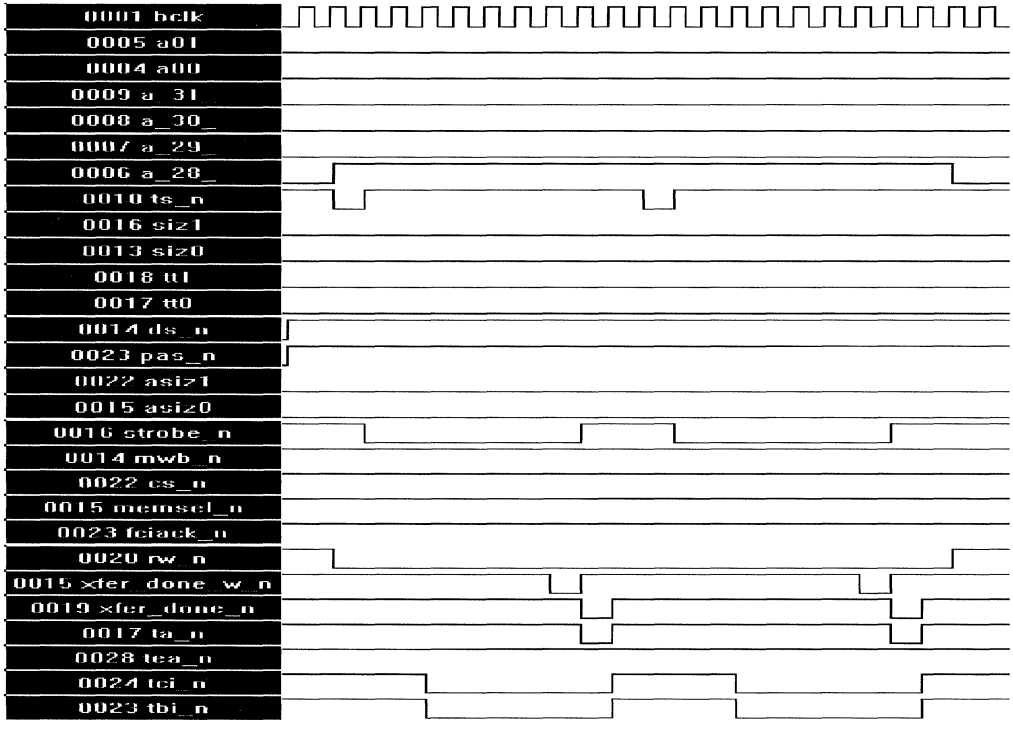


Figure 13. CY7C964 Register Access

MWB*, PAS* and DS* are driven active when the cycle is decoded and the ASIZ1 and ASIZ0 are driven based on the address from the 68040. *Table 1* shows how the address from the 68040 is decoded.

Table 1. 68040 Address Decode

68040 Address	Address Size	ASIZ1/ASIZ0
$2xxxxxxx_{16}$	A16	1/0
$3xxxxxxx_{16}$	A24	1/1
$Fxxxxxxx_{16}$	A32	0/1

The 68040 also indicates the memory space that is to be accessed with its TM2–TM0 lines. These signals are driven the the VIC64's FC2 and FC1 signals for generating AM codes on the VMEbus. The VIC64 will control the buffer control signals to the CY7C964's based on the size of the data that is to be transferred (indicated by the SIZ1 and SIZ0 signals from the 68040) and will initiate a VMEbus read with the appropriate VMEbus signals.

Master Read Cycle Termination

Once there has been a VMEbus read cycle initiated by the VIC64, there are three typical ways the cycle can be terminated. The cycle can be ended normally, be deadlocked and retried, or be terminated abnormally via a bus error.

Master Read Cycle Normal Termination

A normal master read will be terminated when the Cycle Termination PLD receives one or both of the DSACK0* or DSACK1* signals from the VIC64. These signals are driven by the VIC64 in response to a DTACK* signal from the addressed slave on the VMEbus backplane. The performance of a normal-

ly terminated cycle can vary due to the response time of the slave board being addressed and whether or not the VIC64 was granted access to the VMEbus quickly. *Figure 14* illustrates two back to back read cycles on the VMEbus that are terminated normally.

Master Read Cycle Deadlock/Retry Termination

A master read that ends in deadlock occurs when a slave cycle and a master cycle are asserted to the VIC64 at the same time. The VIC64 indicates that a deadlock has occurred by asserting the DEDLK* signal when the local side attempts access during a slave transaction. In response to the DEDLK* signal from the VIC64, the Cycle Termination PLD drives both the $\overline{\text{TEA}}$ and $\overline{\text{TA}}$ signals active to the 68040. This will cause the 68040 to end its cycle, wait one BCLK cycle (due to the Bus Arbitration PLD), and then attempt the cycle again.

The 68040 will continue to retry the cycle until the cycle is ended either with $\overline{\text{TEA}}$ or $\overline{\text{TA}}$ only. On each attempt the 68040 makes to the VIC64, the Address and Cycle Decode PLDs look at the state of the DEDLK_S signal from the Cycle Termination PLD. DEDLK_S is a double-registered version of the DEDLK* signal from the VIC64. If the DEDLK_S is active on an otherwise valid attempt to access the VIC64's private bus, the DEAD_N signal will activate instead of the normal signal (MWB*, CS*, etc.). The assertion of *DEAD*_N will not affect the Bus Arbitration state machine but will allow the Cycle Termination PLD to again cause a retry to the 68040 with $\overline{\text{TEA}}$ and $\overline{\text{TA}}$ together.

This method is used because there is a possibility that DEDLK* could go inactive *during* a 68040

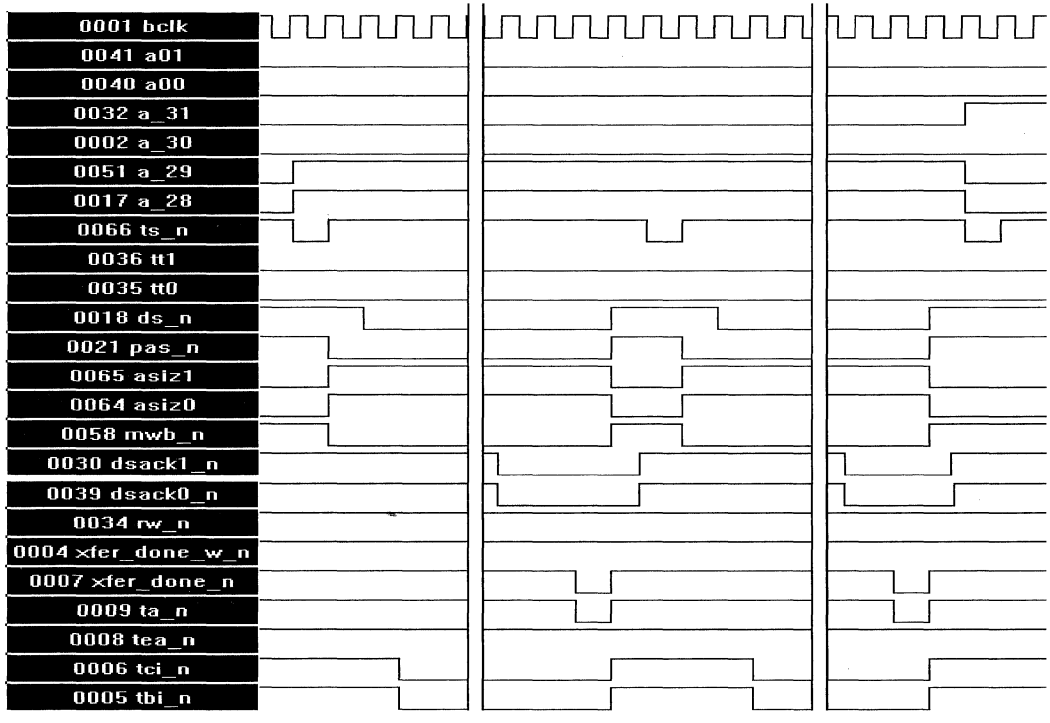


Figure 14. Master Reads

cycle. If this occurs, the Cycle Termination PLD could see DEDLK* active and terminate the cycle with $\overline{TEA}$ and $\overline{TA}$ active, thus indicating a retry to the 68040. However, the Bus Arbitration PLD may see that the Address and Cycle Decode PLD is signaling a valid cycle with LBR* inactive and an active select signal. This would cause the Cycle Termination PLD and the Bus Arbitration PLD to lose synchronization with each other. By preventing the Bus Arbitration PLD from even seeing a cycle that potentially could have a deadlock (with the *dead_n* signal), it will not arbitrate that cycle and the Cycle Termination PLD will cause a retry.

When the DEDLK* has been released by the VIC64, the 68040 will be able to finally complete the cycle that it has been retrying. The cycle will be a normal master read. *Figure 15* shows 4 cycles attempted by the 68040. The first cycle ends in retry when a DEDLK* is recognized in the middle of the

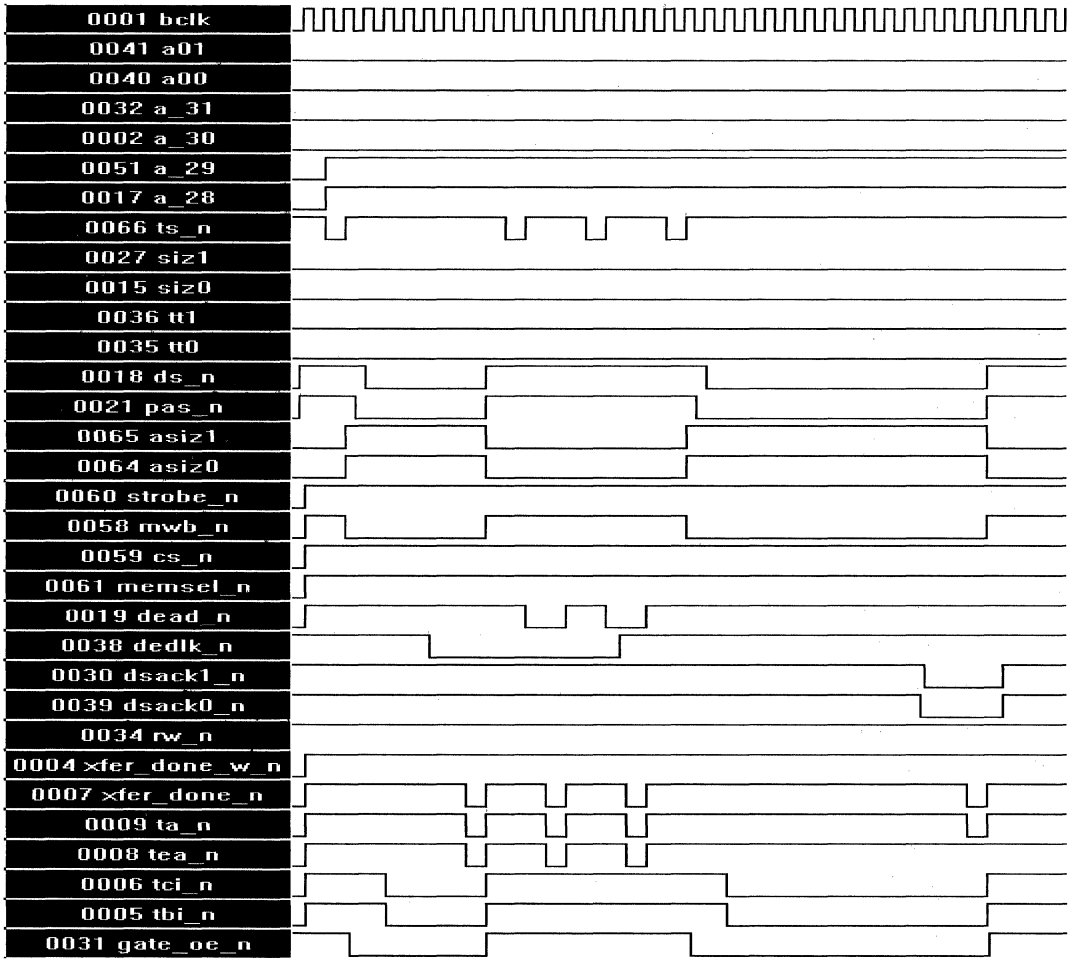


Figure 15. Master Reads with Deadlock

cycle. The next two cycles begin as deadlocked cycles and thus are immediately forced to be retried by the Cycle Termination PLD. The last cycle occurs after the deadlock and thus begins and ends as a normal master read.

Master Read Cycle Bus Error Termination

A master read will be terminated as a bus error to the 68040 when the Cycle Termination PLD receives the LBERR* signal from the VIC64. This signal is driven by the VIC64 in response to a BERR* signal from the addressed slave on the VMEbus backplane or a VMEbus timeout (based on the configuration of the TTR, register $A3 in the VIC64). A cycle terminated with a Bus Error will look similar to the timing shown in *Figure 14*. The differences will be twofold. First, the LBERR* signal will be driven by the VIC64 instead of DSACK1* and DSACK0*. Second, the $\overline{TEA}$ signal will be driven to the 68040 instead of the $\overline{TA}$ signal. Other than these differences, the cycles are equivalent.

Master Write, Writepost, and BLT Initiation Cycles

Like the Register Access cycles and Master Read cycles, Master Write cycles are also controlled by three PLDs, two Address and Cycle Decode PLDs and a Cycle Termination PLD. These cycles are very similar to a VIC64 Register write; however, instead of the VIC64 providing data and terminating the cycle, an addressed slave board would provide data and indicate that the data is available with the VMEbus signal, DTACK*. The VIC64 would issue DSACK1* and/or DSACK0* in response to the DTACK* signal.

Commonality Between the Various Write Cycles

Each of the cycles, Master Write, Writepost, and BLT initiation are subtly different. However, each shares the common trait that they are all write cycles from the 68040's perspective and all produce an MWB* signal to the VIC64. In each case however, the data is dissimilar. The Master Write and Writepost actually provide data that is transferred to an addressed slave, while the data from the BLT initiation cycle is the local address where the block transfer begins.

Write Cycle Initiation

If an address of $2xxxxxxx_{16}$, $3xxxxxxx_{16}$, or $Fxxxxxxx_{16}$ is detected, along with R/$\overline{W}$ being in the LOW state, when $\overline{TS}$ from the 68040 is active, a VMEbus master write-access cycle is initiated (or a block transfer initiation cycle if bit 6 of the BTCR is set). The address is qualified by the transfer type from the 68040. MWB*, PAS* and DS* are driven active when the cycle is decoded and the ASIZ1 and ASIZ0 are driven based on the address from the 68040. *Table 2* shows how the address from the 68040 is decoded.

Table 2. 68040 Address Decode

68040 Address	Address Size	ASIZ1/ASIZ0
$2xxxxxxx_{16}$	A16	1/0
$3xxxxxxx_{16}$	A24	1/1
$Fxxxxxxx_{16}$	A32	0/1

The 68040 also indicates the memory space that is to be accessed with its TM2–TM0 lines. These signals are driven as the VIC64's FC2 and FC1 signals for generating AM codes on the VMEbus. The VIC64 will control the buffer control signals to the CY7C964's based on the size of the data that is to be transferred (indicated by the SIZ1 and SIZ0 signals from the 68040) and will initiate a VMEbus write with the appropriate VMEbus signals.

To assure the proper timing on the D07–D00 signals with respect to the DS* signal, DS* is driven active on the cycle following PAS* driven active. This guarantees that during a write cycle data is present at the VIC64 prior to DS* active.

Write Cycle Termination

As with a VMEbus read cycle, once there has been a VMEbus write cycle initiated by the VIC64, there are three typical ways the cycle can be terminated. The cycle can be ended normally, be deadlocked and retried, or be terminated abnormally via a bus error.

Write Cycle Normal Termination

A normal master write will be terminated when the Cycle Termination PLD receives one or both of the

DSACK0* or DSACK1* signals from the VIC64. These signals are driven by the VIC64 in response to a DTACK* signal from the addressed slave on the VMEbus backplane. The performance of a normally terminated cycle can vary due to the response time of the slave board being addressed and whether or not the VIC64 was granted access to the VMEbus quickly.

In order to allow proper data hold times to the VIC64 for BLT initiation cycles and Master Write-posts, the termination of a write cycle is handled differently from the termination of the read cycle. In a read cycle, all active signals (PAS*, DS*, and MWB*) are brought inactive at the same time in response to the XFER_DONE_N signal from the Cycle Termination PLD. However, for writes, an additional signal, XFER_DONE_W_N, is activated a full cycle before XFER_DONE_N. The DS* signal is brought inactive in response to this signal. On the subsequent BCLK cycle, the 68040 is given a $\overline{TA}$ signal and the PAS* and MWB* signals are driven inactive. This insures that the rising edge of DS* is a cycle before the 68040 removes data from the bus, thus guaranteeing the necessary data hold time into the VIC64.

This timing is not an issue for Master writes since the VMEbus specification states that a slave will only issue a DTACK* after it has accepted the data written to it. Thus, hold time on the data is inherent in the delay from DTACK* on the VMEbus to DSACKx* on the local bus to $\overline{TA}$ from the cycle termination PLD. *Figure 16* illustrates two back-to-back write cycles on the VMEbus that are terminated normally.

Write Cycle Deadlock/Retry Termination

A write that ends in deadlock occurs when a slave cycle and a master cycle are asserted to the VIC64 at the same time. The VIC64 indicates that a deadlock has occurred by asserting the DEDLK* signal when the local side attempts access during a slave transaction. In response to the DEDLK* signal from the VIC64, the Cycle Termination PLD drives both the $\overline{TEA}$ and $\overline{TA}$ signals active to the 68040. This will cause the 68040 to end its cycle, wait one BCLK cycle (due to the Bus Arbitration PLD), and then attempt the cycle again.

As with deadlock on a read cycle, there is a timing relationship between the bus arbitration PLD and the cycle termination PLD that must be maintained. This timing is discussed in the Master Read Cycle Deadlock/Retry Termination section above. Timing for write cycles that deadlock is identical to the timing shown in *Figure 16*. The only difference is the relationship of DS* to both MWB* and PAS* as described above.

Write Cycle Bus Error Termination

A master write will be terminated as a bus error to the 68040 when the Cycle Termination PLD receives the LBERR* signal from the VIC64. This signal is driven by the VIC64 in response to a BERR* signal from the addressed slave on the VMEbus backplane or a VMEbus timeout (based on the configuration of the TTR, register $A3 in the VIC64). A cycle terminated with a Bus Error will look similar to the timing shown in *Figure 16*. The differences will be twofold. First, the LBERR* signal will be driven by the VIC64 instead of DSACKx*. Second, the $\overline{TEA}$ signal will be driven to the 68040 instead of the $\overline{TA}$ signal. Other than these differences, the cycles are equivalent.

Interrupt Acknowledge Cycles

Interrupt Acknowledge cycles are controlled by the Interrupt PLD, Cycle Termination PLD, and Address and Cycle Decode PLDs. Typical functionality of the Interrupt PLD is shown in *Figure 17*. Operation of the Address Decode PLDs and the Cycle Termination PLD is comparable to a Master Read Cycle except that FCIACK* is active rather than MWB*.

Operation At Reset

Although not an interrupt-related function, the Interrupt PLD controls the configuration of the 68040's buffer mode via the $\overline{IPL2}$, $\overline{IPL1}$, and $\overline{IPL0}$ signals. During a board reset, the signals are all driven to a HIGH state to configure the 68040's signals to Large Buffer mode.

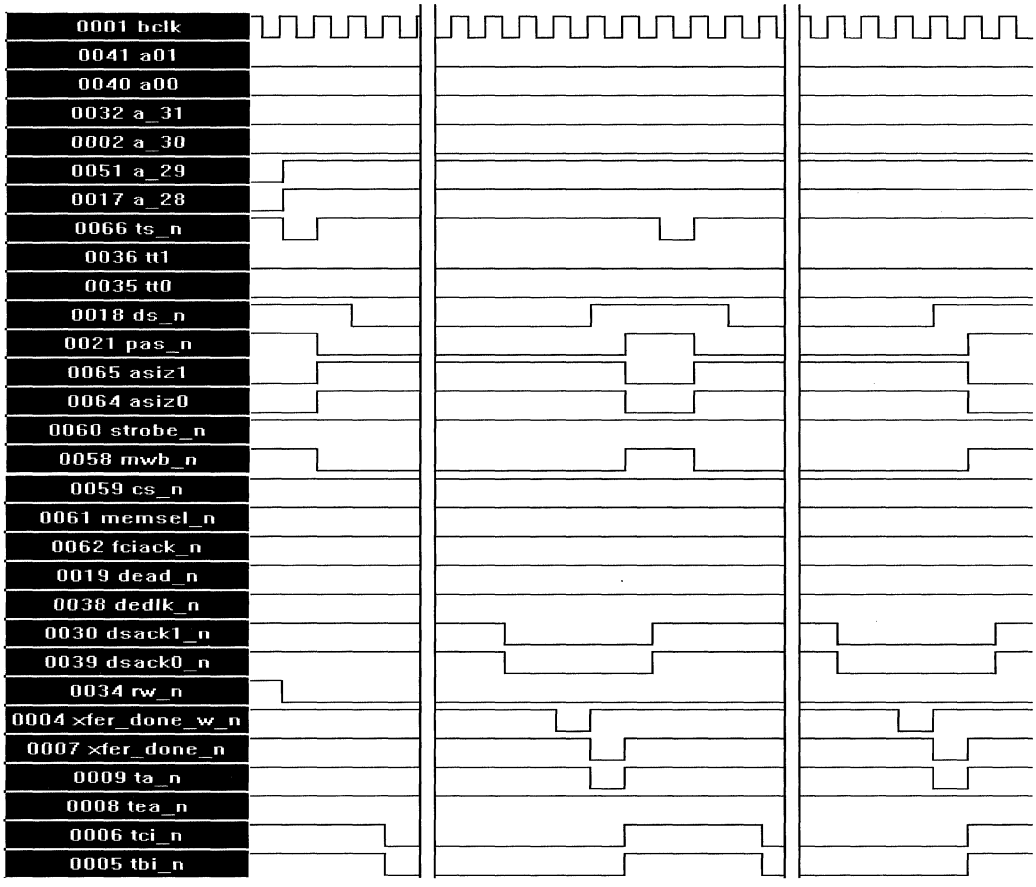


Figure 16. Master Writes

VMEbus vs. Local Interrupts

There are two possible sources for interrupts, the VMEbus and local interrupts. For VMEbus interrupts, the 68040 will only be involved if the VIC64 is configured as an interrupt handler. When the VMEbus interrupter generates an interrupt, the VIC64 will assert an interrupt to the 68040 via the IPL2*–IPL0* lines. The 68040 will respond with an interrupt acknowledge cycle. When the VIC64 sees the interrupt acknowledge cycle from the 68040, it obtains the VMEbus to request the Status/ID vector from the Interrupter. As the Status/ID vector is placed on data bus, it is passed through to the 68040 and the VIC64 terminates the cycle.

For local interrupts, a device on the board requiring service will assert an interrupt to the VIC64 via the LIRQ7*–LIRQ0* lines. The VIC64 will then assert an interrupt to the 68040 via the IPL2*–IPL0* lines. The 68040 will respond with an interrupt acknowledge cycle. There are two possible responses to the interrupt acknowledge cycle from the 68040. If the VIC64 is enabled to supply a vector for the current interrupt, it will do so and terminate the cycle with the DSACKx* signals. If the VIC64 is not

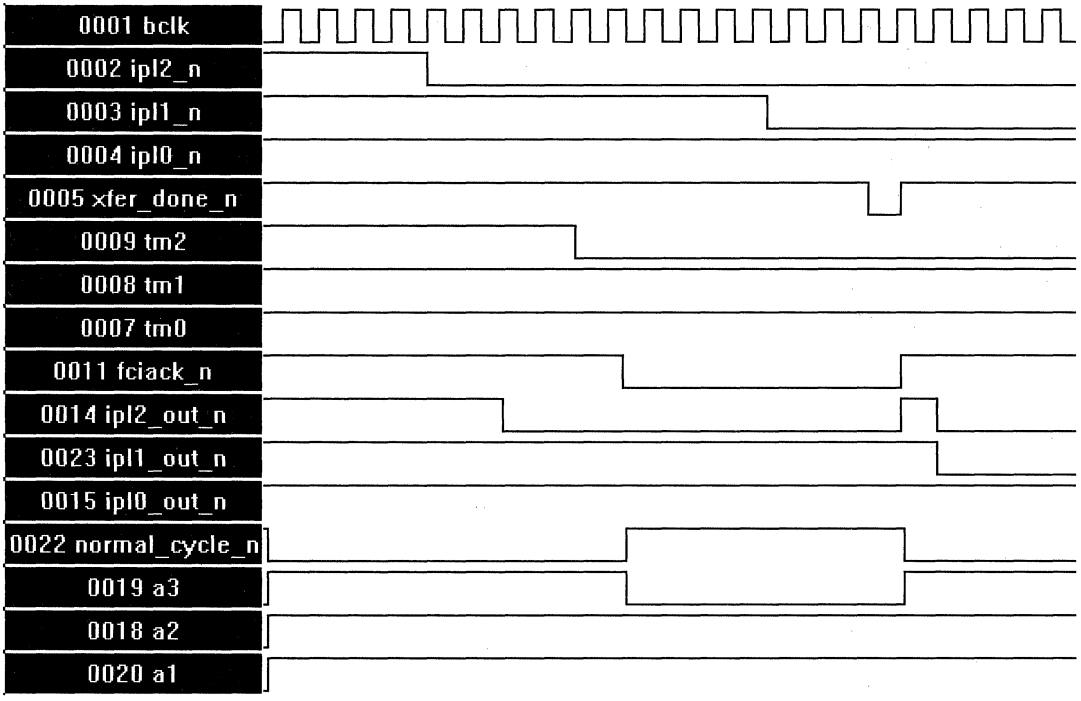


Figure 17. Interrupt Initiation and Acknowledge

enabled to provide a vector, it will assert the LIACKO* signal instead. The Cycle Termination PLD asserts AVEC to the 68040 in response to LIACKO* active and then terminates the cycle.

Another possible configuration for local interrupts is to have the LIACKO* from the VIC64 tell the interrupting device to supply a Status/ID vector to the 68040 and then terminate the cycle with a $\overline{TA}$ to the 68040.

Interrupt Initiation from the VIC64

As described above, the VIC64 issues an interrupt via its IPL2*–IPL0* signals. The IPL2*–IPL0* signals are normally in a HIGH state and are pulled LOW to request interrupt service from the 68040. When the IPL2*–IPL0* signals are pulled LOW, the Interrupt PLD synchronizes the signals before providing them to the 68040. The VIC64 may have up to 10 ns of skew in the IPL2*–IPL0* signals and that skew could be expanded to a full BCLK cycle through synchronization. However, the 68040 must see the interrupt level for two full BCLK cycles before it is considered valid so the skew is inconsequential.

Interrupt Cycle Initiation by the 68040

When the 68040 begins a bus cycle with $\overline{TS}$ active and the TT1 and TT0 signals are both HIGH, an interrupt acknowledge cycle is indicated. The SIZ1 and SIZ0 signals are also qualified to make sure they are indicating a byte-width operation. The Address Decode PLDs respond to the cycle by issuing FCIACK*, PAS*, and DS*. The VIC64 recognizes the beginning of an interrupt acknowledge cycle on the overlap of FCIACK*, PAS*, and DS* active.

Interrupt Cycle Decode

When FCIACK* goes active, the Interrupt PLD captures the current state of the $\overline{IPL2}$–$\overline{IPL0}$ signals and holds them throughout the cycle. The use of a Cypress PALC22V10D for this PLD guarantees the required hold times on the $\overline{IPL2}$–$\overline{IPL0}$ signals to the 68040 are met. When the cycle terminates, the $\overline{IPL2}$–$\overline{IPL0}$ signals are driven inactive for at least one BCLK cycle before a new interrupt level can be driven.

Another function that the Interrupt PLD performs is steering the TM2–TM0 signals from the 68040 onto the A3–A1 address lines on the VIC64. The TM2–TM0 signals from the 68040 contain the level of the interrupt being acknowledged and the VIC64 requires that information be passed on address lines A3–A1.

Interrupt Cycle Termination

The interrupt cycle is terminated in one of two ways from the VIC64. If the VIC64 is configured to supply a Status/ID vector, it will place that vector on D7–D0 and supply DSACKx* to the Cycle Termination PLD. If the VIC64 is not configured to supply a vector, it will issue a LIACKO* signal which will cause the Cycle Termination PLD to issue AVEC to the 68040 and then terminate the cycle with a $\overline{TA}$.

Summary

This application note has designed a possible VIC64 to Motorola 68040 interface. The issues and assumptions that must be addressed in the interface have been covered. The circuitry required for bus arbitration, resets, reads, writes, and interrupts has been designed. VHDL code for the PLDs used in the application note as well as timing diagrams and schematics have been provided in the following Appendices.

References

1. Cypress Semiconductor, *VIC068A/VAC068A User's Guide*, June, 1992.
2. Cypress Semiconductor, *VIC64/CY7C964 Design Notes*, October, 1993.
3. Motorola, Inc., *MC68040 Microprocessors User's Manual (M68040UM/AD)*, 1992.
4. Mazor, S., and P. Langstraat, *A Guide to VHDL*, Boston: Kluwer Academic, 1992.

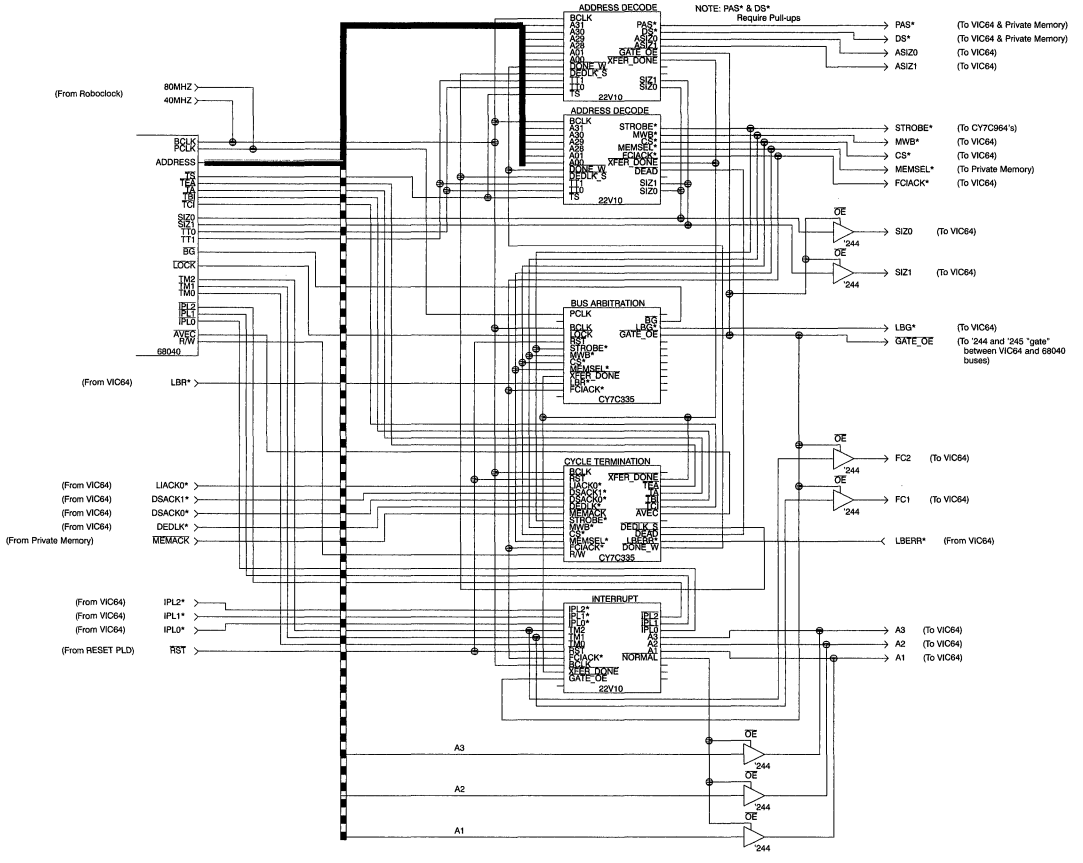


Figure 18. Schematic of Control Logic Circuitry

Appendix A. Reset Control PLD (CY7C335)

```
-- RESET PLD design
--
-- The following table is a cross reference between the PLD port names and
-- those on the schematic in the application note text.
--
--    clock            =  BCLK
--    vic_rst_n        =  RESET*
--    rst_040_n        =  RSTO "bar"
--    pwrup_rst_n      =  PURST "bar"
--    irst_n           =  IRESET*
--    brd_rst_n_out    =  RST "bar"
--    ipl0_n           =  IPL0*

ENTITY rst_ctrl IS
        PORT (clock, vic_rst_n, rst_040_n, pwrup_rst_n : in bit;
              irst_n, brd_rst_n_out : out bit;
              ipl0_n : inout x01z);
attribute part_name of rst_ctrl:entity is "c335";
attribute pin_numbers of rst_ctrl:entity is "clock:1 vic_rst_n:4 "
                                    & "rst_040_n:5 pwrup_rst_n:6";
END rst_ctrl;

USE work.rtlpkg.all;

ARCHITECTURE operation OF rst_ctrl IS
        signal pwrup_rst_n_rising:bit;
        signal pwrup_rst_n_reg:bit;
        signal pwrup_rst_n_reg1:bit;
        signal brd_rst_n:bit;
        signal start:bit;
        signal expired: bit;
        signal timer_count:integer(0 to 4);
        signal ipl0_oe: bit;
        signal ipl0_sig: bit;
        type states is (idle, rst1, rst2, rst3, wait_for_no_rst);
        signal rst_state: states;

        BEGIN

-- This process captures the power-up or pushbutton reset in two registers
-- in order to synchronize the signal and create a pulse on the rising edge
-- of the reset

        sync_rst: PROCESS BEGIN
            WAIT UNTIL clock = '1';
            pwrup_rst_n_reg <= pwrup_rst_n;
            pwrup_rst_n_reg1 <= pwrup_rst_n_reg;
        END PROCESS;

-- This concurrent assignment creates a pulse when the rising edge of the
-- power-up or pushbutton reset occurs.  This is used to begin a global reset
-- to the VIC which must be reset AFTER the Vcc and Oscillator are stable.
```

Appendix A. Reset Control PLD (CY7C335) (continued)

```
        pwrup_rst_n_rising <= '1' WHEN ((pwrup_rst_n_reg = '1') AND
                                        (pwrup_rst_n_reg1 = '0'))
                              ELSE '0';

-- This concurrent assignment guarantees that the reset out of this part will
-- be low when either the power-up/pushbutton reset is low or the brd_rst_n
-- signal generated in the process below is low.  This assures that as this
-- PLD powers up, the reset out of it will be low even if the process below
-- has not begun stable operation.

        brd_rst_n_out <= '0' WHEN ((brd_rst_n = '0') OR
                                   (pwrup_rst_n = '0'))
                         ELSE '1';

        rst: PROCESS BEGIN
            WAIT UNTIL clock = '1';

-- Do the following while the board is powering up or the pushbutton
-- is depressed

            IF (pwrup_rst_n_reg = '0') THEN
                  rst_state <= idle;
                  brd_rst_n <= '0';
                  ipl0_sig <= '1';
                  irst_n <= '1';
                  start <= '0';
            END IF;
            CASE (rst_state) IS

-- When the state machine is in the idle state, we look for the VIC to
-- initiate a reset or the 68040 to initiate a reset.  We also look for the
-- end of the powerup reset.  Remember that even though this state machine
-- is waiting for the end of power-up/pushbutton reset, the board is still
-- held in reset by the brd_rst_n_out signal above.

                  WHEN idle =>
                        IF ((pwrup_rst_n_rising = '1') OR
                             (vic_rst_n = '0') OR
                             (rst_040_n = '0')) THEN
                                rst_state <= rst1;
                                start <= '1';
                                irst_n <= '0';
                                brd_rst_n <= '0';
                        END IF;

-- In the rst1 state, we wait for one of two events.  If the VIC responds to
-- the reset from this PLD before the timer expires, we pull ipl0_n low and
-- continue to the rst2 state.  This would be the normal procedure.  If for
-- some unknown reason the VIC doesn't respond, we would wait for the timer
-- to expire and then assert ipl0_n.
```

Appendix A. Reset Control PLD (CY7C335) (continued)

```
                    WHEN rst1 =>
                            start <= '0';
                            IF (vic_rst_n = '0') THEN
                                    rst_state <= rst2;
                                    ipl0_sig <= '0';
                                    start <= '1';
                            ELSIF (expired = '1') THEN
                                    rst_state <= rst2;
                                    ipl0_sig <= '0';
                                    start <= '1';
                            END IF;

-- Just wait around in this state until the timer expires.  Remove the ipl0_n
-- signal at the end of this state.

                    WHEN rst2 =>
                            start <= '0';
                            IF ((expired = '1') AND (start = '0')) THEN
                                    rst_state <= rst3;
                                    ipl0_sig <= '1';
                                    start <= '1';
                            END IF;

-- Just wait around in this state until the timer expires.  Remove resets at
-- the end of this state.

                    WHEN rst3 =>
                            start <= '0';
                            IF (expired = '1') THEN
                                    rst_state <= wait_for_no_rst;
                                    irst_n <= '1';
                                    brd_rst_n <= '1';
                            END IF;

-- We remain in this state until the VIC comes out of reset.

                    WHEN wait_for_no_rst =>
                            IF (vic_rst_n = '1') THEN
                                    rst_state <= idle;
                            END IF;

-- Make the state machine complete.

                    WHEN others => rst_state <= idle;
            END CASE;
        END PROCESS;

-- The following makes the ipl0_sig signal a three-state signal on the
-- pin of the device.  This is required since ipl0 is normally driven by
-- the VIC64 but needs to be driven by this PLD during global reset.

        ipl0_oe <= NOT ipl0_sig;
        ipl0: bufoe port map (ipl0_sig, ipl0_oe, ipl0_n, open);
```

Appendix A. Reset Control PLD (CY7C335) (continued)

```
-- The timer process runs a counter that times how long the state machine
-- above should remain in a state.

        timer: PROCESS BEGIN
                WAIT UNTIL clock = '1';
                IF (pwrup_rst_n_reg = '0') THEN
                        timer_count <= 0;
                        expired <= '0';
                ELSE
                        IF (timer_count /= 0) THEN
                             timer_count <= timer_count + 1;
                        ELSE
                             timer_count <= 0;
                        END IF;

                    IF start = '1' THEN
                             timer_count <= 1;
                        END IF;

                        IF timer_count = 4 THEN
                             expired <= '1';
                    ELSE
                         expired <= '0';
                        END IF;

                END IF;
        END PROCESS;
END operation;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D)

```
-- ADDRESS DECODER design 1
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk             =  BCLK on 68040
--    a(31) to a(28)   =  Address bus on 68040
--    ts_n             =  TS "bar" on 68040
--    tt1              =  TT1 on 68040
--    tt0              =  TT0 on 68040
--    siz1             =  SIZ1 on 68040
--    siz0             =  SIZ0 on 68040
--    xfer_done_n      =  xfer_done_n from TERMINATION PLD
--    xfer_done_w_n    =  xfer_done_w_n from TERMINATION PLD
--    gate_oe_n        =  gate_oe_n from BUS ARBITRATION PLD
--    dedlk_s          =  dedlk_s from TERMINATION PLD
--    asiz1            =  ASIZ1 to VIC64
--    asiz0            =  ASIZ0 to VIC64
--    pas_n            =  PAS* to VIC64
--    ds_n             =  DS* to VIC64

ENTITY address_decoder IS
        PORT (a01, a00, bclk, ts_n, tt1, tt0, siz1, siz0 : in bit;
              xfer_done_n, xfer_done_w_n, gate_oe_n, dedlk_s : in bit;
              a : in bit_vector(31 downto 28);
              asiz1, asiz0 : out bit;
              pas_n, ds_n : inout x01z);
attribute part_name of address_decoder:entity is "c22v10";
END address_decoder;

USE work.rtlpkg.all;

ARCHITECTURE operation OF address_decoder IS
          SIGNAL tt, siz : bit_vector(1 downto 0);
          SIGNAL pas_sig, ds_sig : bit;
          SIGNAL open_gate : bit;
          CONSTANT byte : bit_vector(1 downto 0) := "01";
          CONSTANT word : bit_vector(1 downto 0) := "10";
          CONSTANT lword : bit_vector(1 downto 0) := "00";
          CONSTANT acknow : bit_vector(1 downto 0) := "11";
          CONSTANT normal : bit_vector(1 downto 0) := "00";
BEGIN

     tt <= tt1 & tt0;
     siz <= siz1 & siz0;

     PROCESS BEGIN
          WAIT UNTIL bclk = '1';

-- At the start of a 68040 cycle, determine which signals should be activated
-- However, if the dedlk_s signal from the CYCLE TERMINATION PLD is active,
-- a transfer should not be begun to the VIC64.
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
                IF (ts_n = '0') AND (dedlk_s = '1') THEN
                     -- 964 Registers
                     IF (a = "0001") AND (a01 & a00 = "00") AND (tt = normal) AND
                       (siz = lword) THEN
                          asiz1 <= '0';
                          asiz0 <= '0';
                          pas_sig <= '1';
                     -- VIC64 Registers
                     ELSIF (a = "0001") AND (a01 & a00 = "11") AND (tt = normal) AND
                          (siz = byte) THEN
                          asiz1 <= '0';
                          asiz0 <= '0';
                          pas_sig <= '0';
                     -- A16 Addressing
                     ELSIF (a = "0010") AND (tt = normal) THEN
                          asiz1 <= '1';
                          asiz0 <= '0';
                          pas_sig <= '0';
                     -- A24 Addressing
                     ELSIF (a = "0011") AND (tt = normal) THEN
                          asiz1 <= '1';
                          asiz0 <= '1';
                          pas_sig <= '0';
                     -- A32 Addressing
                     ELSIF (a = "1111") AND (tt = normal) THEN
                          asiz1 <= '0';
                          asiz0 <= '1';
                          pas_sig <= '0';
                     -- VIC64's Private Memory
                     ELSIF (a = "0100") AND (tt = normal) THEN
                          asiz1 <= '0';
                          asiz0 <= '0';
                          pas_sig <= '1';
                     -- Interrupt Acknowledge
                     ELSIF (tt = acknow) AND (siz = byte) THEN
                          asiz1 <= '0';
                          asiz0 <= '0';
                          pas_sig <= '0';
                     -- Not a cycle for us
                     ELSE
                          asiz1 <= '0';
                          asiz0 <= '0';
                          pas_sig <= '1';
                     END IF;
                END IF;

-- DS will follow whatever PAS does on the subsequent cycle

                IF (pas_sig = '0') THEN
                     ds_sig <= '0';
                END IF;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
-- If the cycle was a write cycle, the ds_sig must be pulled high to latch
-- data into the VIC64.  This is to assure that the local data hold time of
-- 0ns to the VIC64 is not violated.  If this were a cycle sending data
-- across the VMEbus, pulling ds_sig high before pas_n will not cause
-- problems because the slave board that is being written to would have
-- captured data when it asserted DTACK* to the VIC64.

               IF (xfer_done_w_n = '0') THEN
                    ds_sig <= '1';
               END IF;

-- When the cycle has been completed, the signals are all returned to their
-- inactive states.

               IF (xfer_done_n = '0') THEN
                    asiz1 <= '0';
                    asiz0 <= '0';
                    pas_sig <= '1';
                    ds_sig <='1';
               END IF;
          END PROCESS;

-- The pas_n and ds_n are driven by the VIC64 when it has access to its
-- private bus.  By looking at the state of the gate_oe_n signal, the owner
-- of the bus can be determined.  If the gate_oe_n signal is asserted (low),
-- the '040 has control of the bus and the pas_n and ds_n signals must be
-- active.

     open_gate <= NOT gate_oe_n;

     pas: bufoe PORT MAP (pas_sig, open_gate, pas_n, open);
     ds:  bufoe PORT MAP (ds_sig,  open_gate, ds_n,  open);

END operation;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
-- ADDRESS DECODER design 2
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk             =  BCLK on 68040
--    a(31) to a(28)   =  Address bus on 68040
--    ts_n             =  TS "bar" on 68040
--    tt1              =  TT1 on 68040
--    tt0              =  TT0 on 68040
--    siz1             =  SIZ1 on 68040
--    siz0             =  SIZ0 on 68040
--    xfer_done_n      =  xfer_done_n from TERMINATION PLD
--    xfer_done_w_n    =  xfer_done_w_n from TERMINATION PLD
--    dedlk_s          =  dedlk_s from TERMINATION PLD

--    dead_n           =  dead_n to TERMINATION PLD
--    memsel_n         =  chip select for VIC64's private memory
--    strobe_n         =  STROBE* on CY7C964's
--    mwb_n            =  MWB* to VIC64
--    cs_n             =  CS* to VIC64
--    fciack_n         =  FCIACK* to VIC64

ENTITY address_decoder IS
        PORT (a01, a00, bclk, ts_n, tt1, tt0, siz1, siz0, xfer_done_n : in bit;
              xfer_done_w_n, dedlk_s : in bit;
              a : in bit_vector(31 downto 28);
              memsel_n, strobe_n, mwb_n, cs_n, fciack_n, dead_n : out bit);
attribute part_name of address_decoder:entity is "c22v10";
END address_decoder;

USE work.rtlpkg.all;

ARCHITECTURE operation OF address_decoder IS
          SIGNAL tt, siz : bit_vector(1 downto 0);
          CONSTANT byte : bit_vector(1 downto 0) := "01";
          CONSTANT word : bit_vector(1 downto 0) := "10";
          CONSTANT lword : bit_vector(1 downto 0) := "00";
          CONSTANT acknow : bit_vector(1 downto 0) := "11";
          CONSTANT normal : bit_vector(1 downto 0) := "00";

BEGIN

     tt <= tt1 & tt0;
     siz <= siz1 & siz0;

     PROCESS BEGIN
          WAIT UNTIL bclk = '1';

-- At the start of a 68040 cycle, determine which signals should be activated
-- This will be run only if we are not seeing a deadlock situation via the
-- dedlk_s signal.
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
IF (ts_n = '0') AND (dedlk_s = '1') THEN
     -- 964 Registers
     IF (a = "0001") AND (a01 & a00 = "00") AND (tt = normal) AND
       (siz = lword) THEN
          strobe_n <= '0';
          mwb_n    <= '1';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- VIC64 Registers
     ELSIF (a = "0001") AND (a01 & a00 = "11") AND (tt = normal) AND
          (siz = byte) THEN
          strobe_n <= '1';
          mwb_n    <= '1';
          cs_n     <= '0';
          memsel_n <= '1';
          fciack_n <= '1';
     -- A16 Addressing
     ELSIF (a = "0010") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '0';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- A24 Addressing
     ELSIF (a = "0011") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '0';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- A32 Addressing
     ELSIF (a = "1111") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '0';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- VIC64's Private Memory
     ELSIF (a = "0100") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '1';
          cs_n     <= '1';
          memsel_n <= '0';
          fciack_n <= '1';
     -- Interrupt Acknowledge
     ELSIF (tt = acknow) AND (siz = byte) THEN
          strobe_n <= '1';
          mwb_n    <= '1';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '0';
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
        -- Not a cycle for us
        ELSE
             strobe_n <= '1';
             mwb_n    <= '1';
             cs_n     <= '1';
             memsel_n <= '1';
             fciack_n <= '1';
        END IF;
   END IF;


-- This is the section of code that will be run if there is a deadlock.
-- If the decoded address/tt/siz information would have normally decoded
-- to a valid cycle, we send out the dead_n signal instead.  This lets the
-- TERMINATION PLD know that the 68040 is issuing a valid request to the
-- VIC64 but that the VIC64 can't be bothered cause it is currently finishing
-- a slave operation.

   IF (ts_n = '0') AND (dedlk_s = '0') THEN
        strobe_n <= '1';
        mwb_n    <= '1';
        cs_n     <= '1';
        memsel_n <= '1';
        fciack_n <= '1';

        -- 964 Registers
        IF (a = "0001") AND (a01 & a00 = "00") AND (tt = normal) AND
          (siz = lword) THEN
             dead_n <= '0';
        -- VIC64 Registers
        ELSIF (a = "0001") AND (a01 & a00 = "11") AND (tt = normal) AND
             (siz = byte) THEN
             dead_n <= '0';
        -- A16 Addressing
        ELSIF (a = "0010") AND (tt = normal) THEN
             dead_n <= '0';
        -- A24 Addressing
        ELSIF (a = "0011") AND (tt = normal) THEN
             dead_n <= '0';
        -- A32 Addressing
        ELSIF (a = "1111") AND (tt = normal) THEN
             dead_n <= '0';
        -- VIC64's Private Memory
        ELSIF (a = "0100") AND (tt = normal) THEN
             dead_n <= '0';
        -- Interrupt Acknowledge
        ELSIF (tt = acknow) AND (siz = byte) THEN
             dead_n <= '0';
        -- Not a cycle for us
        ELSE
             dead_n <= '1';
        END IF;
   END IF;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
-- If the cycle was a write cycle, the strobe_n must be pulled high to latch
-- data into the '964's.  This is to assure that the local data hold time of
-- 5ns to the 964's is not violated.

               IF (xfer_done_w_n = '0') THEN
                    strobe_n <= '1';
               END IF;

-- When the cycle has been completed, the signals are all returned to their
-- inactive states.

               IF (xfer_done_n = '0') THEN
                    strobe_n <= '1';
                    mwb_n    <= '1';
                    cs_n     <= '1';
                    memsel_n <= '1';
                    fciack_n <= '1';
                    dead_n   <= '1';
               END IF;
          END PROCESS;

END operation;
```

Appendix C. Cycle Termination PLD (CY7C335)

```
-- CYCLE TERMINATION PLD
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk             =  BCLK on 68040
--    rw_n             =  R/W "bar" on 68040
--    rst_n            =  brd_rst_n_out from RESET PLD
--    liacko_n         =  LIACKO* from VIC64
--    dsack1_n         =  DSACK1* from VIC64
--    dsack0_n         =  DSACK0* from VIC64
--    lberr_n          =  LBERR* from VIC64
--    dedlk_n          =  DEDLK* from VIC64
--    memack_n         =  MEMACK* from private memory
--    memsel_n         =  MEMSEL* from ADDRESS DECODE PLD
--    strobe_n         =  STROBE* from ADDRESS DECODE PLD
--    mwb_n            =  MWB* from ADDRESS DECODE PLD
--    cs_n             =  CS* from ADDRESS DECODE PLD
--    fciack_n         =  FCIACK* from ADDRESS DECODE PLD
--    dead_n           =  dead_n from ADDRESS DECODE PLD

--    avec_n           =  AVEC "bar" to 68040
--    xfer_done_n      =  XFER_DONE "bar" to BUS ARBITRATION/ADDRESS DECODE PLD's
--    xfer_done_w_n    =  XFER_DONE_W "bar" to BUS ARBITRATION/ADDRESS DECODE PLD's
--    tea_n            =  TEA "bar" to 68040
--    ta_n             =  TA "bar" to 68040
--    tci_n            =  TCI "bar" to 68040
--    tbi_n            =  TBI "bar" to 68040
--    dedlk_s          =  Double registered (sync'ed) dedlk_n signal to
--                                ADDRESS DECODE PLDS


ENTITY cycle_termination IS
        PORT (bclk, liacko_n, dsack1_n, dsack0_n : in boolean;
              lberr_n, dedlk_n, memack_n, rst_n, rw_n : in boolean;
              dead_n : in boolean;
              memsel_n, strobe_n, mwb_n, cs_n, fciack_n : in boolean;
              avec_n, tea_n, ta_n, tci_n, tbi_n : out bit;
              dedlk_s : out bit;
              xfer_done_w_n, xfer_done_n : buffer bit);
attribute part_name of cycle_termination:entity is "c335";
END cycle_termination;

USE work.rtlpkg.all;
USE work.table_bv.all;

ARCHITECTURE operation OF cycle_termination IS
     SIGNAL any_access     : boolean;
     SIGNAL any_access_reg : boolean;
     SIGNAL cycle_end      : boolean;
     SIGNAL start, expired : bit;
     SIGNAL timer_count    : integer(0 to 7);
```

Appendix C. Cycle Termination PLD (CY7C335) (continued)

```
     SIGNAL liacko_n_reg    : boolean;
     SIGNAL dsack0_n_reg    : boolean;
     SIGNAL dsack1_n_reg    : boolean;
     SIGNAL dedlk_n_reg     : boolean;
     SIGNAL lberr_n_reg     : boolean;

BEGIN

     any_access <= NOT memsel_n OR NOT strobe_n OR NOT mwb_n OR NOT cs_n OR
                   NOT fciack_n;

     cycle_end <= (NOT memsel_n AND NOT memack_n) OR
                  (NOT strobe_n AND (timer_count = 3)) OR
                  (NOT mwb_n    AND (NOT dsack0_n_reg OR NOT dsack1_n_reg)) OR
                  (NOT fciack_n AND (NOT dsack0_n_reg OR NOT dsack1_n_reg)) OR
                  (NOT fciack_n AND (NOT liacko_n_reg)) OR
                  (NOT cs_n     AND (NOT dsack0_n_reg OR NOT dsack1_n_reg));

     controller: PROCESS BEGIN
          WAIT UNTIL bclk;
               liacko_n_reg   <= liacko_n;
               dsack0_n_reg   <= dsack0_n;
               dsack1_n_reg   <= dsack1_n;
               dedlk_n_reg    <= dedlk_n;
               IF dedlk_n_reg THEN
                    dedlk_s <= '1';
               ELSE
                    dedlk_s <= '0';
               END IF;
               lberr_n_reg    <= lberr_n;
               start          <= '0';
               xfer_done_n    <= '1';

               IF xfer_done_n = '0' THEN
                    any_access_reg <= FALSE;
               ELSE
                    any_access_reg <= any_access;
               END IF;

-- Normal beginning of a cycle starts the cycle timer and asserts the tbi_n
-- and tci_n to inhibit bursts and caching.

               IF any_access_reg THEN
                    tbi_n <= '0';
                    tci_n <= '0';
                    start <= '1';
               END IF;

-- Normal end to a write cycle will assert the xfer_done_w_n followed by an
-- assertion of xfer_done_n and ta_n.  Normal end to a read cycle is
-- xfer_done_n and ta_n asserted.

               IF cycle_end AND NOT rw_n AND (xfer_done_n = '1') THEN
                    xfer_done_w_n <= '0';
                    start <= '0';
```

Appendix C. Cycle Termination PLD (CY7C335) (continued)

```
                END IF;

                IF (cycle_end AND rw_n) OR (xfer_done_w_n = '0') THEN
                     xfer_done_w_n <= '1';
                     xfer_done_n <= '0';
                     ta_n <= '0';
                     start <= '0';
                END IF;

--  Error endings.  If dedlk_n_reg is active and an access is being attempted,
--  retry the cycle with ta_n and tea_n asserted together.  This will occur
--  only during a cycle.  If dead_n is active, we have already had an
--  initial deadlocked cycle and we are now in a sequence of retries to the
--  68040.
--  If there is a lberr_n assertion, just end the cycle with tea_n to
--  indicate an erred cycle.
--  xfer_done_n is also asserted in either case to shut off the selects in
--  the ADDRESS DECODE PLD's.

                IF (any_access_reg AND (NOT dedlk_n_reg)) OR (NOT dead_n) THEN
                     ta_n <= '0';
                     tea_n <= '0';
                     xfer_done_n <= '0';
                     start <= '0';
                END IF;

                IF any_access_reg AND (NOT lberr_n) THEN
                     tea_n <= '0';
                     xfer_done_n <= '0';
                     start <= '0';
                END IF;

--  liacko_n being asserted means that the processor should autovector the
--  current interrupt.

                IF (NOT liacko_n) THEN
                     avec_n <= '0';
                END IF;

--  Conclusion of the cycle.  xfer_done_n and all other outputs from this
--  PLD are placed in their inactive state.

                IF (xfer_done_n = '0') THEN
                     xfer_done_n <= '1';
                     tbi_n <= '1';
                     tci_n <= '1';
                     ta_n <= '1';
                     tea_n <= '1';
                     avec_n <= '1';
                     start <= '0';
                END IF;

-- Reset condition takes priority over any of the above assignments.
```

Appendix C. Cycle Termination PLD (CY7C335) (continued)

```
                IF (NOT rst_n) THEN
                     xfer_done_n <= '1';
                     xfer_done_w_n <= '1';
                     tbi_n <= '1';
                     tci_n <= '1';
                     ta_n <= '1';
                     tea_n <= '1';
                     avec_n <= '1';
                     start <= '0';
                END IF;

     END PROCESS;

     timer: PROCESS BEGIN
          WAIT UNTIL bclk;
                IF (timer_count /= 0) THEN
                     timer_count <= timer_count + 1;
                ELSE
                     timer_count <= 0;
                END IF;
                -------------------------------------
                IF start = '1' AND (timer_count = 0) THEN
                     timer_count <= 1;
                END IF;
                -------------------------------------
                IF xfer_done_n = '0' OR start = '0' OR (NOT rst_n) THEN
                     timer_count <= 0;
                END IF;
     END PROCESS;
END operation;
```

Appendix D. Bus Arbitration PLD (CY7C335)

```
-- BUS ARBITER PLD design
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    pclk             =  PCLK on 68040
--    bclk             =  BCLK on 68040
--    bg_n             =  BG "bar" on 68040
--    lock_n           =  LOCK "bar" on 68040 (requires external pullup)
--    cs_n             =  cs_n from ADDRESS DECODE PLD
--    strobe_n         =  strobe_n from ADDRESS DECODE PLD
--    mwb_n            =  mwb_n from ADDRESS DECODE PLD
--    memsel_n         =  memsel_n from ADDRESS DECODE PLD
--    fciack_n         =  fciack_n from ADDRESS DECODE PLD
--    xfer_done_n      =  xfer_done_n from TERMINATION PLD
--    rst_n            =  brd_rst_n_out from RESET PLD
--    lbr_n            =  LBR* on VIC64
--    lbg_n_out        =  LBG* on VIC64
--    gate_oe_n        =  OE on GATE between 040 bus and VIC64 bus

ENTITY arbiter IS
        PORT (pclk, bclk, lock_n, cs_n, strobe_n, mwb_n : in bit;
              memsel_n, xfer_done_n, rst_n, lbr_n, fciack_n : in bit;
              bg_n, lbg_n_out, gate_oe_n : out bit);
attribute part_name of arbiter:entity is "c335";
attribute pin_numbers of arbiter:entity is "pclk:1 bclk:3";
END arbiter;

USE work.rtlpkg.all;

ARCHITECTURE operation OF arbiter IS
        signal lbr_n_reg1, lbr_n_reg2:bit;
        signal lbg_n : bit;
        signal selects:bit_vector(4 downto 0);
        type states is (reset, only040, slow_down040, both);
        signal arb_state: states;
        constant no_selects:bit_vector(4 downto 0) := "11111";
BEGIN

-- The local bus grant to the VIC64 must be removed within 1 VIC64 clock
-- cycle or the VIC64 would respond with an unsolicited bus request.

        lbg_n_out <= '0' WHEN ((lbr_n = '0') AND (lbg_n = '0')) ELSE '1';

-- This process captures the lbr_n signal from the VIC64 and double
-- registers it using the pclk signal.

        caputure_lbr: PROCESS BEGIN
            WAIT UNTIL pclk = '1';
            lbr_n_reg1 <= lbr_n;
            lbr_n_reg2 <= lbr_n_reg1;
        END PROCESS;
```

Appendix D. Bus Arbitration PLD (CY7C335) (continued)

```
-- gate_oe_n is triggered in a "Mealy" fashion to begin VIC64 cycles as
-- soon as possible.

        gate_oe_n <= '0' WHEN ((arb_state = slow_down040)
                                    OR
                               ((arb_state = only040) AND
                                   ((lock_n = '0' OR lbr_n_reg2 = '1') AND
                                    (selects /= no_selects)))
                                    OR
                               ((arb_state = both) AND
                                  ((lbr_n_reg2 = '1') AND
                                   (selects /= no_selects))))
                     ELSE '1';

        selects <= cs_n & strobe_n & mwb_n & memsel_n & fciack_n;

        arb_machine: PROCESS BEGIN
            WAIT UNTIL bclk = '1';
                CASE arb_state IS
                    WHEN reset =>
                        IF rst_n = '0' THEN
                            arb_state <= reset;
                            lbg_n <= '1';
                            bg_n <= '0';
                        ELSE
                            arb_state <= only040;
                            lbg_n <= '1';
                            bg_n <= '0';
                        END IF;
                    WHEN only040 =>
                        IF (lbr_n_reg2 = '0' AND lock_n = '1') THEN
                            arb_state <= both;
                            lbg_n <= '0';
                            bg_n <= '0';
                        ELSIF (lock_n = '0' OR lbr_n_reg2 = '1') AND
                              (selects /= no_selects) THEN
                            arb_state <= slow_down040;
                            lbg_n <= '1';
                            bg_n <= '1';
                        ELSE
                            arb_state <= only040;
                            lbg_n <= '1';
                            bg_n <= '0';
                        END IF;
                    WHEN slow_down040 =>
                        IF (xfer_done_n = '0') THEN
                            IF (lbr_n_reg2 = '0' AND lock_n = '1') THEN
                                arb_state <= both;
                                lbg_n <= '0';
                                bg_n <= '0';
                            ELSE
                                arb_state <= only040;
                                lbg_n <= '1';
```

Appendix D. Bus Arbitration PLD (CY7C335) (continued)

```
                        bg_n <= '0';
                    END IF;
                ELSE
                    arb_state <= slow_down040;
                    lbg_n <= '1';
                    bg_n <= '1';
                END IF;
            WHEN both =>
                IF (lbr_n_reg2 = '1') THEN
                    IF (selects /= no_selects) THEN
                        arb_state <= slow_down040;
                        lbg_n <= '1';
                        bg_n <= '1';
                    ELSE
                        arb_state <= only040;
                        lbg_n <= '1';
                        bg_n <= '0';
                    END IF;
                ELSE
                    arb_state <= both;
                    lbg_n <= '0';
                    bg_n <= '0';
                END IF;
            WHEN OTHERS => arb_state <= reset;
                           lbg_n <= '1';
                           bg_n <= '0';
        END CASE;
        IF (rst_n = '0') THEN
            arb_state <= reset;
        END IF;
    END PROCESS;
END operation;
```

Appendix E. Interrupt Synchronizing PLD (22V10D)

```
-- INTERRUPT PLD
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk             =  BCLK on 68040
--    iplx_n           =  IPLx* from VIC64
--    tmx              =  TM2-TM0 on 68040
--    board_reset_n    =  brd_rst_n from RESET PLD
--    fciack_n         =  fciack_n from ADDRESS DECODE PLD
--    gate_oe_n        =  gate_oe_n from BUS ARBITRATION PLD
--    xfer_done_n      =  xfer_done_n from TERMINATION PLD

--    normal_cycle_n   =  OE to '244's driving A3-A1 from 68040
--    a3, a2, a1       =  a3-a1 to VIC64
--    iplx_out_n       =  IPLx "bar" on 68040

ENTITY interrupt_ctrl IS
        PORT (bclk, ipl0_n, ipl1_n, ipl2_n : in bit;
              tm2, tm1, tm0 : in bit;
              board_reset_n, fciack_n : in bit;
              gate_oe_n, xfer_done_n : in bit;
              normal_cycle_n : out bit;
              a3, a2, a1 : inout x01z;
              ipl0_out_n, ipl1_out_n, ipl2_out_n : buffer bit);
END interrupt_ctrl;

use work.cypress.all;
use work.rtlpkg.all;

ARCHITECTURE operation OF interrupt_ctrl IS
     signal addr_oe, ipl0_n_reg, ipl1_n_reg, ipl2_n_reg : bit;
BEGIN

-- Synchronize the incoming ipl signals from the VIC64 to eliminate skew
     PROCESS BEGIN
          WAIT UNTIL bclk = '1';
               ipl0_n_reg <= ipl0_n;
               ipl1_n_reg <= ipl1_n;
               ipl2_n_reg <= ipl2_n;
     END PROCESS;

-- If the board is in reset, the ipl signals must be driven high to configure
-- the driver capability in the 68040.  Otherwise, the following equations
-- will keep the ipl signals from changing to the 68040 during an acknowledge
-- cycle and will synchronize them.  When the acknowledge cycle is finished,
-- the ipl signals will return to inactive state before reading the current
-- input values from the VIC64.

     PROCESS BEGIN
          WAIT UNTIL bclk = '1';
               IF board_reset_n = '0' THEN
                    ipl0_out_n <= '1';
                    ipl1_out_n <= '1';
```

Appendix E. Interrupt Synchronizing PLD (22V10D) (continued)

```
                ipl2_out_n <= '1';
            ELSIF xfer_done_n = '0' THEN
                ipl0_out_n <= '1';
                ipl1_out_n <= '1';
                ipl2_out_n <= '1';
            ELSIF fciack_n = '0' THEN
                ipl0_out_n <= ipl0_out_n;
                ipl1_out_n <= ipl1_out_n;
                ipl2_out_n <= ipl2_out_n;
            ELSE
                ipl0_out_n <= ipl0_n_reg;
                ipl1_out_n <= ipl1_n_reg;
                ipl2_out_n <= ipl2_n_reg;
            END IF;
    END PROCESS;

-- The normal_cycle_n signal is low most of the time to enable the a3-a1
-- signals from the 68040 to the VIC64.  However, if we are in an interrupt
-- acknowledge cycle, we would steer the tm2-tm0 signals from the 68040
-- onto the a3-a1 signals on the VIC64 since the tmx signals indicate which
-- interrupt level is being acknowledged

    normal_cycle_n <= NOT (fciack_n AND (NOT gate_oe_n));

    addr_oe <= NOT fciack_n;
    a3_map: bufoe port map(tm2, addr_oe, a3, open);
    a2_map: bufoe port map(tm1, addr_oe, a2, open);
    a1_map: bufoe port map(tm0, addr_oe, a1, open);

END operation;
```

Interfacing the CY7C611A with the VIC64

The popularity of the VMEbus and the Motorola 680x0 family of microprocessors has produced a large number of peripheral controllers with 680x0-compatible asynchronous local bus interfaces. Many of these parts are mature, proven, and inexpensive, making them attractive candidates for low-bandwidth I/O applications.

This application note describes an interface between the synchronous CY7C611A SPARC processor and asynchronous bus peripherals such as the Cypress Semiconductor VIC64 64-bit VMEbus interface chip. It is based on the design of a SPARC-based VIC64 VMEbus evaluation board developed by Cypress Semiconductor. Only the synchronous-to-asynchronous bus conversion logic is discussed within this application note; however, the full schematics of the board and all PLD design files are available from Cypress Semiconductor.

Related Documents

The reader may also wish to consult the following documents for additional information:

- *VIC068A/VAC068A User's Guide*
- *VIC64 and CY7C964 Design Notes*
- "Memory Protection and Address Exception Logic for the CY7C611A SPARC Controller" application note
- "Understanding the 361" application note
- Motorola's *MC6800 Family Reference*

With the exception of the Motorola document, these documents are available through your local Cypress Semiconductor field sales office.

Typical Asynchronous Bus Operation

Asynchronous buses operate using some type of handshake system. The processor presents or requests data from a peripheral and an acknowledge is generated by the selected device. The length of the processor cycle is determined by the performance level of the peripheral. The processor maintains a bus cycle until it receives an acknowledge.

With this type of bus, operation problems can occur if the processor attempts a cycle to an address region that does not select valid memory or peripherals. In this situation an acknowledge signal is not issued to the processor and the system operation halts.

To avoid this potential lock-up condition, most asynchronous bus protocols have a separate signal for acknowledging erroneous cycles. Assertion of this signal releases the processor from the pending bus cycle and can also be used to inform the system software that the bus cycle did not terminate properly. These cycles are typically known as bus error and memory exception cycles.

Memory Exception Cycles are Important

Asynchronous microprocessor buses are not unique in the inclusion of memory exception cycles. The CY7C601A and CY7C611A include a similar mechanism. In normal system operation, memory exceptions should not occur regularly. They can be used to furnish beneficial debug and system configuration information in some applications.

VMEbus applications where logic boards can be added and removed from systems often use the bus error mechanism to determine system configura-

tion. CPU board initialization software can *hunt* the VMEbus address regions, searching for other cards. Address regions that respond with normal acknowledge signals can then be further interrogated and initialized.

Overview of the CY7C611A Memory Interface

The CY7C611A is a 32-bit, four-stage, pipelined SPARC RISC integer processor. The processor is synchronous and, after initializing the pipeline, it can execute one instruction per clock cycle.

The CY7C611A memory interface consists of a group of signals that control memory loads/stores, pipeline control, and memory exception generation. These signals are listed in *Table 1*.

$\overline{\text{MHOLD(A/B)}}$

These two signals are logically ORed together within the CY7C611A. Asserting either of these signals (Low) freezes the processor's pipeline, causing the processor to remain on the same execution cycle. The $\overline{\text{MHOLDA}}$ and $\overline{\text{MHOLDB}}$ signals allow the processor to communicate with slow peripherals.

$\overline{\text{MDS}}$

MDS is used to strobe data or instructions into the processor after the pipeline has been frozen by the assertion of $\overline{\text{MHOLD(A/B)}}$. Asserting $\overline{\text{MDS}}$ with the pipeline frozen enables the processor to clock the information present on the external data bus into the processor. MDS is also used to strobe in the $\overline{\text{MEXC}}$ signal.

$\overline{\text{MEXC}}$

Asserting this signal (Low) informs the processor that the memory system could not supply the data or instruction requested. When the signal is asserted, either a data or instruction access trap occurs. The type of trap directly corresponds to the type of memory cycle in progress. $\overline{\text{MEXC}}$ is strobed into the processor by asserting the $\overline{\text{MDS}}$ signal.

Table 1. CY7C611A Memory Interface Signals

Name	Description	Type
$\overline{\text{MHOLD(A/B)}}$	Memory Hold A/B	Input
$\overline{\text{MDS}}$	Memory Data Strobe	Input
$\overline{\text{MEXC}}$	Memory Exception	Input
$\overline{\text{INULL}}$	Integer Unit Nullify	Output
$\overline{\text{WE}}$	Write Enable	Output
$\overline{\text{WRT}}$	Advanced Write	Output
$\overline{\text{RD}}$	Read Access	Output

INULL

The assertion of INULL (High) indicates that the memory cycle in progress is being nullified. Memory cycles are nullified when the processor determines the the current address is invalid or that the information being read is not required. This improves performance because no time is wasted communicating with slow peripherals or reloading cache line data that is not needed. INULL is asserted by the processor in the following situations:

- During the second cycle of any store operation. The same address is presented on the first and second cycle of all store operations, the second occurrence is nullified because it is not truly the *next* address being requested by the processor.
- On all traps. This nullifies the third instruction fetch after the trap is encountered, because the processor vectors to the appropriate trap handler.
- On a load with the hardware interlock active.
- On JMPL and RETT instructions.

$\overline{\text{WE}}$

Write Enable (active Low) indicates that the processor is performing a store operation. This signal is asserted in the second clock cycle of the store operation, the same cycle that the store data is presented.

WRT

Advanced Write (active High) notifies the external control logic that a store operation is in progress. The processor asserts this signal on the first cycle of the operation, before the data is available.

RD

Read Access (active High) indicates that a load cycle is in progress.

CY7C611A Load and Store Cycles

Two general bus cycles, load and store, are described at a high level of abstraction within this section. Many variations of these cycles exist.

When loading data, the processor supplies the address information on the rising edge of a the processor clock and expects the data on the next rising edge. The Read signal (RD) remains active (High) during the cycle with $\overline{WE}$ and WRT inactive (High and Low respectively).

The process for storing data is similar, but one additional clock cycle occurs before the processor presents the data. On the first cycle of the store the address is presented, RD is driven inactive (Low), and WRT is driven active (High). On the second clock cycle, the store address is again placed on the bus, WE is asserted (Low), WRT is deasserted (Low), and the data is placed on the bus. INULL becomes active after the falling edge of the second clock cycle, nullifying the second occurrence of the store address.

Figures 1 and *2* show Store Single and Load Single CY7C611A bus cycles. In general, the address execution unit operates one clock cycle ahead of the data unit. The cycles shown assume that the peripherals or memory are capable of operating at the performance level of the processor. The pipeline is never frozen using MHOLDA or MHOLDB, and both cycles terminate normally without generating memory exceptions.

Overview of the VIC64 Asynchronous Interface

The Cypress Semiconductor VIC64 is compatible with the 680x0 asynchronous microprocessor bus. It is a 64-bit VME interface chip capable of performing D16, D32, and D64 block transfers on the VMEbus at transfer rates up to 70 Mbytes/sec. The VIC64 and its associated control logic are also capable of performing Direct Memory Access (DMA) operations during VMEbus block transfers. VIC64 DMA operations generate 680x0-compatible bus cycles to transfer data to and from local memory.

The basic control signals required to communicate with a VIC64 or other generic 680x0-style peripherals are listed in *Table 2*.

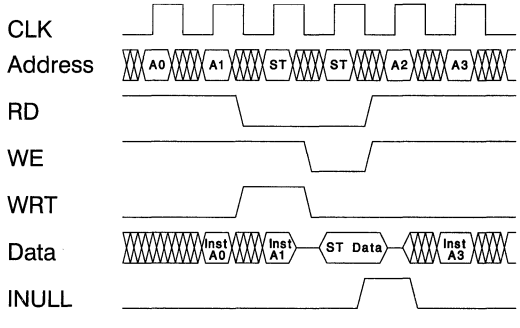


Figure 1. CY7C611A Store Single Operation

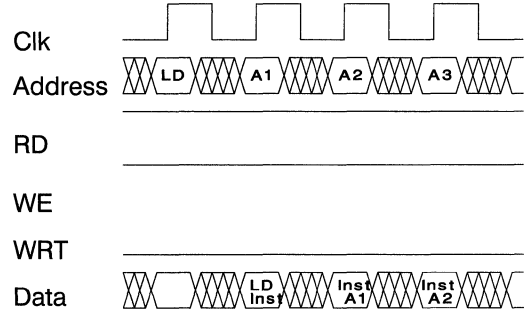


Figure 2. CY7C611A Load Single Operation

Table 2. 680x0 Basic Control Signals

Name	Description	Type
AS	Address Strobe	(Normally) Input
DS	Data Strobe	(Normally) Input
R/W	Read Write	(Normally) Input
DSACK0/1	Data Acknowledge	(Normally) Output
BERR	Bus Error	(Normally) Output

The signal types, input or output, have been referenced in a normal operating mode for dumb peripherals. Since the VIC64 is also capable of becoming a *bus master* during local DMA transfers, it can source AS, DS, and R/W as well as receive these signals. This also holds for the output signals DSACK1/0 and BERR. If the VIC64 is generating the bus cycle, these control signals become inputs.

AS

Address Strobe is asserted (Low) at the beginning of a bus cycle to indicate that a valid address is currently on the address bus. The address must remain constant while Address Strobe is active. Address Strobe remains active for the length of the bus cycle. On the VIC64 this signal is named Processor Address Strobe (PAS).

DS

The assertion of Data Strobe informs the receiving peripheral device or memory that it may place data on or extract data from the bus.

R/W

The Read/Write signal indicates the type of cycle in progress. This signal is High for read cycles and Low for write cycles.

DSACK0/1

The DSACK0/1 signals are driven by the peripheral device to tell the device performing the bus cycle that the data has been accepted or is available on the bus. Bus cycles persist until an acknowledge or BERR signal is detected. There is no limit to the length of this type of bus cycle. Many 680x0 peripheral devices have only a single acknowledge, often named *DTACK*.

VIC64 has two DSACK signals, 0 and 1, which adhere to the Motorola dynamic bus sizing convention and report the bus width, (8, 16, or 32 bits), of the peripheral acknowledging the bus cycle.

BERR

Asserting BERR terminates a pending bus cycle and forces the processor to trap to an exception handler. This signal terminates erroneous bus cycles. Many systems have bus timeout timers that monitor the length of all bus cycles and assert BERR if a cycle persists for the timeout period.

680x0 Asynchronous Read and Write Cycles

As with the corresponding section on the CY7C611A load and store cycles, the read and write cycles within this section are only described at the High level. Many variations of these cycles exists. Refer to the *VIC068A/VAC068A User's Guide* or the Motorola microprocessor documentation for more information.

Write cycles begin with an address being placed on the bus by the controlling processor or peripheral. The R/W signal is driven Low to indicate a write cycle. Address Strobe is asserted (Low), denoting the beginning of the cycle. One clock later, after the data has been placed on the bus, DS is asserted (Low). All signals remain stable in this state until a normal acknowledge, DSACK0/1, or error acknowledge is received (Low).

Reads cycles operate in a similar manner. An address is placed on the bus by the controlling peripheral and R/W is driven High to indicate a read cycle. AS and DS are driven Low simultaneously, informing the peripheral that data can be placed on the bus. These signals remain in this state until an acknowledge of some sort is received.

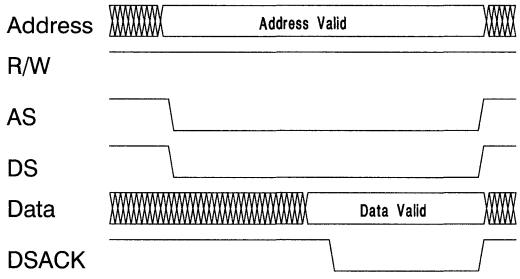


Figure 3. 680x0-Compatible Read Cycle

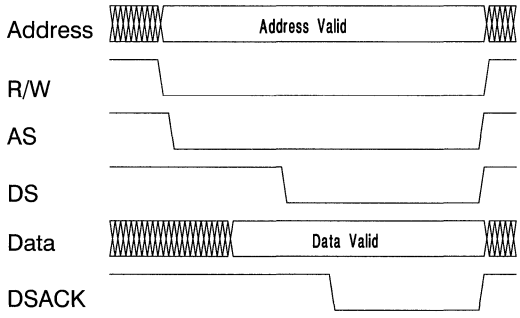


Figure 4. 680x0-Compatible Write Cycle

Figures 3 and *4* show typical bus cycles for asynchronous 680x0 peripheral devices like the VIC64.

Clear Differences in Cycle Types

As can be seen with even a cursory view of the two styles of bus cycles, interfacing between the CY7C611A and peripherals like the VIC64 can be challenging. In general, the problem is slowing the CY7C611A down to operate with the peripheral. This can be accomplished in a number of ways, each having its own set of considerations.

Pipeline Freezing Using MHOLDA/B

Per design, the CY7C611A contains control logic that allows the execution unit to be held for communication with slow memory devices or peripherals. The logic sequences required to suspend execution have some tight timing requirements. If an MHOLD signal is not asserted quickly enough, the processor advances to the next cycle. The CY7C611A, unlike the CY7C601, does not have an MAO pin. Therefore, if the processor does advance to the next cycle, there is no way to have it place the last address back on the bus. This can become a significant problem. Obviously other undesirable situations can occur when control logic does not or cannot meet necessary timing constraints.

These potential problems can be overcome by using high-performance logic like the CY7B336, CY7B337, CY7B338, and CY7B339 family.

Clock Stretching

Another method of interfacing the CY7C611A to slow memory and peripherals is a procedure known as clock stretching. The CY7C611A is a fully static microprocessor. This furnishes a simple method for slowing the processor down, simply by delaying or changing the duty cycle of the clock. The processor can be held within an execution state without asserting $\overline{\text{MHOLDA/B}}$. This technique allows execution to resume without strobing data into the processor with $\overline{\text{MDS}}$.

This procedure works well for peripherals with fixed access times. When the bus cycle begins, the clock is stretched. When the peripheral has completed the data transaction the clock is allowed to advance.

There are two subtle problems with this method of interfacing:

- Additional logic is required to operate with peripherals that are truly asynchronous in nature
- Memory exceptions cannot be generated because they require $\overline{\text{MDS}}$, $\overline{\text{MEXC}}$, and $\overline{\text{MHOLD}}$

Each of these problems becomes a significant issue when interfacing to the VIC64. Using the VIC64 to perform single-cycle processor transfers across the VMEbus has no guaranteed cycle time. The length of the cycle is directly dependant on the performance level of the slave plus the acquisition time required to obtain the VMEbus. Therefore, using a fixed clock-stretch cycle time would either be too short for slow slave boards, or a significant performance barrier when communicating with faster boards.

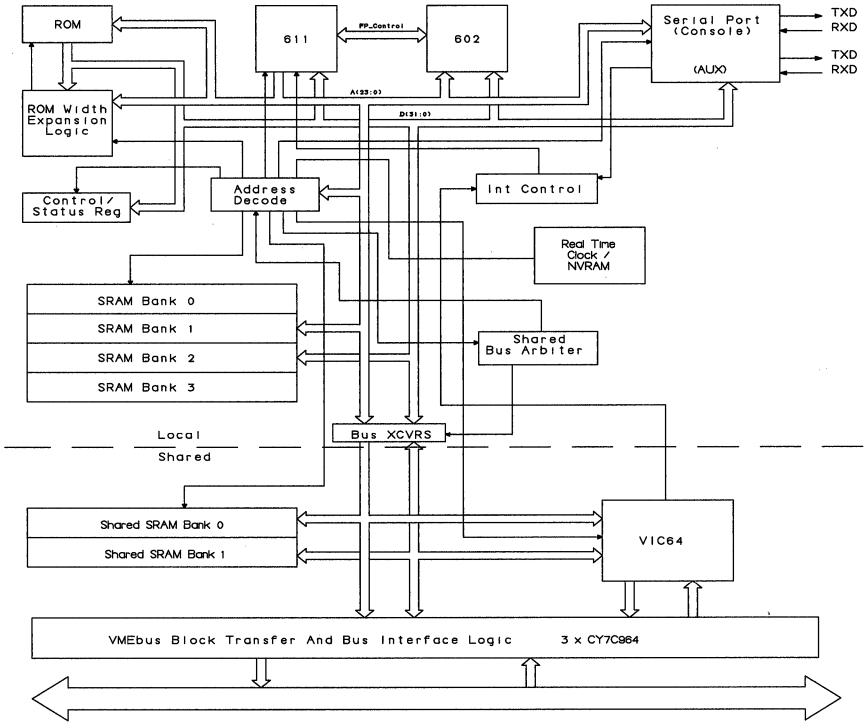


Figure 5. CY7C611A / VIC64 VMEbus Board Block Diagram

Bus errors are also an integral part of the VMEbus and the VIC64's operation. The inability to use this feature would significantly limit the functionality of many systems. Mapping this function into an interrupt is not desirable because if interrupts are disabled, or if interrupt latency is encountered because higher-priority interrupts are pending, the software's ability to determine the cycle that caused the error is hampered.

CY7C611A/VIC64 VMEbus Board

The CY7C611A/VIC64 evaluation board is a typical single-board computer with the following features:

- 25-MHz CY7C611A embedded-control SPARC RISC processor
- 25-MHz CY7C602 floating-point unit
- 64 Kbytes to 4 Mbytes of local SRAM
- 64 Kbytes to 2 Mbytes of dual-port SRAM
- 128 Kbytes to 512 Kbytes of EPROM
- MC68681 DUART
- 2 Kbytes of non-volatile storage
- Time-of-day clock calendar
- Split address and data bus for high-performance VMEbus block transfer operation

The block diagram of the board is shown in *Figure 5*. The local SRAM on the board operates at zero wait states, removing the need for an instruction or data cache. With the exception of the local SRAM and a

system control/status register, all other peripherals operate using asynchronous 680x0-style bus cycles.

Having all peripherals operate using one of the two cycle types simplifies the interface and control logic. The 680x0-style cycle is essential since the VIC64 and MC68681 DUART communicate on this type of interface. The shared SRAM also needs to operate using this type of cycle to be compatible with the VIC64 during VMEbus block transfer DMA operations. It is then simple to adapt other slow peripherals (ROM, non-volatile SRAM, and Time-of-Day clock) to the slow, 680x0-style bus cycle.

The CY7C611A-to-680x0 Bus Converter

As discussed in the previous sections, there was a strong desire to build an interface that was logically simple but preserved memory exception capability. The scheme was implemented as a hybrid technique using the CY7C611A pipeline freezing and memory exception logic along with a clock-stretching technique.

The control logic is implemented within two PLDs, a CY7C361 and a 22V10B, operating as pseudo master slave devices. The logic is split between two devices because of other functionality needed on the board, which is well suited for the CY7C361. If these other functions were removed from the CY7C361, the entire synchronous to asynchronous conversion logic could fit within the CY7C361. However, the CY7C361 on the CY7C611A/VIC64 board provides:

- Generation and control of clocks for the processor and peripherals
- Local bus arbitration for the VIC64 and CY7C611A
- Synchronization of asynchronous signals, which is needed for the slave 22V10B

This bus conversion scheme operates as follows. The processor begins execution and an address is presented, latched, and decoded. If the address region decodes to a slow 680x0-compatible cycle, the clock to the processor and control logic is stretched. If the cycle terminates normally, the clock is re-enabled to the processor and to control logic, which advances to the next execution cycle. If the cycle terminates in a memory exception or bus error, $\overline{\text{MHOLDA}}$ is asserted to the processor, freezing the pipeline, and the clock is re-enabled. With the pipeline frozen and the processor and control logic clock running, $\overline{\text{MEXC}}$ and $\overline{\text{MDS}}$ are asserted to the processor, generating the exception.

Clock Control Using The CY7C361

To simplify interface design and maximize performance, microprocessor control logic typically needs to operate at twice the clock frequency of the processor. Even with the relatively slow 25-MHz clock frequency of the CY7C611A, routing, managing skew, and operating TTL control logic at 50 MHz can significantly increase the complexity of a design.

To eliminate this problem, the CY7C361 was selected as a clock-generation device. The CY7C361 is an ultra high speed PLD that features an internal clock doubler, double input registers for metastable hardening of asynchronous inputs, and 32 general-purpose state macrocells. While this is not a typical application for a PLD, the CY7C361 has a pin-to-pin skew of 2 ns maximum.

Operating the CY7C361 at 50 MHz externally and 100 MHz internally allows the generation of three different 25-MHz clocks. While the system still requires a 50-MHz clock, the CY7C361 is the only device operating from it, simplifying routing and termination problems. Since no other device on the board operates from the 50-MHz clock, no relationship needs to be maintained between the CY7C361 clock input and output pins, removing the clock-to-output propagation delay from the timing analysis. The 25-MHz clocks operate all sequential logic on the board with the exception of the 3.68-MHz clock needed by the MC68681 DUART for baud-rate generation.

The Clock-Generation Machine

The clock-generation state machine within the CY7C361 has the following input and output signals:

NNULL (Input)

This synchronous input is a conditioned active-Low signal formed by combining the CY7C611A INULL and the CY7C602A FNULL signals. FNULL is the corresponding nullify signal from the floating-point unit. It operates in the same manner as INULL. The NNULL signal is used to filter out the nullifies that occur during every store cycle. The store nullifies were a don't care for the board's control logic since the signal is generated to nullify the second occurrence of the store address.

$$\overline{\text{NNULL}} = (\text{INULL OR FNULL}) \text{ AND LWE}$$

INULL and FNULL are active High and LWE is simply the latched WE signal from the CY7C611A. LWE is latched on the rising edge of CPUHCLK.

LWRT (Input)

This synchronous input is the latched WRT signal from the CY7C611A. This signal is latched on the rising edge of CPUHCLK.

$\overline{\text{MHOLDA}}$ (Input)

This is the synchronous CY7C611A $\overline{\text{MHOLDA}}$ signal. This signal is generated by the 22V10B that generates Motorola-style bus cycles.

DONE (Input)

A synchronous signal generated elsewhere within the CY7C361 that indicates that a Motorola bus cycle has been acknowledged. All acknowledges returned from the board are asynchronous signals. Double input registers on the CY7C361 are used to synchronize it for state logic use. DONE is active Low and is asserted if the cycle terminates normally or in a bus error.

HOLD (Input)

HOLD is a synchronous signal from the address decoding logic indicating that the selected peripheral requires a slow, 680x0-style bus cycle and that the CY7C611A and control logic clock must be stretched.

CPUCLK (Output)

This is a free running 25-MHz clock that is used for much of the sequential control logic. Although the name may imply it, this clock is not used by the CY7C611A CPU.

CPUHCLK (Output)

This is a 25-MHz stretched version of CPUCLK. It is the clock used to control the CY7C611A, CY7C602A, and address decode/latch logic. This clock is stretched by the CY7C361 if the address decoding logic reports that a slow, asynchronous cycle should be performed. When this clock is operating, it is always in phase with CPUCLK.

CPU90 (Output)

This is a 25-MHz free-running clock that lags the CPUCLK by 90 degrees (1/2 cycle). This clock, in conjunction with CPUCLK, provides the board control logic with a time base with 10 ns of resolution.

START (Output)

The assertion of START (Low) informs the slave 22V10B state machine that a 680x0 cycle should begin. This signal is not actually an output of the state machine, but of external state logic that is controlled by this state machine.

The 680x0 cycle cannot start at the beginning of the stretched clock cycle because of the latency associated with the assertion of INULL and FNULL from the CY7C611A and CY7C602A. If the NNULL signal is not asserted 40 ns after the clock stretching has started, the bus cycle is deemed valid and the START is asserted.

The state diagram of this machine is shown in *Figure 6.*

The transition equations for this machine are:

1. (State3) OR (HOLD AND /MHOLDA AND LWRT)
2. State3 AND /HOLD AND /MHOLDA AND /LWRT
3. State7 AND /DONE AND NNULL
4. (State7) OR (DONE AND /NNULL)

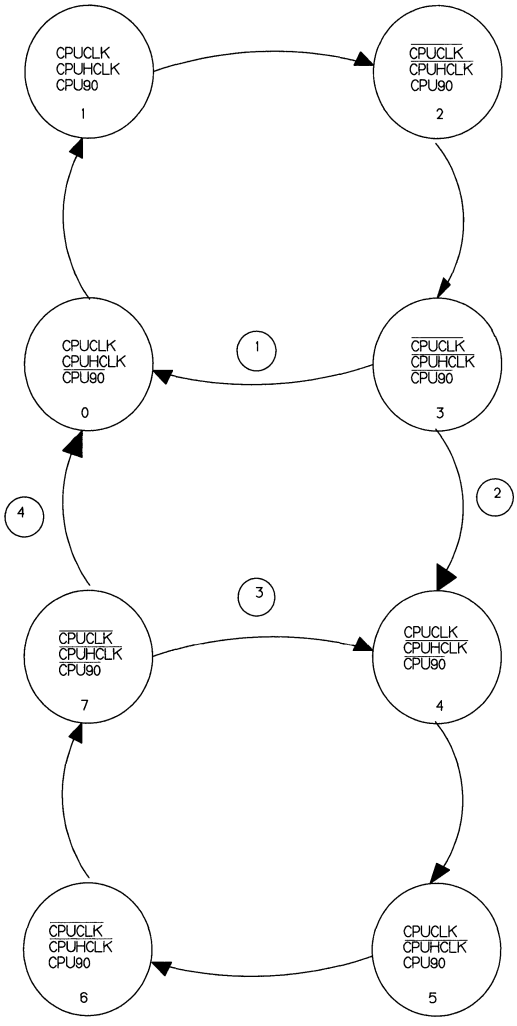


Figure 6. Clock-Generation State Machine

Clock-Stretch Machine Operation

This machine has two main paths of operation. The first is a sequence in which all three clock outputs are operating, sequencing through states 0, 1, 2, 3, and back to 0. The second is a clock-stretched path sequencing through states 4, 5, 6, 7, and back to 4. This machine enters state 0 at the deassertion of reset and therefore always begins execution generating all clocks.

While within state 3, the machine samples the HOLD, $\overline{\text{MHOLDA}}$, and LWRT (Latch Advanced Write signal). HOLD High indicates that memory address on the bus is not selecting a slow device and that the clock should not be stretched. $\overline{\text{MHOLDA}}$ Low in this circuit indicates that a memory exception has occurred, and that the processor clock should continue to operate. The clock must be re-enabled so that $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ can strobe the memory exception condition into the device.

The third signal sampled is Latched Advanced Write (LWRT). When this signal is High it indicates that the processor is starting a store cycle. The CY7C611A does not provide data to be stored until the second clock cycle of the operation. Therefore the processor must be advanced by at least one clock to place the data on the external bus. LWRT High and MHOLDA Low cause the processor clock to continue operating even if the address decoding logic asserts HOLD Low, indicating that the cycle should be stretched.

If HOLD is asserted (Low) and neither LWRT or MHOLDA are in their active states, then the state machine moves to state 4 and the clock is disabled to the processor and control logic. The other output clocks (CPUCLK and CPU90) continue to operate as the machine sequences through states 4, 5, 6, and 7. State 7 is also a decision-making state within this machine. At this point the machine either continues stretching or re-enables the processor and control logic clock. The clock is only re-enabled if either of two conditions (NNULL or DONE are detected active (Low)) is true.

NNULL is asserted if the current cycle is not a store cycle and the CY7C611A or CY7C602A nullifies the cycle. The processor does not generate the INULL or FNULL until late in the cycle. Therefore nullified asynchronous bus cycles end up being stretched for 40 ns before this is determined. If the stretched bus cycle is nullified by the CY7C611A or CY7C602A, the NNULL is asserted before the machines samples the signal in state 7. If NNULL has not been asserted upon entering state 7 for the first time after clock stretching has begun, START is as-

serted (Low) . This signals the slave 22V10B that a 680x0 style cycle should begin.

680x0 Bus Cycle Machine

A Mealy state machine implemented in a 22V10B performs the 680x0-compatible asynchronous bus cycle and asserts $\overline{\text{MHOLDA}}$, $\overline{\text{MDS}}$, and $\overline{\text{MEXC}}$ to the CY7C611A if a bus error is detected. This machine uses the CY7C361 to synchronize all asynchronous signals and therefore operates in a totally synchronous environment. This simplifies the implementation of the machine and enhances performance.

The input and output signals for the machine are:

START (Input)

This signal is asserted (Low) by the CY7C361 clock control state machine to indicate that a Motorola bus cycle should start. This signal remains asserted until the bus cycle completes.

LRD (Input)

This is the latched Read Access (RD) signal from the CY7C611A.

SPARC_WB (Input)

This is an output from the local bus arbiter for the board. If the asynchronous bus cycle is accessing something on the shared half bus, this signal is asserted by the address decode logic. If this signal is active (Low) the bus cycle must not begin until the grant signal, SPARC_GB, is asserted, granting access to the shared bus.

SPARC_GB (Input)

This active-Low signal is the bus grant signal from the local bus arbiter.

DONE (Input)

This is the synchronous active-Low signal from the CY7C361, indicating that some form of asynchronous cycle acknowledge has been received.

BERR (Input)

An asynchronous active-Low input that is combined with DONE for synchronization.

AS (Output)

This is the active-Low 680x0 compatible address strobe.

DS (Output)

This is the active-Low 680x0 compatible data strobe

$\overline{\text{MHOLDA}}$ (Output)

This is the $\overline{\text{MHOLDA}}$ signal to freeze the pipeline of the CY7C611A and CY7C602A.

$\overline{\text{MDS}}$ (Output)

This output is the $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ signals to the CY7C611A. Since this bus control cycle only requires $\overline{\text{MDS}}$ for memory exception cycles, it was possible to reduce these two into a single output on the machine.

The state diagram of the machine is shown in *Figure 7.*

The transition equations for this machine are:

1. (/START AND /LRD AND /SPARC_WB AND /SPARC_GB) OR (/START AND /LRD AND SPARC_WB)
2. (/START AND LRD AND /SPARC_WB AND /SPARC_GB) OR (/START AND LRD AND SPARC_WB)
3. /DONE AND /BERR
4. /DONE AND BERR

Machine Operation

This machine resets to state 0 and waits for the assertion of the START signal from the CY7C361. When this signal is active, the machine samples the states of the CY7C611A Latched Read Access signal (LRD), the Local Bus Request signal (SPARC_WB), and the Local Bus Grant signal (SPARC_GB). The state of LRD instructs the 22V10B to perform either a read or write cycle. If SPARC_WB is asserted (Low), then the

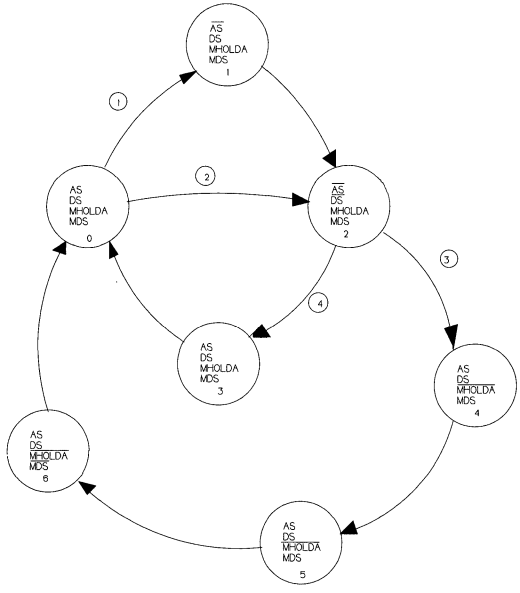


Figure 7. 680x0 Bus Cycle State Machine

peripheral device being accessed is on the shared section of the board. The bus cycle cannot start until a grant has been issued by the local bus arbiter. When SPARC_GB is asserted (Low), the shared bus has been granted to the CY7C611A. Read or write cycles that access peripherals on the local section of the card (CY7C611A access only) do not need permission from the local bus arbiter. CY7C611A local accesses can occur simultaneously with VIC64 local DMA accesses.

When the conditions have been met to start a cycle, AS strobe is asserted (Low). If the cycle is a read, DS is also asserted (Low). On write cycles, the assertion of DS is delayed one clock cycle to mimic the 680x0 cycle. This may not be necessary for many peripherals since, unlike Motorola processors, the CY7C611A has already placed the data on the data bus before the assertion of AS.

The machine moves to state 2 were it waits for the assertion of DONE (Low) from the CY7C361. DONE is generated by combining the board's asynchronous peripheral acknowledge and bus error signals. This combined signal is then run through the double input register structure of the CY7C361 to synchronize it. Double registering these asynchronous signals with the CY7C361 is the most efficient manner of synchronization as these registers are being clocked internally at 100 MHz. The entire double-register synchronization process takes only 20 ns. DONE is further qualified with the appropriate 25-MHz clock from the CY7C361 so that it can be considered completely synchronous to the 22V10B.

When the 22V10B detects DONE asserted, it samples the BERR input. If BERR is inactive (High), then the cycle terminates normally. The machine drives AS and DS inactive and advances back to through state 3 to state 0 to prepare for the next cycle. State 3, a delay state, is necessary to allow the control logic recovery time before the next cycle begins. This is a Mealy machine and removing state 3 would allow situations to occur were AS and DS would not meet the minimum High times required by slow peripherals. Refer to the waveforms on the following pages, which show the control signal sequencer for normally terminated and memory exception cycles.

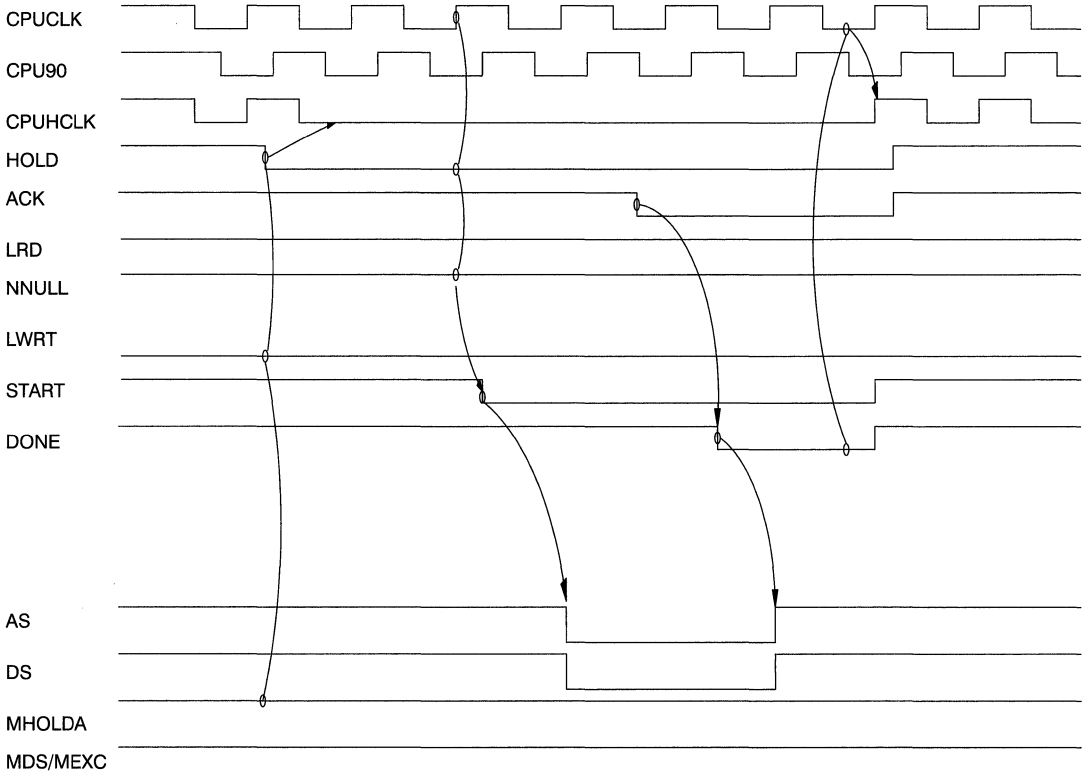
If BERR is active (Low) when DONE is asserted, the asynchronous cycle is terminated in a bus error or memory exception. The 22V10B asserts MHOLDA (Low) to the CY7C611A freezing the pipeline. The assertion of this signal informs the CY7C361 to re-enable clocking to the CY7C611A and control logic so that the exception can be strobed in using $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$. $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ are then asserted simultaneously to the CY7C611A, indicating that a memory exception has occurred. Since memory exceptions are the only cycles that freeze the CY7C611A pipeline, $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ are always asserted simultaneously. This allows the generation of a single signal, rather than two, freeing up an output on the PLD.

Conclusion

This hybrid bus conversion has worked well on the CY7C611A/VIC64 VMEbus board. Using the CY7C361 as a clock-generation device, allowing all of the logic on the board to operate from the relatively slow 25-MHz clocks, greatly simplified the timing analysis without sacrificing performance. The CY7C361 also provides logic functions that are not discussed within this application note. In many applications it may be possible to move the slave 680x0 bus-cycle generation state logic into the CY7C361. This would reduce the bus conversion logic to a single device.

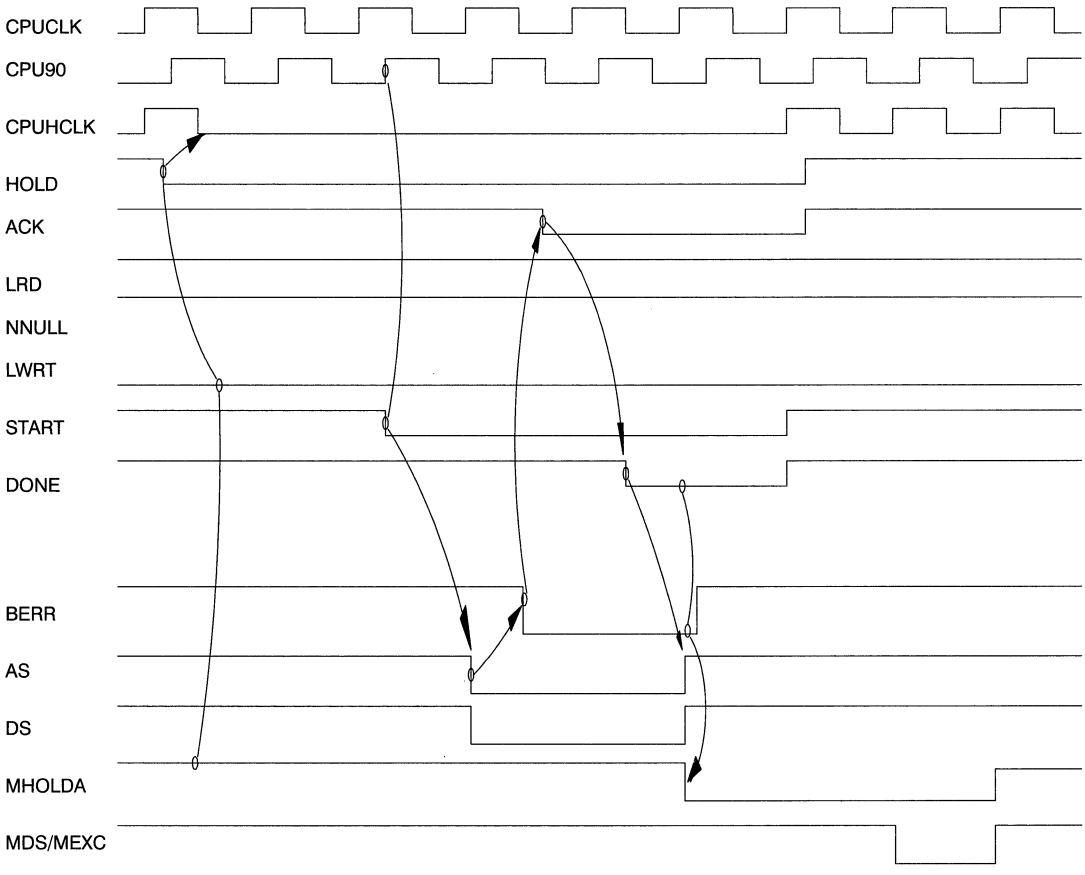
Output Waveforms

CY7C611A to 680x0 Normally Terminated Cycle



Output Waveforms (continued)

CY7C611A to 680x0 Memory Exception Cycle



An SVIC to 68020 Arbiter Design

Introduction

VME board functionality and their interfaces vary quite widely from application to application. The most complex type of VME interface is a VMEbus System Controller, which has complete VME master and slave capability and is the VME Interrupt handler. There are many devices on the market that can satisfy this need and Cypress has devices that can perform this function, namely the VIC068A and VAC068A 32-bit VMEbus Interface Controllers. In addition to this, the VIC64 provides all the functionality of the VIC068A but with the addition of D64 VME block transfer capability.

However there are many applications that do not require the complexity of the VIC/VAC products. These VME boards might often be slave-only type applications. Cypress has introduced the Slave VIC devices (SVIC for short), the CY7C960 and CY7C961. These devices are simple VME interface controllers, without having any of the complexity of being a VME System Controller or VME Interrupt Handler. The CY7C960 is a slave and the CY7C961 is a slave with DMA master.

Typical applications for slave-only products are memory boards and I/O boards. Memory boards can be as diverse as SRAM, DRAM, UVEPROM or FLASH EPROM (in, say, solid state mass storage). The I/O type applications could be for Ethernet, SCSI, FDDI, MIL STD 1553, RACE, Parallel/Serial I/O or even a VSB bridge. Memory boards do not require the use of a microprocessor, as they invariably rely on the VME master to initiate either a read or a write. Local timing and bank switching, etc., can be controlled with programmable logic devices (either CPLDs or FPGAs) and a microcontroller may also be needed.

Again, most I/O applications operate in a similar way to the memory card, in that reads and writes are initiated by the VME master. However, if there are several interfaces on the I/O card, then a local microprocessor may be useful for reducing the overhead of the main system processor. If the local processor could take over much of this overhead, such as pre-processing, then the VME master may only be required to extract data on a block transfer basis. Such a set-up could allow data to be transferred at up to 80 Mbytes/second.

This application note provides an example of how to design the arbiter between one of the SVIC devices and a microprocessor. It has been assumed that the local microprocessor is a Motorola 68020. The arbitration associated with this device is fairly standard with most of the Motorola processors. Also, the Motorola processors are well suited to the VMEbus, requiring some byte swapping for 8- or 16-bit transfers, but little else.

The SVIC Devices (CY7C960 and CY7C961)

Features List

- 80 Mbyte per second Block Transfer Rates
- VME64 compliance (A64, A40, A32, A24, A16)
- Auto Slot ID
- All standard VMEbus transactions implemented
- VMEbus Interrupter
- No Local CPU necessary
- Programmable from VMEbus or Serial PROM

- DRAM Controller including refresh
- Local I/O Controller
- Flexible VMEbus address scheme
- User-configured VMEbus personality
- Limited VME Master support (CY7C961 Only)
- TQFP, PQFP, CQFP packaging

Slave VIC Operational Overview

The Slave VMEBus Interface Controller (SVIC) provides the board designer with an integrated, full-featured VME64 Interface. This device can be programmed to handle every transaction defined in the VME64 specification (as a slave device). The SVIC contains all the circuitry needed to control large DRAM arrays and local I/O circuitry without the necessity of complex programmable logic to drive the timing. There are no registers to read or write and no complex command blocks to be constructed in memory. The SVIC simply fetches its own configuration parameters during the power-on reset period. After reset, the SVIC responds to VMEBus activity and local circuitry transparently.

The SVIC acts as a bridge between the VMEbus and the local DRAM, as well as the local I/O. The VMEbus control signals are connected directly to the SVIC. The VMEbus address and data signals are connected to address and data transceivers that are controlled by the SVIC. Typically, these are devices such as the FCT543T. The SVIC may also be seamlessly connected to the ideal companion device, the CY7C964 VMEbus Interface Logic Circuit from Cypress. For an A32/D32 application, there is one CY7C964 required per byte width of address and data. Thus a total of four devices are required—maximum. The CY7C964 provides a slice of data and address logic that has been optimized for VME64 transactions. As well as providing the required drive strength and timing for VME64 transactions, the CY7C964s contain all the circuitry needed to multiplex the address/data bus functions for multiplexed VMEbus transactions. The CY7C964 contains counters and latches needed during block transfer operations. It also contains the address comparators that are used in the board's Slave Address Decoder. For an A32 or larger application four CY7C964 devices are required. For A24/D32 applications, then, three CY7C964s and the SVIC are required. For A24/D16 applications, only two CY7C964s, the SVIC and an FCT543T (or equivalent) are required. For A16/D16 applications, only two CY7C964s and the SVIC are required.

VMEbus transactions supported by the SVIC include D8, D16, D32 (include unaligned transfers (UAT)), MD32, D64, A16, A24, A32, A40, A64 single cycle and block transfer reads and writes.

Figure 1 shows the internal blocks that comprise the SVIC. The architecture includes several functions that remove most of the VMEbus problems from the board designer's shoulders. All VMEbus signals are handled automatically. The user has to program the Region AM table during configuration and then the SVIC handles the transactions as defined by the table set-up. Local circuitry is simplified by the Refresh Controller, the DRAM Controller, and the output pattern table. Block transfers are supported by the local address controller together with the CY7C964 circuitry (if used). Local timing is determined during initial configuration and the handshaking is determined from the Data Byte Enable Controller. Local interrupts are supported through the VME Interrupt Interface. The SVIC contains an internal Power-On Reset circuit and also responds to the VME SYSRESET* signal.

Design Example

Introduction

The design example has been chosen as a typical example of a VME board design. *Figure 2* shows that the design is based on a Motorola 68020 microprocessor. The processor has boot software located in the Boot EEPROM. After setting up the stack and implementing the reset exception routine, the processor would normally jump to running code from the EPROM. This will allow the processor to set up the DUART, RTC and any other programmable functions within the peripherals. This may well include setting up the SVIC, even though this is normally performed by either a serial EPROM or, alternatively, via the VMEbus.

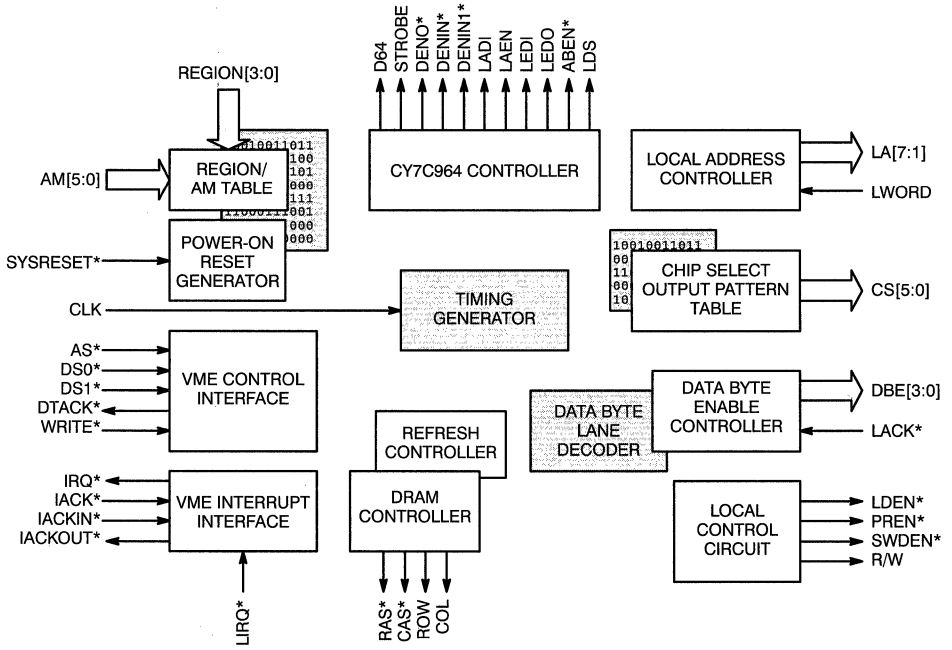


Figure 1. SVIC Block Diagram

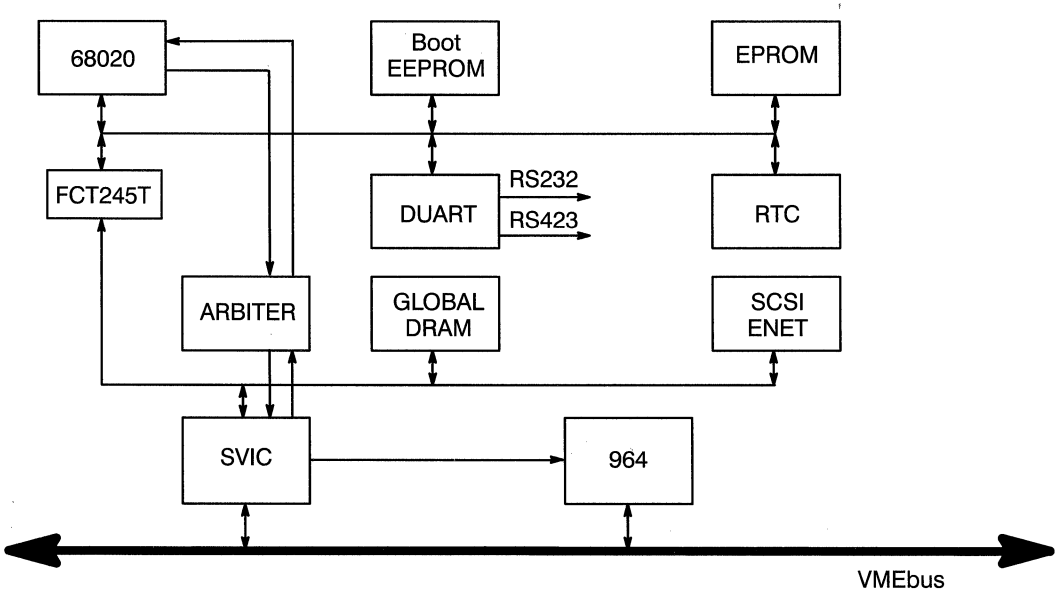


Figure 2. Typical VMEbus Design

There are two potential local bus masters in this design—either the 68020 or the SVIC. The task of arbitration (i.e., determining which master has control) is done by the Arbiter. The design is based on a single bus structure. The presence of the FCT245T devices reduces capacitive bus loading to maintain better performance.

Overview of the Motorola 68020

The Motorola 68020 was the first 32-bit implementation of the M68000 family of microprocessors from Motorola. The 68020 is object-code compatible with other members of the 68000 family. The non-multiplexed bus structure of the 68020 uses 32 bits of data and 32 bits of address. This lends itself very well to the VMEbus architecture, which is based on a 32-bit data and 32-bit address structure. For the purposes of data transfers, a D64 block transfer on the VMEbus is automatically split up into two 32-bit data transfers on the local bus, which keeps the 68020 compliant even in a D64 environment as provided by the SVIC.

The 68020 provides support for a dynamic bus sizing arrangement where the processor can transfer operands to or from devices while dynamically allowing the local bus logic to determine the port width for the 68020 on a cycle by cycle basis. This allows for access to devices of differing port width without the software engineer having to take special care over data alignment restrictions.

68020 Arbitration Methodology

Bus arbitration is the process in which a device on a bus may become bus master. The 68020 has a bus controller that controls the bus arbitration for the local bus that the processor sits on. This means that the 68020 has the lowest priority on the local bus. The design of the 68020 allows for a single bus master to be on the local bus at any one time. This includes an external device or the processor itself.

68020 Bus Arbitration Sequence

The bus arbitration sequence for the 68020 is:

1. An external device asserts the BR* signal.
2. The processor asserts the BG* signal to indicate that the local bus will become available at the end of the current bus cycle.
3. Once the local bus is released, the external device asserts the BGACK* signal back to the processor to indicate that it has assumed bus mastership.

The 68020 Bus Request Mechanism

Any devices on the local bus that are capable of becoming a local bus master must assert the BR* signal to the processor. The BR* signals from many potential bus masters can be arranged in a wire-ORd fashion even though they need not be open collector signals. (Rescinding three-statable signals are preferable to wire-ORd as the circuit does not rely on RC effects for the signal to drift up to an inactive level.) Once BR* has been asserted to the processor, this indicates that some external device wants control of the local bus. The design of the 68020 is such that it is always at a lower level bus priority than the external device that wants control of the bus and so the processor is compelled to relinquish the bus after it has completed its current cycle. If the BGACK* signal is inactive while the BR* signal is asserted, then the processor remains the bus master once BR* is negated. This feature reduces unnecessary interruptions in ordinary processing if the arbitration circuitry inadvertently responds to noise or if the alternate bus master decides that it doesn't need to be bus master before it has been granted bus mastership.

The 68020 Bus Grant Mechanism

The processor issues a bus grant in response to the bus request issued by the external device. BG* assertion immediately follows after internal synchronization. However, if the processor is performing a read-modify-write cycle or has already made an internal decision to perform a single bus cycle, then it must complete that operation first. During a read-modify-write cycle, the processor cannot assert the BG* signal unless the entire cycle has completed. The RMC* signal is asserted to indicate that the bus has been locked. When an internal decision has been made to execute another bus cycle, then

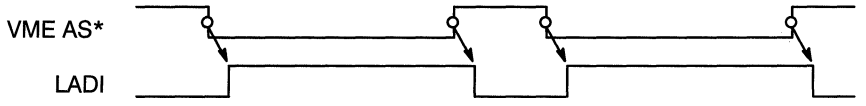


Figure 3. LADI with BUS HOLD OFF Disabled

the BG* cannot be asserted to the external device until the bus cycle has begun. The 68020 design allows the BG* signal to be routed through a daisy-chained network or, alternatively, through a priority encoded network such as an external arbiter. (The 68020 allows any kind of external arbiter as long as the arbitration sequencing is followed precisely.)

The 68020 Bus Grant Acknowledge Mechanism

Once the external device has received the BG* from the 68020, then it must wait until the local AS*, DSACK0*, DSACK1*, and BGACK* are negated before asserting its own BGACK* to the processor. The removal of the AS* signal indicates that the previous master has released the bus. The negation of the DSACK0* and DSACK1* signals indicates that the previous slave has terminated the cycle with the previous master.

The SVIC Local Bus Philosophy

The bus arbitration of the SVIC is much simpler than the 68020. This is known as the BUS HOLD-OFF feature of the SVIC.

The SVIC is intended to be the highest priority on the local bus. This implies that when a VME slave transaction occurs, then nothing will prevent the SVIC from reading or writing to local resources. Normally the SVIC starts a local cycle assuming that no other master may be in control of the local bus. This optimizes the response time of the SVIC by preventing the VME cycle being extended by local bus contention. This philosophy is not beneficial in all cases, such as where there is a local processor to consider. Some rudimentary control of the local bus shall be required from time to time by other devices.

SVIC Local Bus Arbitration Methodology

The SVIC can be prevented from starting a local cycle or a refresh of any local DRAM by using a BUS HOLD OFF function. To explain how this works, first consider the VMEbus activity. Without the bus hold function being enabled, whenever the AS* is asserted by the VME master, the SVIC will drive LADI HIGH and RAS* LOW (Row Address Strobe to DRAM) (see *Figure 3*). Then the VMEbus address is driven onto the local address bus under control of the SVIC. This happens for all VMEBus cycles whether the cycle is intended for the slave or not (the reason for this is to reduce bus latency).

When the BUS HOLD OFF feature is enabled, LADI is a 'local bus busy' signal. It indicates to the local arbitration logic that the SVIC has control of the local bus for either VME slave accesses or when the SVIC is performing DRAM refresh cycles.

As can be seen from *Figure 4*, the LADI signal goes HIGH when there is a VME AS* signal. If the cycle is not intended for the SVIC then the LADI signal is deasserted. It can be seen that LADI is also used to indicate to the local bus arbiter that a DRAM refresh cycle is taking place.

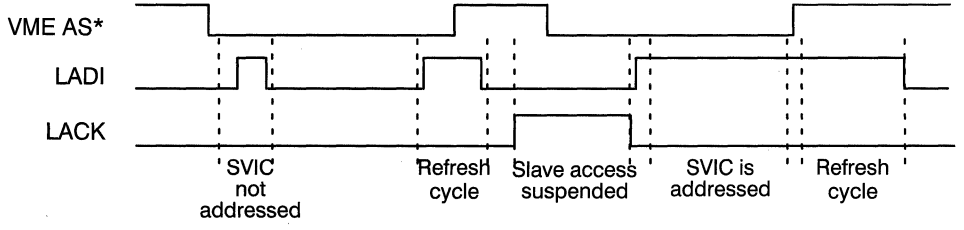


Figure 4. LADI with BUS HOLD OFF Enabled

The local bus arbiter monitors the LADI signal to determine when the SVIC does not have control of the local bus. Once the LADI signal is LOW, there is no current VME slave cycle or DRAM refresh taking place.

There are two scenarios that need to be considered for holding off the SVIC from further accesses:

1. Once the LADI signal goes LOW, the local arbiter is able to prevent the SVIC from regaining control of the local bus. As can be seen in *Figure 4*, the local bus arbiter sets the LACK signal to a '1' to 'hold off' the SVIC from regaining control of the local bus. If the LACK signal is set to a '1' by the **SECOND RISING EDGE** of the SVIC clock, then the local arbiter is guaranteed to have prevented the SVIC from getting control of the local bus.

2. The other condition for an alternate master gaining control of the local bus is when the LADI signal has been set LOW for greater than two clock cycles (i.e., when there is little VMEbus traffic). When the alternate master desires control of the local bus, the local arbiter drives the LACK signal to be a '1'. However, after **TWO RISING EDGES** of the SVIC clock signal, the arbiter must sample the LADI signal to make sure that it is still LOW. This takes account of potential metastable conditions as a result of a VME AS* being asserted to the SVIC at the same time as LACK is asserted.

When the local bus is not available to the SVIC, all VME slave transactions and will hold until the local bus is made available again. Once control of the local bus has been returned to the SVIC the refresh engine shall have priority and burst all the missed refresh cycles up to modulo 64. After this, the SVIC will respond to a pending VME slave request.

The 'bus hold off' function is enabled by a bit in the configuration bit stream. If the bit is not enabled then the SVIC cannot be prevented from performing DRAM refresh or from starting a local cycle. The function of LACK* is then simply to extend the completion of local cycles, allowing for slow local peripherals.

Design Considerations

There are certain special cases that the design engineer must consider when designing the SVIC into a VME board that can have more than one local bus master.

In the most basic applications where the SVIC is the only bus master, slave select logic is straightforward. *Figure 5* shows how this might be accomplished.

As can be seen from *Figure 5*, the three most significant address bytes are permanently enabled by connecting the LAEN inputs of the three most significant 964s to V_{CC}. This allows the VME addresses to flow directly from the VMEbus and onto the local bus. The region decoder then decodes the local addresses and the four REGION bits are fed directly into the SVIC. When a VME address appears which targets the VME board, one of the REGION bits becomes active which is then validated by the falling edge of VME AS*.

If the VME board design is such that there may be more than one bus master, then a more suitable arrangement can be seen in *Figure 6*.

Figure 6 shows that when the SVIC does not have control of the local bus, then the local addresses become isolated from the VME interface. The VME addresses are still monitored, however, by the region decoder. The output of the region decoder can then be used as the SVIC local bus request signals. These signals can be fed into the arbiter along with VME AS* to qualify the local bus request.

SVIC to 68020 Arbiter Design

The arbiter design represents a challenge to the designer. The reason for this is that the assumption of the SVIC is that it requires the highest priority and normally has control of the local bus all of the time. On the other hand, the 68020, which contains its own arbitration circuit, has the lowest priority.

The arbiter design must allow control of the local bus to default to the 68020. In addition the SVIC must not be allowed to take control of the local bus if there is any activity on the VME AS* signal unless the VME cycle is targeted towards the SVIC itself.

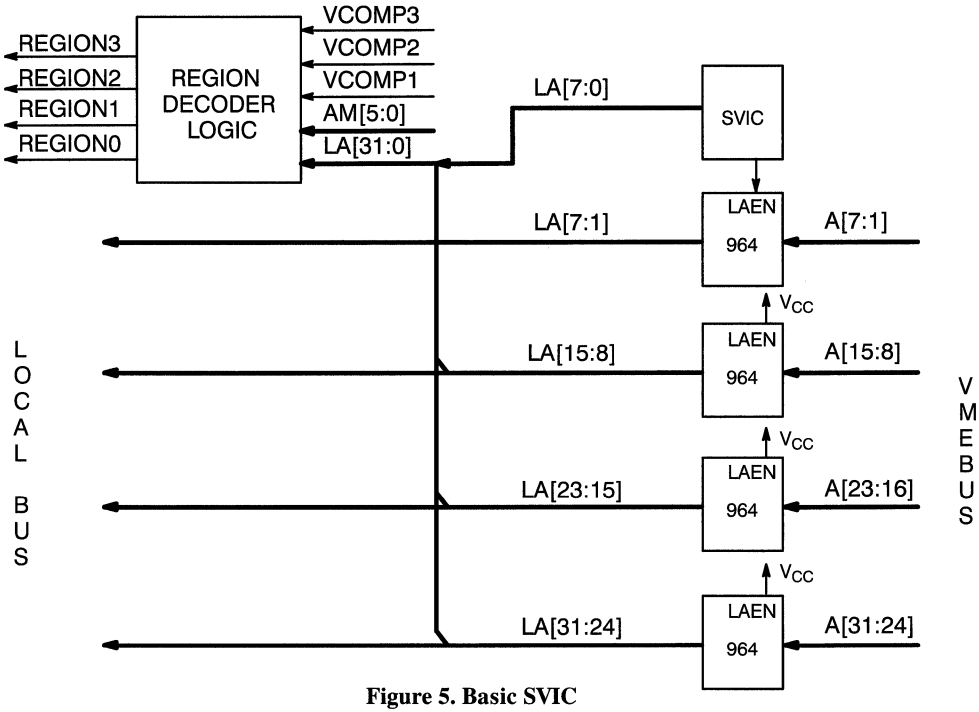


Figure 5. Basic SVIC

Coping with Metastable Events

The arbiter design is based on a state machine. The state machine is driven at the processor bus clock frequency of 20 MHz. The SVIC is driven at a higher frequency of 80 MHz. The design will require the use of RoboClock to keep the rising edges of the two clock frequencies aligned. This will greatly reduce the instances of metastability. The crystal oscillator required to drive the RoboClock will be 20 MHz (see *Figure 7*). This crystal oscillator frequency is a common frequency and is easily obtained from many crystal oscillator vendors. In addition, this frequency oscillator is easily available to military specifications.

Using RoboClock is one method of reducing potential metastable events by using clock edges that line up. There are signals, however, that are totally unpredictable as to when they arrive. One of these is the VME AS* signal. The VMEbus is totally asynchronous to both the SVIC 80-MHz clock and also the processor 20-MHz clock. To make sure that these types of signals don't make the arbiter metastable, one of two methods should be employed. One is to utilize a register that is resilient to being metastable, (i.e., it catches an event or doesn't); the other more straightforward method is to use double registering. This saves on board space and can easily be implemented in programmable logic devices. The FLASH370™ series of CPLDs supports double registering at the dedicated inputs.

Handling the DRAM Refresh

Once the SVIC has been put into holdoff mode, it has no way of indicating to the local logic that there are any pending DRAM refreshes. The SVIC can store up to 64 refresh events while it is held off, (if the number of pending refreshes exceeds 64 then the count will roll around to 0 again and 64 pending refreshes will be lost). Once the SVIC gets control of the local bus, it will initiate a burst of refresh pulses. The most straightforward way for the SVIC

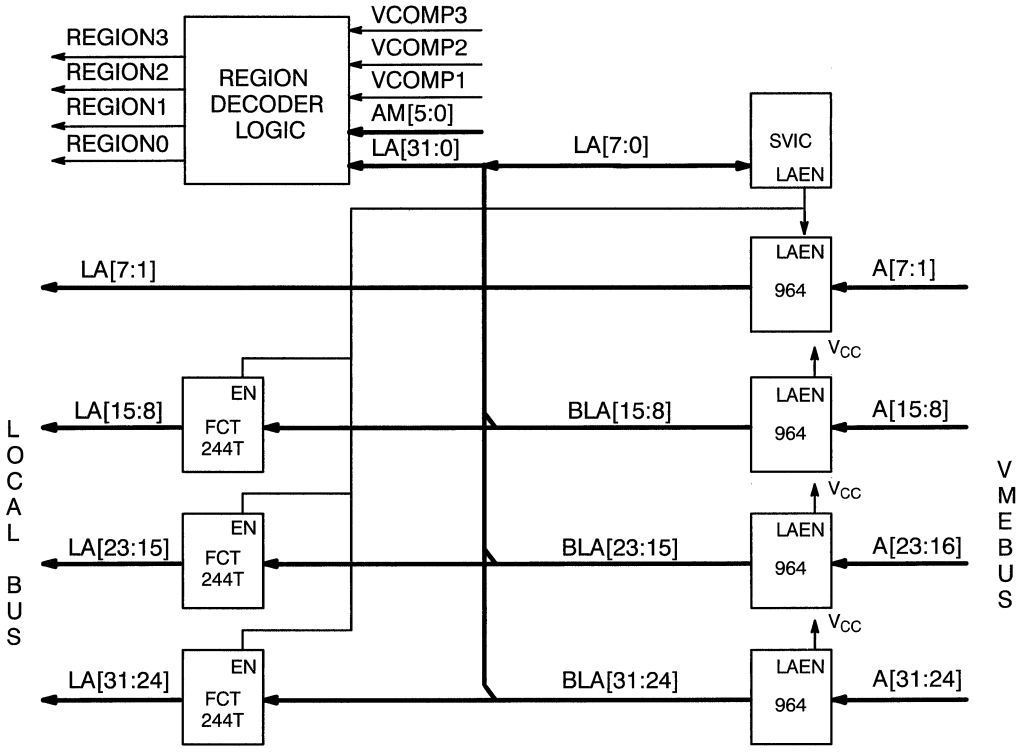


Figure 6. SVIC Implementation with More than One Bus Master

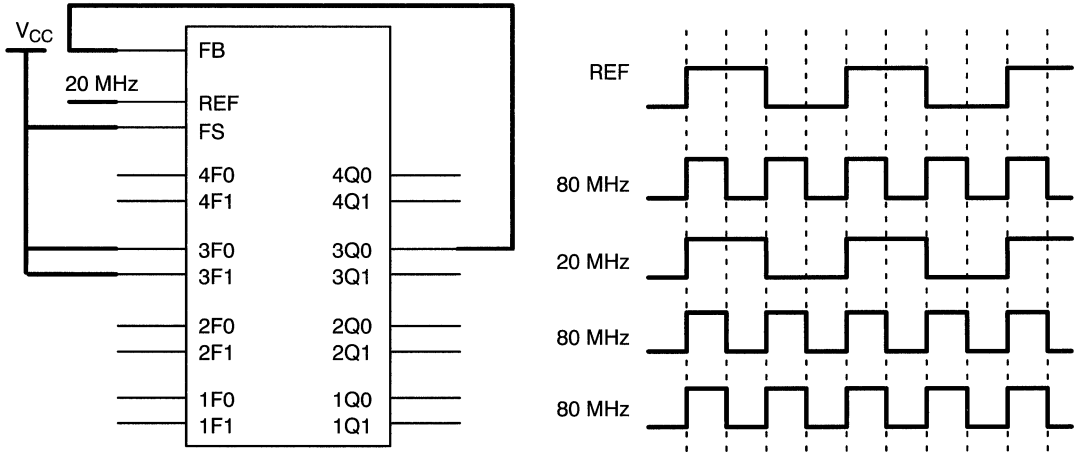


Figure 7. RoboClock Frequency

to get hold of the local bus is when a slave access takes place from the VMEbus. Once the SVIC has been granted control of the local bus, the SVIC will perform the pending DRAM refresh cycles as a higher priority. Once all of the pending refreshes have been done, then the VME master is allowed to proceed with the data transfer.

There is a case, however, when there are minimal VMEbus access requests to the SVIC. Such a situation would mean that the pending DRAM refresh cycles would build up without any chance of the SVIC of being granted control of the local bus. Hence part of the arbiter design requires the use of a counter timer that counts 125 µs. If there have been no VME cycles targeted towards the SVIC in this time (which is quite possible), then the arbiter needs to hand over control of the local bus to the SVIC and then monitor the LADI signal being inactive. Once LADI is inactive, this will indicate to the arbiter that the DRAM refresh cycles are complete and control can be taken from the SVIC. *Figure 8* shows the state diagram that is the basis of the state machine.

The source code for the design has been written in VHDL. The target device is a FLASH371−110 device. However if more registers and/or combinatorial logic is required for future upgrades or additions then the designer can migrate to a FLASH372 without having to change the real estate in the PCB that is already being used.

The flow of the state machine is shown in the timing diagram shown in *Figure 9*.

Appendix A shows the VHDL source code for the double buffering section. This was designed as hierarchical VHDL (the designer only has to instantiate the function as a single line of VHDL in the main code). This will be especially useful if a 25-MHz or 33-MHz 68020 is used. These frequencies are not a multiple of 2 so the clock domain of the 68020 and the clock domain of the SVIC (80 MHz) will be entirely asynchronous. The method of instantiating the double buffer saves time and effort.

Appendix B shows the main source code which contains the state machine design and also the DRAM refresh holdoff timeout counter.

Signal definition (*=active LOW)
BR* = bus request to the 68020
BG* = bus grant from the 68020
BGACK* = bus grant acknowledge to the 68020
SVICREQ* = SVIC requests local bus (VME cycle targets SVIC or refresh hold off times out)
SVICPROC* = SVIC granted local bus (VME cycle targets the SVIC or DRAM refresh hold off times out)
LACK* = SVIC bus grant (input to SVIC)
LADI* = SVIC bus busy (output from SVIC
dsacks = dsack0 AND dsack1
as* = 68020 address strobe

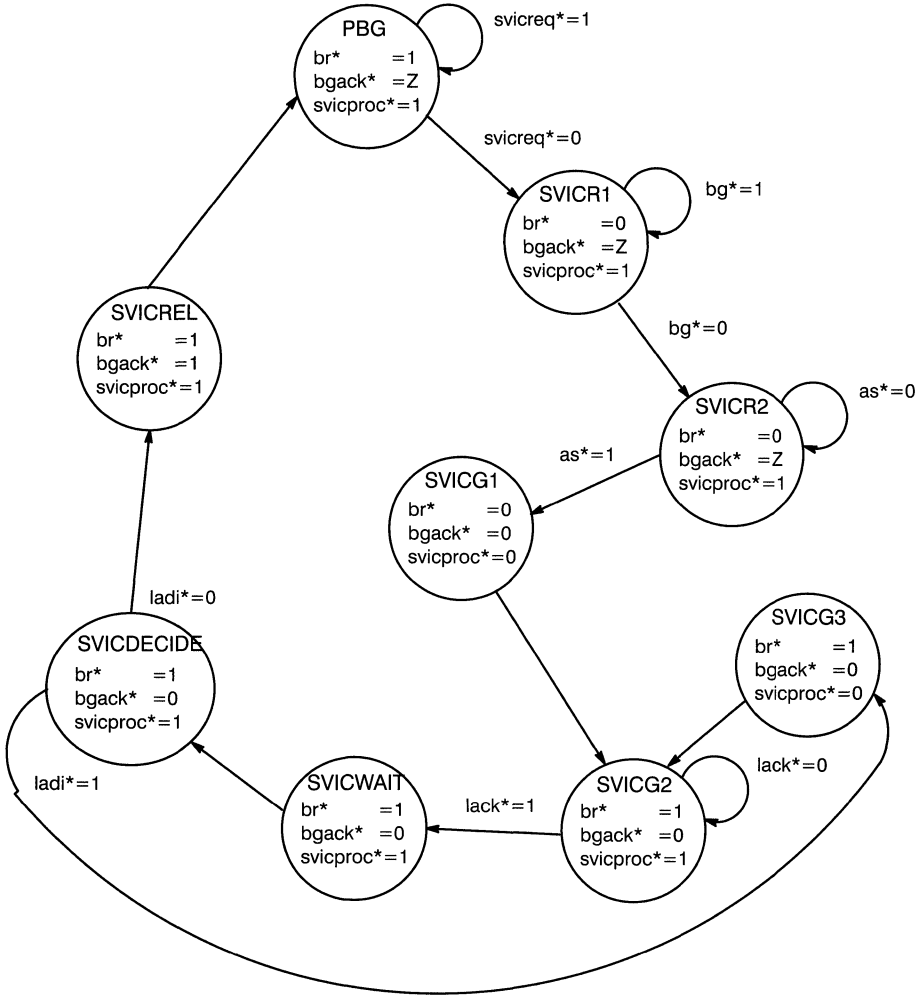


Figure 8. State Diagram of SVIC to 68020 Arbiter

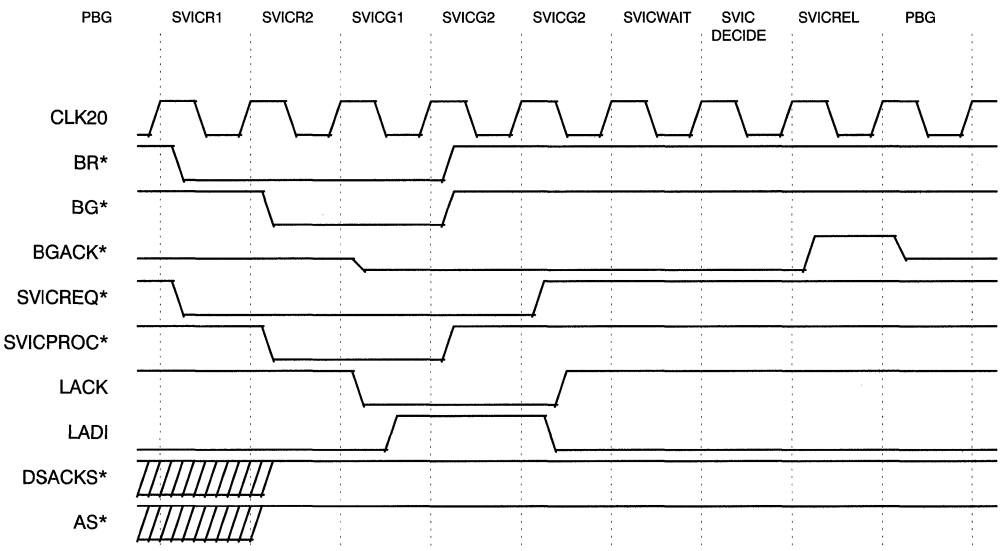


Figure 9. Timing Diagram with States

Appendix A. Source Code for Double Buffering

```
--This package description describes the double buffering technique
--for metastability hardening

PACKAGE sync_tools IS
  COMPONENT synchronise PORT(
  datain,clk:     IN BIT;
  dataout:   OUT BIT);
  END COMPONENT;
END sync_tools;

ENTITY synchronise IS PORT(

datain,clk:     IN BIT;
dataout:   OUT BIT);

END synchronise;

ARCHITECTURE archsynchronise OF synchronise IS

SIGNAL datain1:     BIT;

BEGIN

firstreg:  PROCESS (clk)

BEGIN

  IF clk'EVENT AND clk = '1' THEN
    datain1  <= datain;
  END IF;

end PROCESS firstreg;

secondreg: PROCESS (clk)

BEGIN
  IF clk'EVENT AND clk = '1' THEN
    dataout  <= datain1;
  END IF;

END PROCESS secondreg;

END archsynchronise;
```

Appendix B. Source Code for State Machine and Refresh Hold Off Timer

```
--************************************************************************
--************************************************************************
--**                                                                    **
--**     This design is an arbiter for the SVIC (960 or 961) and        **
--**     and the Motorola MC68020 (20MHz)                               **
--**                                                                    **
--************************************************************************
--************************************************************************

ENTITY arbiter IS PORT(

-- Port list for the 68020

  clk20:            IN BIT;     -- 20MHz Bus clock for the 68020
  dsack0,dsack1:    IN BIT;     -- data strobe acknowledge to 68020
  as:               IN BIT;     -- 68020 address strobe

-- Arbiter signals for the MC68020

  svicbg:           IN BIT;     -- 68020 bus grant to SVIC
  svicack:          INOUT X01Z;  -- SVIC bus grant to 68020
  svicbr:           OUT BIT; -- SVIC bus request
  outpen:           INOUT BIT; -- Output enable for bus grant

-- Arbiter signals for the SVIC

  ladi:             IN BIT;     -- latch address in (SVIC)
  lack:             BUFFER BIT;  -- local data acknowledge (SVIC)

--Port list for the 960

  reset:            IN BIT;     -- reset from reset handler
  clk80:            IN BIT;     -- SVIC 80 MHz clock
  vmeas:            IN BIT;     -- VME address strobe
  region:           IN BIT_VECTOR(2 DOWNTO 0)); -- local VME slave selects

END arbiter;

USE WORK.rtlpkg.ALL;
USE WORK.int_math.ALL;

-- The library sync_tools is a metastability hardening technique utilising
-- double buffering.

USE WORK.sync_tools.ALL;

ARCHITECTURE archarbiter OF arbiter IS

-- Definition of the states for the state machine controlling the
-- arbitration logic
```

Appendix B. Source Code for State Machine and Refresh Hold Off Timer (continued)

```
TYPE state_labels IS (pbg,svicr1,svicr2,svicg1,svicg2,svicg3,svicwait,
          svicdecide,svicrel);
SIGNAL state_bits:    state_labels;

SIGNAL svicreq:          BIT; -- SVIC request to arb logic
SIGNAL svicproc:         BIT; -- SVIC proceed from arb logic
SIGNAL bgack,bgackin:    BIT; -- Bus grant from/to controller direct
SIGNAL count256:         BIT_VECTOR(11 downto 0);   -- Refresh interval timer
SIGNAL co:               BIT; -- carry our from refresh timer
SIGNAL vmeasdel:         BIT; -- synchronised VME AS

--bgack is driven by the CPLD internally. SVICACK is tristate out and
--bgackin is monitored at pin and driven in to device.

BEGIN

--Instantiation pf bufoe to tristate bgack to 68020

  bf: bufoe PORT MAP (bgack,outpen,svicack,bgackin);

  outpen <= '1' WHEN (state_bits=svicg1) OR (state_bits=svicg2)
           OR (state_bits=svicrel) ELSE '0';

--The following process drives the 68020 arbitration

arbcntrl: PROCESS (reset,clk20)

BEGIN

IF reset = '0' THEN
  state_bits <= pbg;
  svicbr   <= '1';
  bgack    <= '1';
  svicproc <= '1';

    ELSIF (clk20'EVENT AND clk20='1') THEN

    CASE state_bits IS

-- PBG is the idle state where the processor has been granted the bus.

    WHEN pbg    =>IF svicreq = '0'
             THEN state_bits <= svicr1;
               svicbr <= '0';
               bgack  <= '1';
               svicproc<= '1';
             ELSE state_bits   <= pbg;
               svicbr <= '1';
               bgack  <= '1';
               svicproc<= '1';
           END IF;
```

Appendix B. Source Code for State Machine and Refresh Hold Off Timer (continued)

```
-- SVICREQ1 is where the SVIC requires the bus but is waiting for bus grant
-- from the 68020

     WHEN svicr1  =>IF svicbg = '0'
              THEN state_bits   <= svicr2;
                svicbr <= '0';
                bgack  <= '1';
                svicproc<= '1';
              ELSE state_bits <= svicr1;
                svicbr <= '0';
                bgack  <= '1';
                svicproc<= '1';
            END IF;

-- SVICR2 is where the SVIC has been granted the bus but the 68020 is
-- still performing a bus cycle

     WHEN svicr2  =>IF as = '1'
              THEN state_bits   <= svicg1;
                svicbr <= '0';
                bgack  <= '0';
                svicproc<= '0';
              ELSE state_bits   <= svicr2;
                svicbr <= '0';
                bgack  <= '1';
                svicproc<= '1';
            END IF;

-- SVICG1 is where the the 68020 has completed its last cycle, the SVIC
-- has been granted the bus and the arbiter asserts bus grant to the 68020
-- and the SVIC is allowed to proceed

     WHEN svicg1  =>   state_bits <= svicg2;
                svicbr <= '1';
                bgack  <= '0';
                svicproc<= '0';

-- SVICG2 waits for the SVIC to terminate a session

     WHEN svicg2  =>IF lack = '1'
              THEN state_bits   <= svicwait;
                svicbr <= '1';
                svicproc<= '1';
                bgack  <= '0';
              ELSE state_bits   <= svicg2;
                svicbr <= '1';
                bgack  <= '0';
                svicproc<= '1';
            END IF;
```

Appendix B. Source Code for State Machine and Refresh Hold Off Timer (continued)

```
-- SVICG3 allows the SVIC to proceed again in the event of a metastable
-- condition where the SVIC misses the LACK* signal going inactive

    WHEN svicg3   =>   state_bits <= svicg2;
               svicbr <= '1';
               bgack  <= '0';
               svicproc<= '1';

-- SVICWAIT is a timing period before sampling LADI

    WHEN svicwait   =>   state_bits <= svicdecide;
               svicbr <= '1';
               bgack  <= '0';
               svicproc<= '1';

-- SVICDECIDE samples LADI. If LADI is inactive then the SVIC is in
-- hold off mode. If LADI is active then the arbiter failed to hold off
-- the SVIC


    WHEN svicdecide   =>IF ladi = '0' THEN
             state_bits <= svicrel;
               svicbr <= '1';
               bgack  <= '1';
               svicproc<= '1';
          ELSIF ladi = '1' THEN
            state_bits <= svicg3;
               svicbr <= '1';
               bgack  <= '0';
               svicproc<= '0';
          END IF;

-- SVICREL hands control of the local bus back to the 68020

    WHEN svicrel =>state_bits    <= pbg;
               svicbr <= '1';
               bgack  <= '1';
               svicproc<= '1';

-- The when others clause prevents implicit memory generation and copes
-- with any illegal states

    WHEN OTHERS   =>state_bits    <= pbg;
               svicbr <= '1';
               bgack  <= '1';
               svicproc<= '1';

    END CASE;

  END IF;
END PROCESS arbcntrl;
```

Appendix B. Source Code for State Machine and Refresh Hold Off Timer (continued)

```
-- The following process defines the counter that defines 128 uS
-- before control is given to the SVIC for the purposes of DRAM
-- refresh. Making the counter wider increases the time period by a factor
-- of 2 every time, but may make logic synthesis more difficult

cnt: PROCESS (reset,clk20)
BEGIN
  IF (reset = '1') THEN        -- asynch reset
    count256 <= x"000";

  ELSIF (clk20'EVENT AND clk20 = '1') THEN
    IF (state_bits = svicrel) THEN
      count256 <= x"000";
    ELSIF ((state_bits = pbg) AND (co =  '0')) THEN
      count256 <= inc_bv(count256);
    END IF;
  END IF;
END PROCESS cnt;

-- The co signal is used to inhibit the counter when it gets to the
-- terminal count

co<= '1' WHEN (count256 = x"9FF") ELSE '0';

--The following section defines the SVIC arbiter

--The VME AS* is asynchronous to the 80MHz clk so needs to be synchronised

sync1:   synchronise PORT MAP (vmeas,clk80,vmeasdel);

svicreq <= '0' WHEN (((region /= "000") AND (vmeasdel = '0'))
    OR count256 = x"9FF") ELSE '1';

lack<= '0' WHEN (svicproc = '0')
    OR ((lack = '0') AND (ladi = '1')) else '1';

END archarbiter;
```

RACEway Products from Cypress Semiconductor

Cypress Semiconductor now offers RACEway interconnect system developers an independent source for Interlink modules, crossbar chips, and RACEway on-ramp components compliant with the RACEway Interlink standard.

The RACEway Interlink standard is published and maintained by VITA (VMEbus and Futurebus International Trade Association). The VITA standards organization (VSO) has ratified the RACEway Interlink Specification which defines the data link protocol and the physical interface definition for the high-performance extension to the VMEbus standard.

RACEway Crossbar CY7C965

- 160 Mbyte per second per path Block Transfer Rates
- Six bidirectional ports
- Non-blocking architecture
- 361-pin CBGA package
- Implements Open Bus Standard (VITA 5–1994)
- Building Block for Scaleable Networks
- Preemptable prioritized transactions
- Adaptive Routing support

The CY7C965 RACEway Crossbar implements in one device the RACEway open standard for cross point interconnect (VITA 5–1994). The RACEway standard allows multiple processor systems to communicate using a crossbar technology that supports very high aggregate data transfer rates. Applications for the RACEway Crossbar include high-performance multiprocessing systems, and distributed processing systems. The RACEway Crossbar can be used in backplane-based applications or as switch elements on single boards.

The RACEway Crossbar can be connected in many different system configurations. In its simplest configuration, the Crossbar is used to interconnect six RACEway nodes using a single crossbar. Higher complexity systems may require the implementation of a large fabric of interconnected Crossbars.

RACEway Interlink Modules

- CYM9652 provides a 4 slot RACEway fabric
- CYM9653 provides an 8 slot RACEway fabric
- CYM9654 provides a 12 slot RACEway fabric
- CYM9655 provides a 16 slot RACEway fabric
- CYM9651 provides a single slot connection for expansion purposes

Cypress's RACEway Interlink Modules bring embedded supercomputing performance to real-time VME-based systems. As a backward-compatible upgrade, RACEway Interlink transforms the topology of an existing VMEbus chassis from a single transaction bus to a scaleable real-time fabric capable of over 1 Gbyte/sec of aggregate bandwidth. Interlink modules add interboard bandwidth to VME-based systems by providing multiple, concurrent, high-speed communication paths between VME boards interfaced to the RACEway Interlink stan-

dard. In addition to increased bandwidth, RACEway Interlink offers low latency and priority control, essential to real-time applications.

Mechanically, the RACEway Interlink Modules mount on the backplane of a VME chassis similar to industry-standard VSB backplane modules. Electrically, these modules are connected to the VME slots through the P2 chassis backplane connector. RACEway Interlink Modules implement the RACEway interconnect fabric, using the Cypress CY7C965 RACEway Crossbar device and appropriate clock and interface circuitry.

RACEway On-ramp: PitCREW

- Used to interface between FIFOs and the RACEway protocol.
- Drives/receives a RACEway port directly.
- Is programmed from the RACEway.
- Has a DMA engine capable of moving data between a local FIFO and the RACEway.
- Moves data at 160 MByte/sec peak and 140 MByte/sec sustained throughput.
- Able to write DMA status to RACEway for polling or mailbox interrupt.
- 144-pin, 8K gate Cypress CY7C387A FPGA.

PitCREW is an I/O data port for RACEway. It defines a simple FIFO interface local data port which is slave to its RACEway port. The PitCREW has an internal DMA engine which moves blocks of data between RACEway nodes and its FIFO port. This DMA engine is set in motion by commands received over the RACEway port. Data move instructions can be issued directly to the PitCREW RACEway port, or caused to be fetched by the PitCREW in a linked list fashion from memory associated with a RACEway node. All the logic required to control data movement between FIFOs and the RACEway resides in this device.

pASIC is a trademark of Quicklogic.

RACEway On-ramp: PitCREWjr

- Used to interface between FIFOs and the RACEway protocol.
- Drives/receives a RACEway port directly.
- Simple master control, automatic slave response.
- Moves data at 160 MByte/sec peak and 140 MByte/sec sustained throughput.
- Implemented in a Cypress CY7C384A, a 2K gate 100-pin FPGA.

PitCREWjr is a simple full-duplex on-ramp to the RACEway fabric. The device has a standard RACEway port and FIFO port. The controller functions either as a RACEway slave, moving data between RACEway and local FIFOs or as a RACEway master, again moving data between RACEway and local FIFOs. It connects to and drives a RACEway interlink port directly providing all required handshaking and control signaling. PitCREWjr's local FIFO port consists of a 32-bit bidirectional data bus and control signals for moving data between PitCREWjr and industry-standard FIFO components. The PitCREWjr has no programmable internal registers. Internal PitCREWjr state machines assemble and disassemble the route, address, and data long words embedded in the RACEway protocol. RACEway mastering is accomplished by controlling a single input signal.

Mercury Computer RIC–RINO Component Files

Data files are available for the RIC–RINO RACEway on-ramp chipset developed by Mercury Computer. This chipset is superseded by the PitCREW RACEway On-ramp for new designs. The two necessary items are a PROM file for the data path EPLD definition and a .CHP file for a CY7C384A pASIC which replaces the FPGA specified by Mercury. These files are provided on request.

Interfacing to RACEway: PitCREW

PitCREW is intended for engineers who are designing an I/O circuit for use as an "on-ramp" to the RACEway switching fabric. This document illustrates a simple but complete FIFO interface to RACEway. This design can be used as described or as the starting point for custom RACEway interface development. This application note describes:

- The design specification for the PitCREW I/O Controller.
- Electrical information for designing a FIFO-based I/O circuit with the PitCREW Controller.

Reference Documents

Use this application note in conjunction with the latest Cypress data books and data sheets and related published standards documents. These resources are as follows:

- Cypress CY7C387P and pASIC380™ Family data sheets
- *Cypress Programmable Logic Data Book 1994/1995*. For more information on using pASIC380 Family devices, see the *Cypress Applications Handbook*
- *RACEway Interlink – Data Link and Physical Layers, VITA 5–1994*, available from the VITA Standards Organization (VSO)
- Cypress CY7C4245 4K x 18 Synchronous FIFO data sheet
- *The VMEbus Specification, VITA 1–1994*
- Cypress CY74FCT162H501T data sheet
- Cypress CY7B9910 Low Skew Clock Buffer data sheet
- Front Panel Data Port Electrical and Physical Layers VITA 17 – 199x

RACEway On-Ramp System Overview

In general, this on-ramp is an I/O data port for a RACEway fabric. It defines a simple FIFO interface which is a slave to its RACEway port. Transactions cannot be initiated via the FIFO interface. Instead, the on-ramp has a DMA engine that moves blocks of data between RACEway nodes and its I/O port. This DMA engine is set in motion by commands received over the RACEway fabric. Data move instructions can be issued directly to the RACEway port, or placed in the memory of another RACEway node in the form of a linked list. This on-ramp should be considered a slave board whose function is controlled from a program executing on one or more RACEway nodes.

The on-ramp is comprised of the PitCREW I/O controller, an input FIFO, an output FIFO, and a bi-directional transceiver with synchronizing latches. *Figure 1* outlines the major components of the on-ramp.

The PitCREW Controller is implemented in a Cypress CY7C387P FPGA. All the logic required to control data movement between the FIFOs and the RACEway fabric resides in this device. PitCREW drives the RACEway fabric directly and implements the features described in the remaining sections. The architecture of a sample interface using PitCREW is shown in *Figure 2.*

Each FIFO is implemented with a pair of CY7C4245 4K x 18 Synchronous FIFOs. The trans-

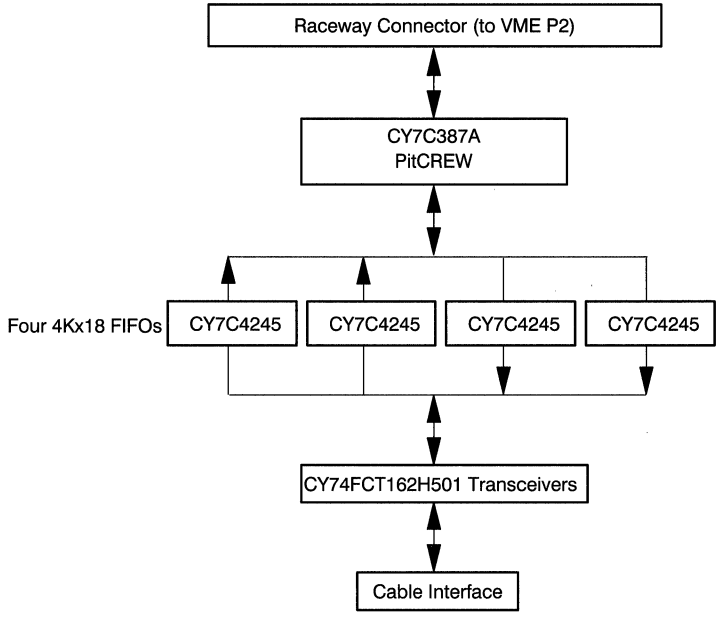


Figure 1. Components of a Sample I/O Interface

ceiver function is handled by a pair of CY74FCT162H501 registered transceivers. The FPGA and the FIFOs are available in 0.5-mm lead pitch TQFP packages (144-lead and 64-lead, respectively). The transceivers are available in a 56-lead SOIC pack.

Features

The on-ramp allows for autonomous DMA transfer through asynchronous data FIFOs. Transfers can be from RACEway to FIFO, FIFO to RACEway, or both (full duplex on the user side of the FIFOs, half duplex over RACEway). Features of the on-ramp circuit include:

- A DMA engine capable of routing a data stream between an external device and any node in the RACEway fabric.
- 160 MB/sec peak and 140 MB/sec sustained throughput.
- A 40-MHz, 32-bit cable interface, compatible with the Front Panel Data Port (FPDP) Standard.
- Flow control, synchronization signals, and user programmable bits available over the cable interface.
- Ability to write status to a RACEway memory location (for local polling) or to a mailbox location (to cause an interrupt).
- Optimal use of the crossbar network bandwidth by automatically buffering blocks of data for burst crossbar transfers at 160 MB/sec.
- Ability to act as a RACEway slave so a RACEway node anywhere on the network fabric can set up, control, or test the operation of the board.

Operation

The PitCREW Controller provides DMA operations on the RACEway Interlink, interfacing either an input FIFO, an output FIFO, or both to the

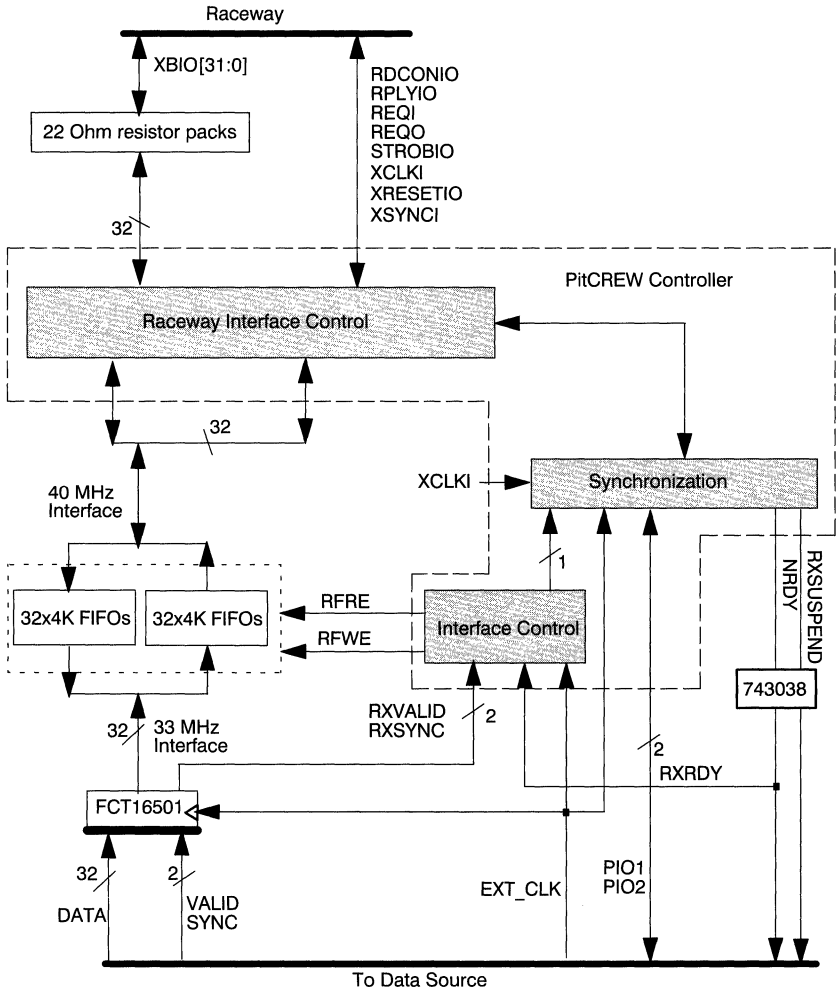


Figure 2. Architecture of a Sample Input Interface Using PitCREW

RACEway. Control signals are provided for the user side of the FIFOs, which can run asynchronously to the RACEway.

PitCREW always functions as a transaction master on the RACEway when it is moving data, and bursts at the full 160 megabyte per second rate. It can be operated in linked-list fashion, fetching a new command packet from the RACEway at the completion of the current one. Each command packet consists of a word count, the new contents of the Control Register, the data route and address, and the next command packet route and address.

The linked list of command packets is built in memory accessible over the RACEway fabric. The DMA engine is started by a RACEway master writing a load and go operation specifying the route and address of the first packet directly into the PitCREW Controller. The Controller then fetches and

executes from the linked list until a command packet is fetched with the GO bit reset. The linked-list structure is shown in *Figure 3*.

A simpler control alternative is to write the "Data Address," "Data Route," and "Word Count" registers each time a DMA transfer is desired. Writing the "Word Count" register will cause a DMA transfer to start.

For reads from the cable interface, as shown in *Figure 4*, the controller counts valid words as they are placed into the input FIFO. When the counter

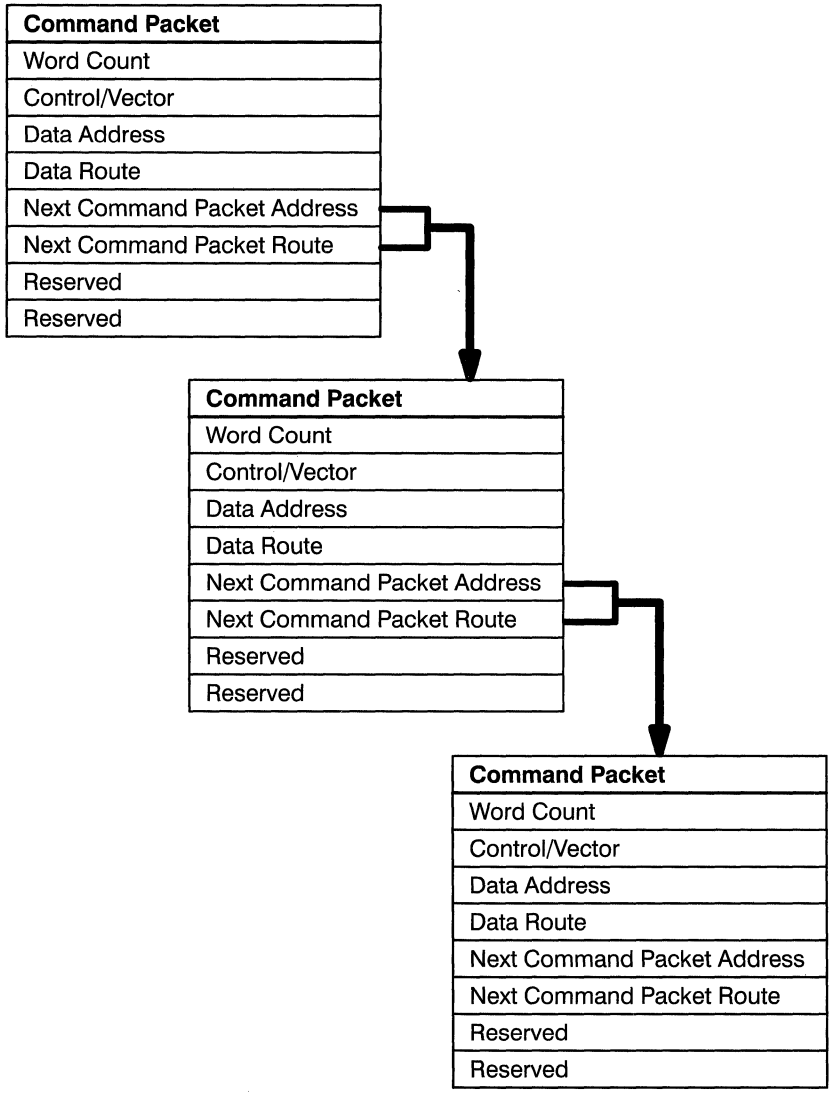


Figure 3. Linked List Operation

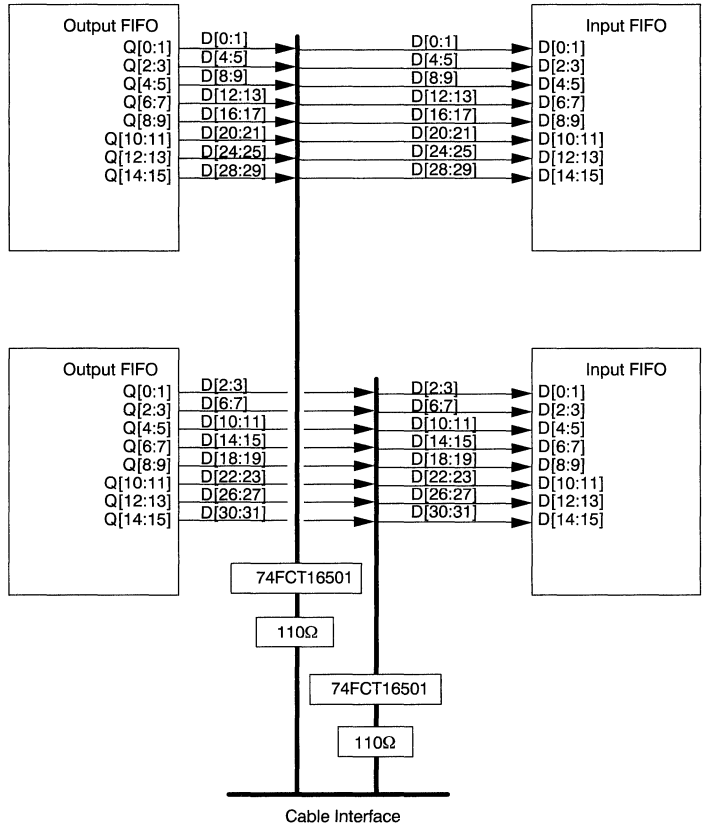


Figure 4. Example—Connecting the FIFOs to a Cable Interface

reaches 2K bytes, data is read from the input FIFO by PitCREW and written to the RACEway as a burst operation. The controller accepts a "data valid" input ($\overline{\text{RXVALID}}$) for qualifying input FIFO loading, as well as a sync input pin ($\overline{\text{RXSYNC}}$) allowing for an external event to start the acquisition.

For writes to the cable interface, a "suspend" signal ($\overline{\text{TXSUSPEND}}$) is provided for throttling the read operation of the cable side of the output FIFOs. When the Programmable Almost Full pin ($\overline{\text{TFPAF}}$) on the output FIFO indicates to PitCREW that there is room in the FIFO, a burst operation transfers data from the RACEway to the output FIFO to fill it up. PitCREW provides the output FIFO interface signals, as well as the ability of placing a sync marker (SET_SYNC) in the output FIFO for framing the data.

Two user-programmable I/O bits (PIO[2:1]), are available for data tagging or other application-specific purposes. These bits may be individually programmed via the PitCREW Control Register to be either inputs or outputs. These bits may be used to tag command packets as they are executed. For example, headers and data may be assigned different tags.

It is possible to perform a Status Write operation in which the DMA status is written to a memory location specified by the PitCREW data route and address registers. It is accomplished by controlling bit 25 of the word count field of a linked-list command

packet. If bit 25 is zero, the linked list entry is a "write status" command instead of a DMA move command. This feature is provided for semaphore operations, and is a mechanism for signaling DMA complete to a RACEway process.

Also provided is the ability to read and write the internal registers of the PitCREW, to write to the output FIFO, and to read from the input FIFO as a slave interface. These functions are all provided mainly for diagnostic purposes.

Connecting the FIFO Interface

Figure 5 describes the connections between the CY7C4245 FIFOs and the PitCREW Controller. For information on the CY7C4245 FIFO and its signals, see the Cypress CY7C4245 4K x 18 Synchronous FIFO data sheet.

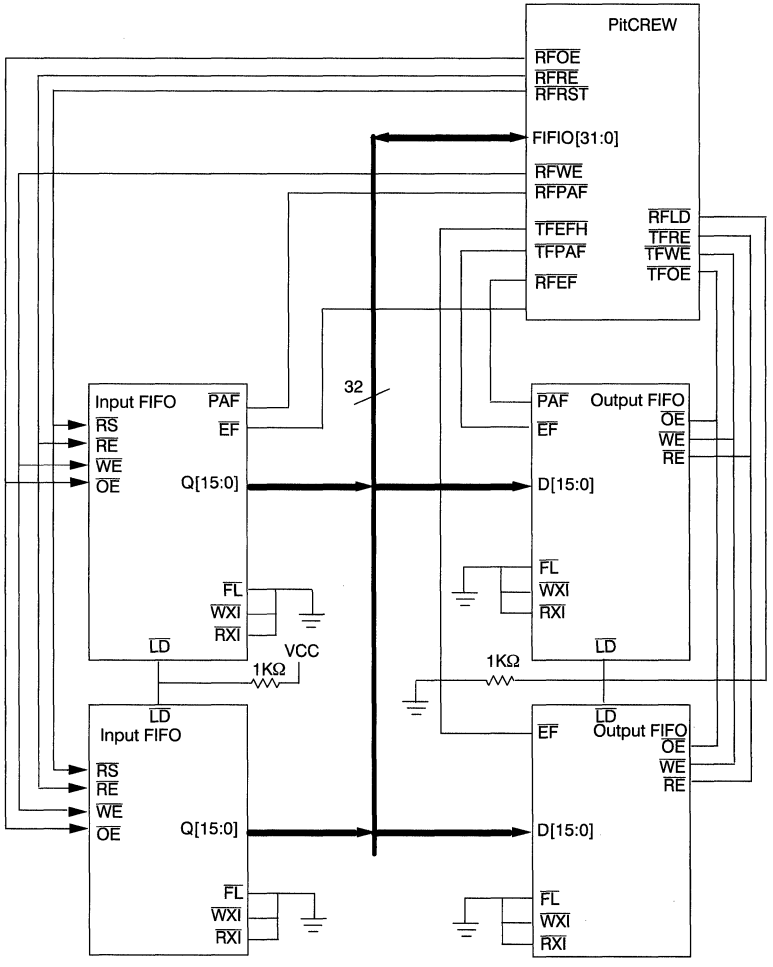


Figure 5. Connecting the FIFOs to the PitCREW Controller

Registers

Register Address Map

The following two tables display the addresses for the PitCREW registers for writing and reading separately. Most of the registers are 32 bits wide but mapped into 64-bit address space, since this is the granularity of a single cycle on the RACEway (there is no address bit 2). A few of the registers are true 64-bit registers as discussed below.

In the Register Write Address Map, entries designated NA (not available) are not writable locations. To perform a write operation to any of the register locations, with the exception of address 0x10, either a 64-bit or a 32-bit write should be specified with the data located in bits 63 through 32.

Address 0x10 is a special address to allow a 64-bit *load and go* operation. If a 64-bit write is specified to address 0x10, the Command Address register is loaded from bits 63 through 32 and the Command Route register is loaded from bits 31 through 0. After the load-and-go write, the Controller will fetch the command packet pointed to by the route and address in the load and go, and execute that packet (this assumes that the GO bit is set in the Command Address register data).

A second method of initiating a transfer is to perform a 32-bit write of the Command Route register data at address 0x10 (with route data located on bits 31 through 0) followed by a write to address 0x18 of the Command Address register (with the GO bit set).

DMA transfers can also be initiated by directly writing the Word Count register after loading appropriate values in Data Route and Data Address registers. This method circumvents use of the linked-list convention of the PitCREW.

It is possible to write directly from a RACEway master to the output FIFO via address 0x28. Users are warned that the last long word of any RACEway write to this address will NOT be written to the output FIFO.

It is also possible to read from address 0x28 to move data from the input FIFO to the RACEway.

Register Write Address Map

Address[5:3]	Bits 63.....32	Bits 31.....0
000	NA	NA
001	Control	NA
010	Command Address	Command Route
011	Command Address	NA
100	Word Count	NA
101	TX FIFO	NA
110	Data Route	NA
111	Data Address	NA

Register Read Address Map

Address[5:3]	Bits 63.....32	Bits 31.....0
000	Status	Status
001	Control	Control
010	Command Route	Command Route
011	Command Address	Command Address
100	Word Count	Word Count
101	RX FIFO entry n	RX FIFO entry n+1
110	Data Route	Data Route
111	Data Address	Data Address

Reading from all addresses except 0x28 will return the same data replicated on the upper and lower 32-bit words. Reading from address 0x28 will return the next two consecutive input FIFO entries (64 bits). This is primarily for diagnostic purposes.

Command Route Register

31	28	25	22	19	16	13	10	7	4 3	2 1	0
Route 0	Route 1	Route 2	Route 3	Route 4	Route 5	Route 6	Route 7	Route 8	Broadcast Accept. Code	Routing Priority	0

The Command Route register is used by the PitCREW to retrieve the next command packet in the linked list. The format of this register is the standard format from the RACEway interlink standard VITA 5–1994. Bit 0 must always be reset to zero in this register.

Command Address Register

31 28	27 3	2	1	0
Width/Alignment	Address	Go	Read	Locked

The Command Address register is used to specify the address of the next command packet in the linked list. The Width/Alignment, Address, and Locked fields are the same format as specified in the RACEway interlink standard. When a command packet is fetched (or written into the registers) with the Go bit set, the next command packet will be fetched at the completion of the current command packet. The last command packet fetched in a linked list should have the Go bit reset.

The Read bit must always be set to a one to specify reading a command packet. Also, the Locked bit should always be set to a one, specifying that the fetch is not locked.

Data Route Register

31	28	25	22	19	16	13	10	7	4 3	2 1	0
Route 0	Route 1	Route 2	Route 3	Route 4	Route 5	Route 6	Route 7	Route 8	Broadcast Accept. Code	Routing Priority	Broadcast/ Single

The Data Route register contains the route for the data packet to be transferred. The format is the same as specified in the RACEway interlink standard.

Data Address Register

31 28	27 3	2	1	0
Width/Alignment	Address	Reserved	Transmit	Locked

The Data Address register contains the address for the data packet to be transferred. The format is the same format as specified in the RACEway interlink standard. The Transmit bit specifies the direction of the transfer: when it is set, data is read from the RACEway and written to the output FIFO. When it is reset, data is read from the input FIFO and written to RACEway.

Status Register

Bit	Function	Active HIGH	Query Control	Description
31:29	Reserved			
28	Read Error	Yes	S/LL	Error reading command or data
27:26	PIO[2:1]		S/LL	User controlled input bits
25:24	Reserved			
23	Output FIFO Greater Than Zero	Yes	S/LL	Data present in output FIFO ($\overline{\text{TFEFL}}$ pin)
22	Ready In	Yes	S/LL	Cable interface ready (RXRDY pin)
21	Valid Packet	Yes	S	Command Packet has valid format. "Not Valid" cleared by a correct packet
20	Overflow	Yes	S/LL	Input FIFO overflow
19	Input Suspended	Yes	S/LL	Input FIFO almost full ($\overline{\text{RFPAF}}$)
18	Input FIFO Greater Than Zero	Yes	S/LL	Data present in input FIFO (dynamic)
17	Reserved			Reserved – read as zero
16	Reserved			Reserved – read as zero
15:4	Board Type		S/LL	0x010=PitCREW
3:0	Board Rev		S/LL	Board Revision

The **Query Control** column displays whether the bit can be queried under either slave (**S**) control, linked-list control (**LL**), or both (**S/LL**). The following paragraphs discuss the different fields in the status register.

The **Read Error** bit is set when an error is detected during a RACEway transfer. It is cleared either by hardware reset or by writing the control register.

The **PIO[2:1]** field is used to read the state of the PIO pins when these pins are operated in input mode.

The **Output FIFO Greater Than Zero** bit is connected directly to the $\overline{\text{TFEFL}}$ pin of PitCREW.

The **Ready In** bit is connected directly to the RXRDY input pin of PitCREW.

The **Valid Packet** bit gets set when a valid packet is fetched. A valid packet is defined as containing a valid packet field in the Word Count register.

The **Overflow** bit is set when a input FIFO overflow occurs. This bit can be cleared by a hardware reset or a software reset of the Input FIFO in the Control register.

The **Input Suspended** bit is essentially the PitCREW $\overline{\text{RFPAF}}$ pin synchronized to the EXT_CLK.

The **Input FIFO Greater Than Zero** bit is an internally generated input FIFO not empty signal.

Control Register

Bit	Function	Active HIGH	Load Control	Description
31:30	PIO Enable[1:0]	Yes	S/LL	User bits direction: 0 = In, 1 = Out
29:28	PIO[2:1] Data		S/LL	User controlled data output
27	Reserved			
26	Output Reset	Yes	S/LL	Self-pulsed output FIFO reset
25	PIO Cntl Enable	Yes	S/LL	Mask for controlling user outputs
24	Input Reset	Yes	S/LL	Self-pulsed input FIFO reset
23	Sync Wait	Yes	S/LL	Self-pulsed Wait For Sync trigger for the input FIFO logic
22	Ready Out	Yes	S	Enable transfers
21	Stop DMA	Yes	S	Stop operation in progress—current packet data may be corrupted
20	Reserved			Reserved for future use
19	Sync Out	Yes	S/LL	Self-pulsed signal setting Send Sync with next output FIFO data
18	Reserved			Reserved for future use
17	$\overline{\text{RSVD2}}$ Out	No	S/LL	For use as a general purpose output pin.
16	Reserved			Reserved for future use
15:0	Rupt Vector		S/LL	Interrupt control

The **Load Control** column displays whether the bit can be loaded under either slave (**S**) control, linked-list control (**LL**), or both (**S/LL**). The following paragraphs discuss the different fields in the Control Register.

The **PIO Enable[1:0]** field provides individual direction control over the two PitCREW programmable I/O pins. When a PIO Enable[1:0] bit is defined as output, the value driven out of that PIO pin is specified in the **PIO[2:1] Data** field. In order to change either the PIO[2:1] Enable or PIO[2:1] Data fields the **PIO Cntl Enable** bit must be set. Writes and link-list loads to the control register with the PIO Cntl Enable bit reset will not affect the PIO[2:1] Enable and PIO[2:1] Data fields.

The **Output Reset** bit performs a reset of the output FIFOs and the associated logic internal to the PitCREW Controller. To perform a reset, a one is written to the Output Reset bit. It is not necessary to follow this with a write of zero—the Output Reset bit is self-pulsed. This reset will be followed by an output FIFO load cycle to load the watermark value of the programmable flags.

The **Input Reset** bit performs a reset of the input FIFOs and the associated logic internal to the PitCREW Controller. Like the Output Reset bit, the Input Reset bit is self-pulsed. Also, a programmable flag load cycle is not performed for the input FIFO since the PitCREW Controller does not have access to the data input of the input FIFO devices.

The **Sync Wait** bit is a self-pulsed bit that puts the input FIFO interface logic in the armed state. Input FIFO write enable ($\overline{\text{RFWE}}$) will not go active until a sync pulse is input on the $\overline{\text{RXSYNC}}$ pin (synchronous to EXT_CLK).

The **Ready Out** bit is used to set and reset the NRDY output pin. The pin will be inverted from the register bit. It is intended that the cable be driven through an inverting open-collector buffer, and then brought back into the RXRDY pin. Note that this bit can only be modified by performing a slave write—not via linked-list load.

The **Stop DMA** bit will stop a transfer in process. The transfer can then be continued or aborted. The integrity of the data packets may be corrupted if used in conjunction with the Output Reset or Input Reset bits in this register (an abort of the command packet).

The **Sync Out** bit allows for a sync marker to be written into the output FIFO to tag the beginning of a data frame. This sync marker moves through the FIFO with the data.

The **$\overline{\text{RSVD2}}$ Out** bit is inverted and connected to the $\overline{\text{RSVD2}}$ pin of the PitCREW. It is for use as a programmable output pin.

Word Count Register

Bit	Word Count Reg	Function
31:27	Reserved	Reserved
26	Bit Bucket	Discard output data
25	Write Type	1= Data Write, 0 = Status Write
24:21	Valid Packet Field	Must be equal to binary '0010'
20	Reserved	Reserved
19:0	Word Count [19:0]	Number of 8–byte words to write (Up to 8 Mbytes)

The Word Count register can be loaded linked-list style, or it may be written or read directly via the RACEway. The following paragraphs describe the Word Count Register fields in greater detail.

The **Bit Bucket** bit, when set, will cause the PitCREW output logic to discard the output data. Data will be read from the RACEway but not written into the output FIFOs. This is useful for diagnostic purposes.

The **Write Type** bit is normally set to a one to perform data writes, however by resetting this bit, a status write will be performed. During this write, bits 31 through 16 of the Status register are concatenated with bits 15 through 0 of the Control register (the Rupt field) and written to the route and address specified in the Data Route and Data Address registers. This is useful for end of transfer notification, and is also a method of performing RACEway interrupts.

The **Valid Packet Field** is used to detect runaway linked lists. The Word Count register is the first register loaded in a linked list. If the Valid Packet Field fetched in the command packet is not equal to binary '0010', then the data transfer is never started and the Valid Packet bit in the Status register is cleared indicating an error.

The **Word Count** field is loaded with the number of 8-byte (64-bit) words to be transferred. In the output direction, the PitCREW Controller checks the $\overline{\text{TFPAF}}$ signal to see if there are 576 empty slots (2304 bytes) available in the output FIFOs. An output transfer cycle will be initiated when the number of available slots is at least 576 (which is 512 data slots plus 64 sync marker slots corresponding to 2304 bytes). The size of the transfer will be equal to the lessor of 2K bytes or the value programmed into the Word Count register. For input cycles, the Controller actually counts the number of entries in the input FIFOs by counting the number of EXT_CLK rising edges with $\overline{\text{RXVALID}}$ active. The pins RXRDY and $\overline{\text{RXSYNC}}$ are also used to define valid input data entries. An input transfer cycle is initiated if the number of entries in the input FIFO is equal to the value in the Word Count Register or 2 Kbytes, whichever is lower.

Signals

The 108 signal pins of the PitCREW controller can be divided into six main groups:

- RACEway interface signals
- Output FIFO interface signals
- Output FIFO control signals
- Input FIFO interface signals
- Input FIFO control signals
- Cable interface signals

The RACEway interface signals provide a port to the RACEway fabric with full 160 megabytes per second capability. These signals are synchronous to the RACEway clock. The RACEway clock frequency is 40 MHz. *Table 1* lists these signals.

The output and input FIFO interface groups provide strobes to reset, read, and write both sets of FIFOs. The input and output FIFOs share the 32-bit FIFO data bus (FIFIO[31:0]) and the asynchronous external clock (EXT_CLK). Pins are provided to interface to the input and output FIFO status flags and to set the initial value of the programmable status flag in the output direction. Setting this value is not possible in the input direction since there is no data bus connection to the inputs of the input FIFOs. Instead, PitCREW counts valid entries as data is clocked into the input FIFOs.

The output FIFO control group provides signals to control data being read from the output FIFOs (TXRDY and $\overline{\text{TXSUSPEND}}$), to provide indication that valid data has been read from the FIFOs (TXVALID), and to generate a start of frame marker to be placed into the output FIFOs ($\overline{\text{TXSYNC}}$).

The input FIFO control group provides signals to control data being placed into the input FIFOs ($\overline{\text{RXVALID}}$, RXRDY), two indicators (opposite polarities) that show the input FIFOs are almost full (RXSUSPEND, $\overline{\text{RXSUSPEND}}$), and a start of frame indicator which allows for the acquisition of data frames based upon an external event ($\overline{\text{RXSYNC}}$). Also provided are two programmable data bits used for data tagging under software control (PIO[2:1]). An overflow pin is provided for input FIFO error indication ($\overline{\text{OVFLOW}}$).

Table 1. PitCREW RACEway Signals

Signal	I/O	Source	Function
RDCONIO	I/O	PitCREW or RACEway	Indicates to the crossbar to three-state the data bus so read data can be driven. It also indicates when a read error has occurred.
RPLYIO	I/O	PitCREW or RACEway	Reply gives the RACEway crossbar permission to send the address or data over the data bus.
REQI	I	RACEway	Request In indicates that the RACEway crossbar is requesting control of the data bus.
REQO	O	PitCREW	Request Out is asserted by the master to request access to the crossbar data bus.
STROBIO	I/O	PitCREW or RACEway	Strobe indicates that address or data is being sent on the data bus. Strobe is sent by the master node after asserting REQO.
XBIO[31:0]	I/O	PitCREW or RACEway	Crossbar Address/Data. These lines must each have 22Ω series termination.
XCLKI	I	RACEway	Crossbar Clock provides the RACEway timing.
XRESETIO	I	RACEway	Reset input from the RACEway connecting port.
XSYNCI	I	RACEway	Crossbar Sync provides control phase information to the crossbar.

Output FIFO Interface

The PitCREW Controller provides interfaces to the cable side of both the input FIFO and the output FIFO, also referred to as the user side of the FIFOs. The following section discusses the output FIFO interface, both the user and RACEway sides.

The output FIFOs can be reset by either the assertion of the $\overline{\text{XRESETIO}}$ pin, or by writing to bit 26 in the PitCREW Control register. Either of these events will cause the $\overline{\text{TFRST}}$ signal to go LOW. An output FIFO reset is always followed by a programmable flag load cycle where the flag data is presented on the FIFIO data bus and the $\overline{\text{TFLD}}$ and $\overline{\text{TFWE}}$ signals asserted. The flag data consists of 0x240, which corresponds to 2304 bytes. This byte size is determined by allocating 2 Kbytes (512 32-bit entries) for data and 256 bytes (64 entries) for sync markers. Note that this places an upper limit of 64 sync markers for every 256 data words which must be adhered to.

The programmable flag load cycle requires that bits 11 through 0 of the FIFO data bus (FIFIO[11:0]) must be connected to bits 11 through 0 of the FIFO that is used to send $\overline{\text{TFPAF}}$ to the Controller (only one of the $\overline{\text{TFPAF}}$ output FIFO flags needs to be connected to the Controller). Also, $\overline{\text{TFLD}}$ must be connected to both output FIFOs in order to prevent an extra write from being registered in the FIFO which does not supply $\overline{\text{TFPAF}}$ (the $\overline{\text{TFWE}}$ pin goes active during a programmable flag load cycle).

The generation of the output FIFO read signal, $\overline{\text{TFRE}}$, is based upon the $\overline{\text{TXSUSPEND}}$, TXRDY, $\overline{\text{TFEFL}}$, and $\overline{\text{TFEFH}}$ input signals. If all of these signals are high then $\overline{\text{TFRE}}$ goes active. The $\overline{\text{TXVALID}}$ output will go LOW in response to the read, if the sync input signal INV_SYNC is not active ($\overline{\text{TXVALID}}$ only goes active for valid data items—not sync markers). $\overline{\text{TXSUSPEND}}$ must be returned to the Controller synchronous to EXT_CLK. In the case of the cable interface, an external synchronizing flip-flop is recommended between the cable signal $\overline{\text{SUSPEND}}$ and the $\overline{\text{TXSUSPEND}}$ pin on the Controller. $\overline{\text{TFRE}}$ is guaranteed to go inactive within four EXT_CLKs from the rising edge of $\overline{\text{TXSUSPEND}}$.

The output FIFO programmable almost full flag $\overline{\text{TFPAF}}$ is used by the PitCREW Controller to burst data over the RACEway. A RACEway burst read cycle is initiated when there are at least 576 (2304 ÷ 4) empty locations in the output FIFO. The size of the burst is equal to the lesser of 2 Kbytes or the value programmed into the PitCREW Word Count register.

It is not required to use the PitCREW output FIFO interface. The data output side of the output FIFO may be clocked asynchronously to the EXT_CLK as long as the $\overline{\text{TFPAF}}$, and both of the FIFO empty flags $\overline{\text{TFEFL}}$ and $\overline{\text{TFEFH}}$, are connected to the Controller.

A sync marker may be placed in the output FIFO using the SET_SYNC, INV_SYNC, and $\overline{\text{TXSYNC}}$ pins. By setting the Sync Out bit in the Control register, a sync marker will be driven out on the SET_SYNC pin. It is intended that this be connected to one of the unused data inputs on the output FIFOs (assuming that the FIFOs are organized 18 bits wide). The output from the data bit should be connected to the INV_SYNC input pin. The $\overline{\text{TXSYNC}}$ output pin is simply an inversion of the INV_SYNC pin going active when the sync marker is read out of the FIFO. Also, the $\overline{\text{TXVALID}}$ output is gated by the INV_SYNC pin and will not go active for the sync marker FIFO read. *Table 2* summarizes the output FIFO interface signals, and *Table 3* summarizes the output FIFO control signals.

Table 2. PitCREW Output FIFO Interface Signals

Signal	I/O	Source	Function
EXT_CLK	I	External	External clock synchronous to FIFO interface. Is common to both TX and RX FIFO logic.
FIFIO[31:0]	I/O	PitCREW	Data lines to the output FIFOs and from the input FIFOs
$\overline{\text{TFLD}}$	O	PitCREW	Output FIFO load for programmable flags.
$\overline{\text{TFOE}}$	O	PitCREW	Output FIFO output enable
$\overline{\text{TFRE}}$	O	PitCREW	Output FIFO read enable
$\overline{\text{TFRST}}$	O	PitCREW	Output FIFO reset
$\overline{\text{TFWE}}$	O	PitCREW	Output FIFO write enable
$\overline{\text{TFEFL}}$,$\overline{\text{TFEFH}}$	I	FIFO	Output FIFO empty flags from both FIFOs for PitCREW
$\overline{\text{TFEF}}$	I	FIFO	Output FIFO empty flag for PitCREW status register reads—connect to either FIFO flag
$\overline{\text{TFPAF}}$	I	FIFO	Output FIFO programmable almost full flag

Table 3. PitCREW Output FIFO Control Signals

Signal	I/O	Source	Function
PIO[2:1]	I/O	PitCREW	Programmable data bits used for software handshaking.
NRDY	O	PitCREW	Generates Ready out to cable interface. This HIGH-active signal should go through an inverting open-collector buffer to drive the cable $\overline{\text{NRDY}}$ signal.
TXRDY	I	External	Should be connected to the output of the NRDY open-collector buffer (to NRDYN). When active indicates that data can be read out of the output FIFO on the next EXT_CLK.
$\overline{\text{TXSUSPEND}}$	I	External	May be asserted to suspend reading out of the output FIFO (to throttle output data).
SET_SYNC	O	PitCREW	Sync (top of frame) marker output to connect to output FIFO data input to tag start of data frame in FIFO.
INV_SYNC	I	FIFO	Sync marker input from output FIFO (output of FIFO input signal SET_SYNC).
$\overline{\text{TXSYNC}}$	O	PitCREW	Indicates the start of a data frame when asserted. Is inverted INV_SYNC for use in driving the cable interface SYNCN signal.
$\overline{\text{TXVALID}}$	O	PitCREW	Indicates valid data has been read out of the FIFOs. May be used to drive the cable interface VALIDN signal.

Input FIFO Interface

The input FIFO interface may be reset either by the assertion of the $\overline{\text{XRESETIO}}$ pin, or by writing to bit 24 in the Control Register. Either of these will cause the $\overline{\text{RFRST}}$ signal to go LOW. Loading of the programmable flags is not performed in the input FIFO interface because the Controller does not have access to the input FIFO input data path.

The generation of the input FIFO write enable $\overline{\text{RFWE}}$ is based upon the RXRDY and $\overline{\text{RXVALID}}$

pins, as well as the sync logic utilizing the $\overline{\text{RXSYNC}}$ pin. Ignoring the sync logic for the moment, if the $\overline{\text{RXVALID}}$ pin is LOW and the RXRDY pin is HIGH, the $\overline{\text{RFWE}}$ signal will go active (LOW). Note that the path from either of these two input signals to the $\overline{\text{RFWE}}$ output is purely combinatorial. $\overline{\text{RXVALID}}$ is intended to be used to gate off individual writes into the input FIFO and RXRDY is intended to be tied to the output of the open-collector buffer driven by the NRDY output (stating that the cable is ready).

The above example assumes that the sync logic is disabled, that is the Sync Wait bit in the PitCREW Control register has not been set. If the Sync Wait bit is set, the logic generating $\overline{\text{RFWE}}$ will wait until a single EXT_CLK pulse on the $\overline{\text{RXSYNC}}$ is detected. The first write will occur on the clock following the assertion of the $\overline{\text{RXSYNC}}$ pin, if RXRDY and $\overline{\text{RXVALID}}$ are also active as described above.

The PitCREW Controller does not use the input FIFO flags to determine when to initiate a RACEway write transfer. Instead, it counts valid input FIFO entries as defined in the above criteria and initiates a RACEway transfer upon detecting the lessor of 2 Kbytes or the value programmed into the PitCREW Word Count register. Data will be read from the FIFO and written to the RACEway until the word count reaches zero, the FIFO empties, a 2-Kbyte boundary is reached, or the RACEway Request In signal is raised (indicating a "Kill" condition). Any of these conditions will cause the Request Out signal to be deasserted.

When the word count reaches zero, the next command packet is fetched and operation continues if the GO bit of the PitCREW Command Address register is set. Using two 4K X 16 FIFOs yields 16 Kbytes of buffering which, at 120 MB/sec, corresponds to 136 microseconds. *Table 4* summarizes the input FIFO interface signals and *Table 5* summarizes the input FIFO control signals. Two other signals, $\overline{\text{RSVD2}}$ and TXDIR, are described in *Table 6*.

Table 4. PitCREW Input FIFO Interface Signals

Signal	I/O	In From/ Out To	Function
EXT_CLK	I	External	External clock synchronous to FIFO interface. Is common to both TX and RX FIFO logic.
FIFIO[31:0]	I/O	FIFO	Data lines to the FIFO.
RFLD	O	PitCREW	Input FIFO load for programmable flags. This pin is a static high-level (no programmable load function performed).
RFOE	O	PitCREW	Input FIFO output enable.
$\overline{\text{RFRE}}$	O	PitCREW	Input FIFO read enable.
$\overline{\text{RSTRF}}$	O	PitCREW	Input FIFO reset.
$\overline{\text{RFWE}}$	O	PitCREW	Input FIFO write enable.
$\overline{\text{RFPAF}}$	I	FIFO	Programmable Almost Full Flag from FIFO.
$\overline{\text{RFEF}}$	I	FIFO	Input FIFO empty flag.

Table 5. PitCREW Input FIFO Control Signals

Signal	I/O	In From/ Out To	Function
$\overline{\text{OVFLOW}}$	O	PitCREW	Indicates a input FIFO overflow has occurred.
PIO[2:1]	I/O	PitCREW	Programmable data bits used for software handshaking.
PIOEN[2:1]	O	PitCREW	Indicate (when LOW) that the PIO[2:1] pins are enabled.
RXRDY	I	External	Should be connected to the output of the NRDY open-collector buffer (to $\overline{\text{NRDY}}$ on the cable interface). When active (HIGH) allows data to be written into the input FIFO.
RXSUSPEND	O	FIFO	Asserted HIGH when the FIFO is almost full (127 words from full).
$\overline{\text{RXSUSPEND}}$	O	FIFO	Asserted LOW when the FIFO is almost full (127 words from full).
$\overline{\text{RXSYNC}}$	I	External	Indicates the start of a data frame when asserted.
$\overline{\text{RXVALID}}$	I	External	Indicates valid data is available to write to the FIFOs when low. Is used to dynamically qualify each data word written into the input FIFOs.

Table 6. Miscellaneous Control Signals

Signal	I/O	In From/ Out To	Function
$\overline{\text{RSVD2}}$	O	PitCREW	Set/reset from bit 17 of the Control Register. This bit is inverted from the value programmed into the Control Register.
TXDIR	O	PitCREW	Used to indicate the direction of data transfer on the cable interface.

Cable Interface Signal Description

The PitCREW Controller can be connected to a bi-directional cable interface compatible with FPDP (see Reference Documents section for related standard). This interface consists of a 32-bit data bus, two user-defined data bits for data tagging, a free-running clock, and five control signals. The cable interface supports multiple destinations, but the required arbitration is not described in this note. The following paragraphs describe how cable interface signals are related to PitCREW control signals.

The source of the data (transmitter) drives the signal $\overline{\text{DIR}}$ LOW to indicate the direction is from the cable interface to the input FIFOs. This signal is included on the PitCREW Controller. All sources and destinations drive the open-collector signal NRDY, which indicates that the cable is ready. This signal is also included on the PitCREW Controller. Sources of data are required to read NRDY in hardware to ascertain that the interface is in the ready state. This is performed via the RXRDY pin on the PitCREW Controller. The free-running clock, STROB, is sourced by the source of the data bus and drives the EXT_CLK pin of PitCREW.

A data synchronization signal, $\overline{\text{SYNC}}$, is provided to frame data at the input FIFO. The input FIFO will wait until a single pulse of $\overline{\text{SYNC}}$ is detected, and then start to acquire data on the next assertion of $\overline{\text{VALID}}$. The $\overline{\text{VALID}}$ signal is used to indicate that valid data is available to be input on a particular rising edge of STROB. $\overline{\text{SYNC}}$ and $\overline{\text{VALID}}$ are synchronized to STROB (EXT_CLK) and connected to PitCREW pins $\overline{\text{RXSYNC}}$ and $\overline{\text{RXVALID}}$ respectively.

A suspend signal is provided to inform the data transmitter to stop sending data. The receiver asserts the $\overline{\text{SUSPEND}}$ signal when its buffer is almost full. The $\overline{\text{RXSUSPEND}}$ output of PitCREW provides this signaling.

Pins

Table 7 identifies the CY7C387P pinout for the PitCREW Controller.

Table 7. Cypress CY7C387P Pinout

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	FIFIO17	37	XBIO26	73	RXSUSPEND	109	NU/GND
2	XBIO17	38	NU/GND	74	$\overline{\text{TXSUSPEND}}$	110	NU/GND
3	FIFIO14	39	XBIO27	75	TXRDY	111	NU/GND
4	XBIO12	40	XBIO25	76	$\overline{\text{RFRST}}$	112	NU/GND
5	XBIO14	41	XBIO30	77	NU/GND	113	$\overline{\text{RXSYNC}}$
6	FIFIO12	42	VCC	78	$\overline{\text{TFEFH}}$	114	VCC
7	VCC	43	FIFIO30	79	VCC	115	$\overline{\text{RFWE}}$
8	XBIO16	44	XBIO31	80	XRPLYIO	116	NRDY
9	FIFIO16	45	FIFIO31	81	NU/GND	117	XBIO5
10	FIFIO11	46	XBIO28	82	$\overline{\text{TFRST}}$	118	XBIO3
11	XBIO11	47	XBIO29	83	XSTROBIO	119	XBIO6
12	FIFIO18	48	FIFIO28	84	XREQ_O	120	XBIO9
13	XBIO18	49	FIFIO29	85	NU/GND	121	FIFIO6
14	FIFIO19	50	GND	86	$\overline{\text{TFPAF}}$	122	GND
15	GND	51	PIO2	87	GND	123	FIFIO7
16	XBIO19	52	PIOEN2	88	TXDIR	124	XBIO7
17	$\overline{\text{XRESETIO}}$	53	FIFIO0	89	XREQ_I	125	XBIO8
18	EXT_CLK	54	GND	90	XCLKI	126	GND
19	VCC	55	FIFIO5	91	VCC	127	FIFIO9
20	RXRDY	56	FIFIO2	92	$\overline{\text{RFPAF}}$	128	PIOEN1
21	$\overline{\text{RXVALID}}$	57	FIFIO4	93	$\overline{\text{TFEFL}}$	129	PIO1
22	VCC	58	VCC	94	VCC	130	VCC
23	XBIO20	59	XBIO2	95	XRDCONIO	131	FIFIO8
24	FIFIO21	60	FIFIO1	96	XSYNCI	132	FIFIO3
25	FIFIO22	61	XBIO4	97	$\overline{\text{OVFLOW}}$	133	RSVD2
26	FIFIO23	62	XBIO0	98	$\overline{\text{RFRE}}$	134	XBIO10
27	FIFIO20	63	$\overline{\text{TFLD}}$	99	$\overline{\text{RFOE}}$	135	NU/GND
28	XBIO22	64	XBIO1	100	NU/GND	136	FIFIO10
29	FIFIO24	65	SET_SYNC	101	NU/GND	137	FIFIO13
30	GND	66	GND	102	GND	138	GND
31	XBIO21	67	$\overline{\text{TFWE}}$	103	NU/GND	139	XBIO15
32	XBIO23	68	INV_SYNC	104	NU/GND	140	XBIO13
33	FIFIO27	69	$\overline{\text{TXVALID}}$	105	NU/GND	141	XTDI
34	FIFIO26	70	$\overline{\text{TXSYNC}}$	106	NU/GND	142	XTDO
35	FIFIO25	71	$\overline{\text{TFRE}}$	107	$\overline{\text{TFOE}}$	143	$\overline{\text{RFLD}}$
36	XBIO24	72	$\overline{\text{RXSUSPEND}}$	108	NU/GND	144	FIFIO15

PitCREW Programming Considerations

Direct access to the PitCREW DMA channel is gained by writing the Word Count register. Writing this register initiates a DMA transfer. Values previously written to the Data Route and Data Address registers are used for RACEway direction and header information. The Word Count register specifies the transfer length.

The PitCREW Controller can also operate in linked-list fashion, fetching a new command packet at the completion of the current one, as shown in *Figure 3*. Each command packet consists of a word count, the new contents of the Control register, the data route and address, and the next command packet route and address.

The linked list of command packets is built in memory and a load-and-go operation specifying the route and address of the first packet is written to the 64-bit location at address 0x10 (the combined Command Route and Address register) to prime the operation. The PitCREW then executes each element in the linked list until a command packet is fetched with the GO bit reset. The GO bit in the Command Address register instructs the Controller to fetch the next command packet at the destination specified the Command Address and Command Route registers. The last command packet in the linked list should have the GO bit reset.

The Read bit in the Command Address register should always be set to a one, to specify reading the command packet from RACEway memory. All Lock bits should always be one specifying that operations are not locked.

The Data Route and Address registers specify the location in RACEway memory where the data packet will be stored (input operation) or fetched (output operation). The output bit in the Data Address register specifies the direction of operation to be either input (reset to zero) or output (set to a one).

Two user programmable I/O bits, PIO[2:1] are provided for data tagging or other application specific purposes. These bits may be individually programmed via the Control register to be either inputs or outputs, and the programming may be accomplished through direct writes to the Control register or under linked-list control. Assigning the data values under linked-list control allows for the tagging of the command packets as they are executed. For example, the first packet may be assigned a value that tags it as a header and subsequent packets may be tagged as data.

In order to program the PIO bits, the PIO Control Enable bit must be set. Writes to the Control register with the PIO Control Enable bit reset will not affect either the direction (PIO Enable[1:0]) or the value (PIOdata[1:0]) fields. It is intended that the PIO bits be connected to the 33rd and 34th bits of the FIFOs. In this way they may be used by custom hardware to distinguish data packets. They cannot, however, be transferred to memory, which has a 32-bit data organization.

The Stop DMA bit may be set to pause or abort a DMA operation in progress. When set by a slave write operation to the Control register, the current DMA operation will be in a paused state. Resetting the Stop DMA bit at a later time will resume the operation. Setting Output Reset and Input Reset bits while in the paused state will cause an abort to take place. Note that the integrity of the data packets may be violated after aborting an operation.

The PitCREW Controller has the ability to write the status of the Controller to the RACEway memory location specified in the Data Route and Address registers. This is accomplished under linked-list control as a separate command packet and is the basic mechanism used for notification of end of packet.

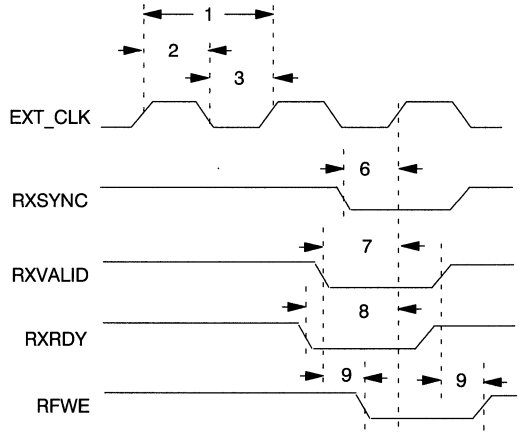
When a command packet is fetched that has bit 25 in the Word Count register reset, a Status Write operation will be performed. In this operation, bits 31 to 16 of the Status register will be concatenated with bits 15 to 0 of the Control register (the Rupt vector) and written to RACEway memory. Note that the Transmit bit in the Data Address register must be reset to zero specifying a write to RACEway memory as the direction.

Status Write operations may be interspersed with actual data transfers in the linked list as a method of sending end of packet status to a controlling process. The location specified in the Status Write operation may be polled by the controlling process, or the location may be specified to be a mailbox interrupt location for a given process on the RACEway. In this way, an end of packet interrupt may be generated to the requesting process via the linked list. If only the Rupt field is desired, the Data Width and Alignment bits in the Data Address register may be used.

Timings

Input Timing

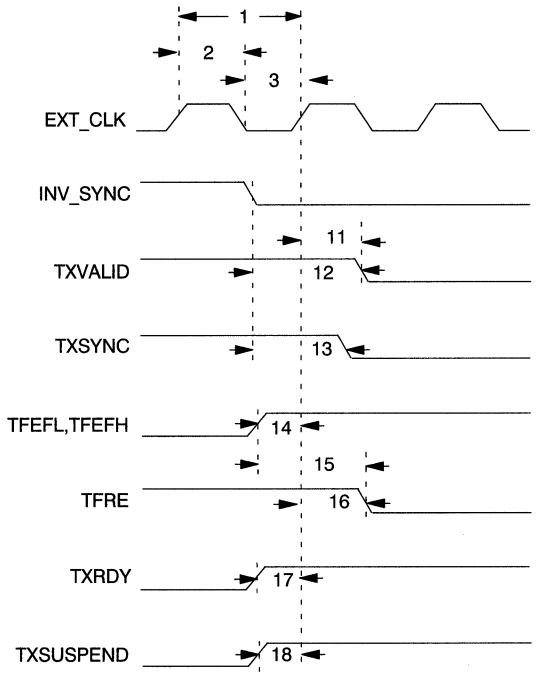
The following timing diagram describes the worst-case timing parameters for the input interface.



Symbol	Parameter	Min.	Max.	Note
1	EXT_CLK clock period	25		
2	EXT_CLK high width	10		
3	EXT_CLK low width	10		
6	$\overline{\text{RXSYNC}}$ set-up time to EXT_CLK	8		
7	$\overline{\text{RXVALID}}$ set-up time to EXT_CLK	19		A
8	RXRDY set-up time to EXT_CLK	21		B
9	$\overline{\text{RXVALID}}$ to $\overline{\text{RFWE}}$ delay		13	A

Output Timing

The following timing diagram describes the worst-case timing parameters for the output interface.



Symbol	Parameter	−1 Min.	−1 Max.	−2 Min.	−2 Max.	Notes
1	EXT_CLK clock period	30		25		
2	EXT_CLK high width	13		10		
3	EXT_CLK low width	13		10		
11	EXT_CLK to $\overline{\text{TXVALID}}$ delay		10		8	
12	INV_SYNC to $\overline{\text{TXVALID}}$ delay		13		11	
13	INV_SYNC to $\overline{\text{TXSYNC}}$		10		8	
14	$\overline{\text{TFEFH}}$, $\overline{\text{TFEFL}}$ set-up time to EXT_CLK	10		8		
15	$\overline{\text{TFEFH}}$, $\overline{\text{TFEFL}}$ to $\overline{\text{TFRE}}$ delay		12		10	
16	EXT_CLK to $\overline{\text{TFRE}}$ delay		10		8	
17	TXRDY set-up time to EXT_CLK	10		8		
18	$\overline{\text{TXSUSPEND}}$ set-up time to EXT_CLK	10		8		A, C

Notes

A. $\overline{\text{RXVALID}}$ to $\overline{\text{RFWE}}$ is a combinatorial path used to dynamically mask writes to the input FIFO. The delay for the path is specified above in symbol 9. Symbol number 7, $\overline{\text{RXVALID}}$ set-up time to EXT_CLK, includes a FIFO set-up time of 6 ns.

B. RXRDY is intended to be a static signal displaying the ready status of the cable interface.

C. $\overline{\text{TXSUSPEND}}$ must meet the set-up time specified in symbol 18. An external synchronizing flip-flop is recommended for the cable interface. The $\overline{\text{TFRE}}$ signal is guaranteed to transition to the inactive state within four EXT_CLK periods from the rising edge of $\overline{\text{TXSUSPEND}}$.

Design Considerations

This section describes the minimum requirements for the design of input and output interfaces.

Basic Input Interface

The simplest input interface requires only EXT_CLK and data signals. However, the basic interface must meet the following conditions:

- EXT_CLK is a free-running clock.
- The data stream must be continuous; the relative starting point within the data stream is arbitrary.
- The aggregate data rate must not exceed the overall sustainable bandwidth.
- Unused control lines must be set to the appropriate state. In a basic synchronous interface, tie both $\overline{\text{RXVALID}}$ and $\overline{\text{RXSYNC}}$ LOW. With $\overline{\text{RXVALID}}$ tied LOW, data is valid on all cycles. With $\overline{\text{RXSYNC}}$ tied LOW, data transfers are not synchronized.

Figure 6 illustrates a basic interface.

Input Data Qualification with $\overline{\text{RXVALID}}$

The $\overline{\text{RXVALID}}$ signal should be asserted LOW when valid data is to be input. *Figure 7* illustrates the use of $\overline{\text{RXVALID}}$. Note that the user should monitor the RXSUSPEND signal, which is a doubly-synchronized version of the RFPAF pin, and stop writing into the input FIFO when RXSUSPEND goes active.

Input Data Qualification with $\overline{\text{RXSYNC}}$ and $\overline{\text{RXVALID}}$

Figure 8 illustrates buffered interface using $\overline{\text{RXSYNC}}$ and $\overline{\text{RXVALID}}$. If using the sync wait mode, data will not be written to the FIFO until the cycle after the first SYNC pulse is received.

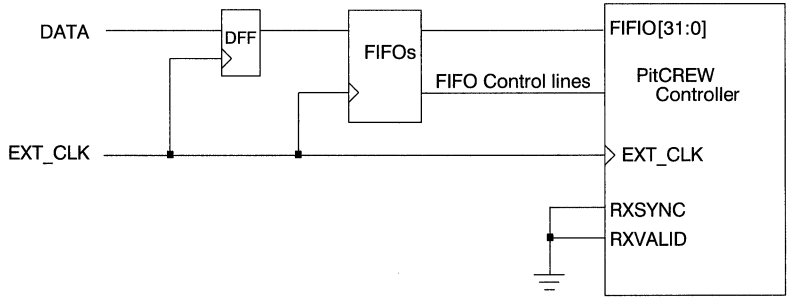


Figure 6. Basic Input Interface

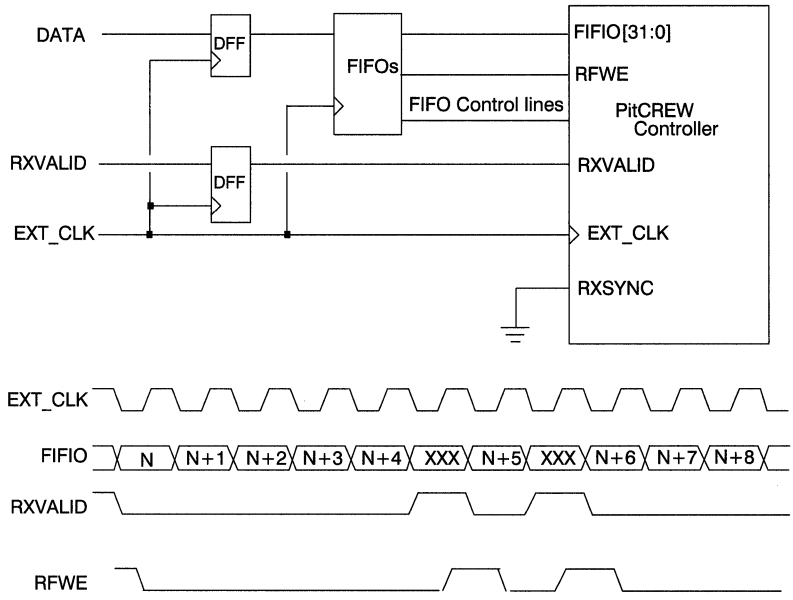


Figure 7. Input Data Qualification with $\overline{\text{RXVALID}}$

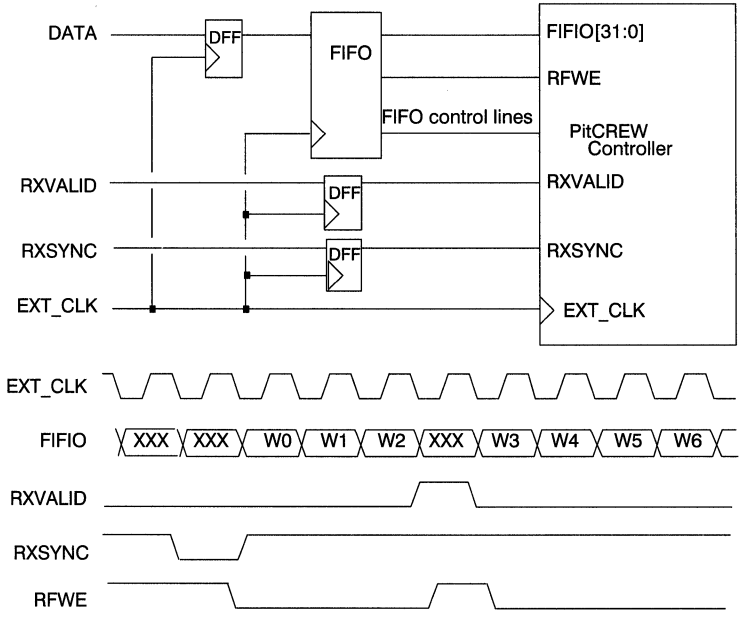


Figure 8. Input Data Qualification With $\overline{\text{RXSYNC}}$ and $\overline{\text{RXVALID}}$

Basic Output Interface

The simplest output interface requires only EXT_CLK and data signals. However, the basic interface must meet the following conditions:

- EXT_CLK is a free-running clock.
- The data stream must be continuous; the relative starting point within the data stream is arbitrary.
- The aggregate data rate must not exceed the overall sustainable bandwidth.
- Unused control lines must be set to the appropriate state.

In a basic synchronous interface, tie both $\overline{\text{TXSUSPEND}}$ and INV_SYNC LOW. With $\overline{\text{TXSUSPEND}}$ tied LOW, data will be continuously read out of the output FIFO. With INV_SYNC tied LOW, a start of frame sync is not generated. *Figure 9* illustrates a basic interface.

Controlling Data Transmission with $\overline{\text{TXSUSPEND}}$ and $\overline{\text{TXSYNC}}$

Figure 10 illustrates an output interface with full controls. Reading out of the output FIFO can be controlled dynamically with the $\overline{\text{TXSUSPEND}}$ pin. The $\overline{\text{TXVALID}}$ pin will be active when valid data is output from the FIFO. With the Sync Out bit set in the Control register, a sync marker will be written into the output FIFO. When the sync marker is later read out, the $\overline{\text{TXSYNC}}$ pin will go active and the $\overline{\text{TXVALID}}$ pin will be invalid.

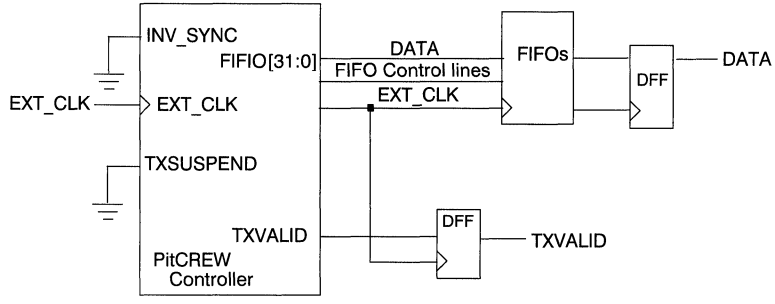


Figure 9. Basic Output Interface

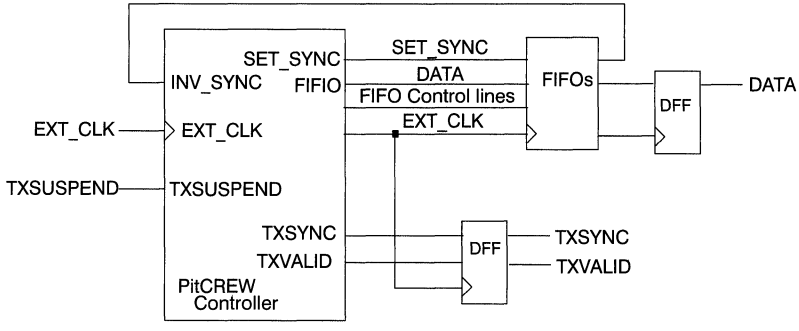


Figure 10. Output Interface with $\overline{\text{TXSUSPEND}}$ and $\overline{\text{TXSYNC}}$

Miscellaneous Design Information

Clocking

The crossbar clock, XCLKI, runs directly from the connector to the Cypress CY7B9910 Low Skew Clock Buffer chip. The clock outputs of this device are used by all on-board components that operate on the 40 MHz RACEway clock frequency. These outputs should be series terminated through 22-ohm resistors. All loads on XCLKI should be connected in series with the daughtercard or P2 connector as the source, and all loads should be within two inches of each other. The ideal configuration is illustrated in *Figure 11.*

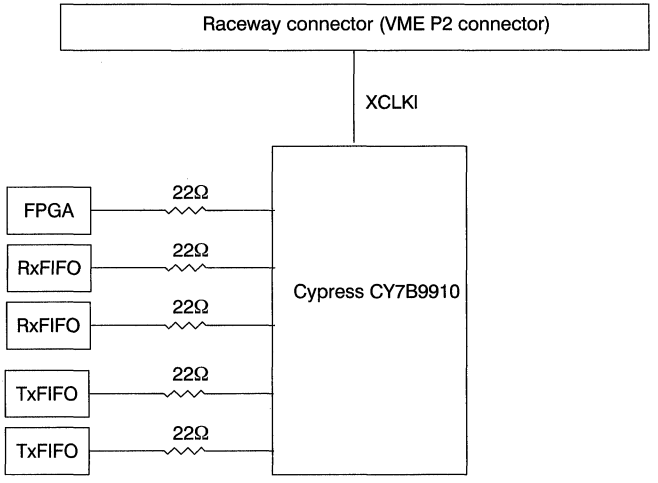


Figure 11. Distributing the XCLKI Clock

RACEway VME J2/P2 Connector

Table 8 describes the use of the VME J2/P2 connector pins for implementing RACEway.

Table 8. RACEway VME J2/P2 Pin Assignments

Pin	Signal	Pin	Signal	Pin	Signal
A1	XCLKI	B1	+5 VOLTS	C1	XRESETIO
A2	GND	B2	GND	C2	Reserved
A3	XBIO9	B3		C3	XSYNCI
A4	XBIO8	B4		C4	GND
A5	GND	B5		C5	XBIO7
A6	XBIO6	B6		C6	GND
A7	GND	B7		C7	XBIO11
A8	XBIO10	B8		C8	GND
A9	XBIO4	B9		C9	STROBIO
A10	GND	B10		C10	RPLYIO
A11	XBIO5	B11		C11	GND
A12	XBIO3	B12	GND	C12	REQI
A13	GND	B13	+5 VOLTS	C13	REQO
A14	RDCONIO	B14		C14	GND
A15	Reserved	B15		C15	XBIO2
A16	GND	B16		C16	XBIO1
A17	XBIO0	B17		C17	GND
A18	XBIO15	B18		C18	XBIO12
A19	GND	B19		C19	XBIO25
A20	XBIO24	B20		C20	GND
A21	XBIO31	B21		C21	XBIO29
A22	GND	B22		C22	XBIO30
A23	XBIO28	B23		C23	GND
A24	XBIO27	B24		C24	XBIO26
A25	GND	B25		C25	XBIO23
A26	XBIO22	B26		C26	GND
A27	XBIO20	B27		C27	XBIO19
A28	GND	B28		C28	XBIO21
A29	XBIO18	B29		C29	GND
A30	XBIO17	B30		C30	XBIO16
A31	GND	B31	GND	C31	XBIO14
A32	XBIO13	B32	+5 VOLTS	C32	GND

pASIC is a trademark of Quicklogic.

Interfacing to RACEway: PitCREWjr

- Used to interface between FIFOs and the RACEway protocol.
- Drives/receives a RACEway port directly.
- Simple master control, automatic slave response.
- Moves data at 160 MByte/sec peak and 140 MByte/sec sustained throughput.
- Implemented in a Cypress CY7C384A, a 2K gate 100-pin FPGA.

Reference Documents

When using this application note refer to the following documents for more information:

- Cypress CY7C384A and pASIC380™ family data sheets.
- *RACEway Interlink – Data Link and Physical Layers*, VITA 5–1994, available from the VITA Standards organization (VSO)
- Cypress CY7C4245 4K x 18 Synchronous FIFO data sheet

General

PitCREWjr is a simple full-duplex on-ramp to the RACEway fabric. The device has a standard RACEway port and FIFO port. The controller functions either as a RACEway slave, moving data between RACEway and local FIFOs or as a RACEway master, again moving data between RACEway and local FIFOs. It connects to and drives a RACEway interlink port, directly providing all required handshaking and control signaling. PitCREWjr's local FIFO port consists of a 32-bit bidirectional data bus and control signals for moving data between PitCREWjr and industry-standard FIFO components. The data flow between the RACEway and FIFOs is shown in *Figure 1*. The PitCREWjr has no programmable internal registers. Internal PitCREWjr state machines assemble and disassemble the route, address, and data long words embedded in the RACEway protocol. RACEway mastering is accomplished by controlling a single input signal. *Figure 2* shows the block diagram for PitCREWjr and *Table 1* shows the driver and signal name description for each pin on the PitCREWjr controller.

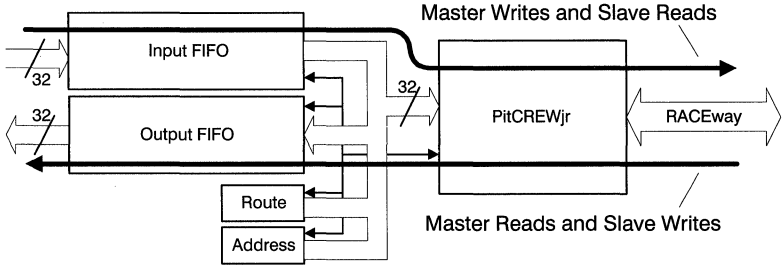


Figure 1. PitCREWjr Data Flow

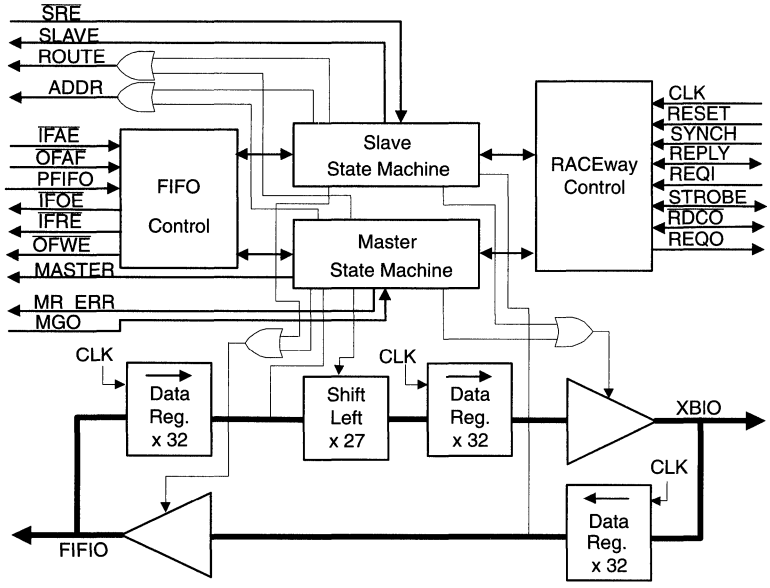


Figure 2. PitCREWjr Block Diagram

Table 1. PitCREWjr Interface Signals

Signal	Source	Function
FIFIO[31:0]	PitCREWjr/Input FIFO	FIFO Data Bus
XBIO[31:0]	PitCREWjr/RACEway	RACEway Data Bus
CLK	RACEway	Crossbar clock
RESET	RACEway	Reset from RACEway
SYNC	RACEway	Crossbar Sync – Provides control and phase information
REPLY	PitCREWjr/RACEway	Gives permission to send the address or data over the data bus
REQI	RACEway	Request In indicates the RACEway crossbar is requesting control of the data bus
STROBE	PitCREWjr/RACEway	Strobe indicates address or data is being sent on the data bus.
$\overline{\text{RDCO}}$	PitCREWjr/RACEway	Indicates to the crossbar to three-state the data bus so read data can be driven. It also indicates when a read error has occurred.
REQO	PitCREWjr	Request Out indicates the PitCREWjr is requesting control of the data bus
$\overline{\text{OFAF}}$	Output FIFO	Output FIFO almost full
$\overline{\text{OFWE}}$	PitCREWjr	Output FIFO write enable
PFIFO	User Hardware	Program output FIFO almost full flag
$\overline{\text{IFAE}}$	Input FIFO	Input FIFO almost empty
$\overline{\text{IFOE}}$	PitCREWjr	Input FIFO output enable
$\overline{\text{IFRE}}$	PitCREWjr	Input FIFO read enable
COUNT	PitCREWjr	Byte counter for master transfers
MR_ERR	PitCREWjr	Error occurred on a master read
MGO	User Hardware	Master GO – starts master state machine
SLAVE	PitCREWjr	Slave transaction in progress
$\overline{\text{SRE}}$	User Hardware	Slave read enable
ROUTE	PitCREWjr	PitCREWjr expecting route to be placed in FIFO data bus
ADDR	PitCREWjr	PitCREWjr expecting address to be placed on FIFO data bus
MASTER	PitCREWjr	Master transaction in progress

FIFOs

The timing generated by PitCREWjr is designed to match with CY7C4245 4K x 18 synchronous FIFOs. PitCREWjr signals can be connected directly to data and control signals of these FIFO components as shown in *Figure 3*. The input FIFO PAE flag should be set to 2. The output FIFO PAF flag should be set at least 16 entries from full.

Slave Function

The slave function of PitCREWjr is accessed whenever an incoming RACEway transaction is received on the RACEway port (REQI is asserted to PitCREWjr) During a slave transaction, the PitCREWjr asserts a status output pin called "SLAVE," which indicates that the PitCREWjr slave state machine is active. When a route word is received from the RACEway, it is driven onto the FIFO data bus. A PitCREWjr output called "ROUTE" is asserted for one XCLKI clock to indicate that a valid route word is present. When an address word is received from the RACEway, PitCREWjr drives this address word onto the FIFO data bus. An output called "ADDR" is asserted by PitCREWjr for one XCLKI clock to indicate that a valid address word is present on the FIFO data bus. PitCREWjr then acknowledges the RACEway with "REPLYIO."

The RACEway protocol communicates data direction in bit 1 of the address word. PitCREWjr's slave state machine branches on this bit value. If the direction of the data is from the RACEway to the local FIFO, the transaction is a slave write (bit 1 of address word is false). As data arrives from the RACEway, it is registered and driven onto the FIFO data bus. (See *Figure 2.*) The PitCREWjr writes the data received from the RACEway to the output FIFO by asserting "$\overline{\text{OFWE}}$" each time a valid word is ready on the FIFO data bus. A PitCREWjr input called "$\overline{\text{OFAF}}$" is used to indicate to PitCREWjr that the output FIFO is full. Assertion of "$\overline{\text{OFAF}}$" causes PitCREWjr to send a kill request to the RACEway master, effectively ending the RACEway transaction. "$\overline{\text{OFAF}}$" would typically be connected to the output FIFO programmable almost full flag. On completion of the RACEway data transfer, PitCREWjr three-states the FIFO data bus and deasserts the "SLAVE" status output.

If the direction of the data is from the input FIFO to the RACEway (a slave read, bit 1 of address word is true), then the FIFO data bus is three-stated by PitCREWjr and PitCREWjr asserts the signal "$\overline{\text{IFRE}}$" and then "$\overline{\text{IFOE}}$" to enable data from the input FIFO onto the FIFO data bus. PitCREWjr asserts this signal pair each time a new word is required from the FIFO. If the input FIFO becomes empty, as signaled by the "$\overline{\text{IFAE}}$" PitCREWjr input, PitCREWjr stops reading the input FIFO for the balance of that transaction and issues an error signal to the RACEway master on completion of the transaction. The kill request is also sent in this case, so that the master ends the transaction soon after the

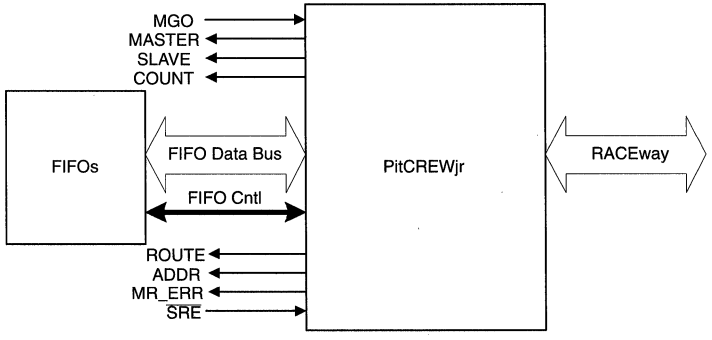


Figure 3. PitCREWjr Signals

underflow. On completion of the RACEway data transfer, PitCREWjr deasserts the "SLAVE" status output.

The intent of the "SLAVE" pin is to indicate a slave transaction in progress. It can be used to tag incoming data, select a data destination, or as a board logic control input.

Note that PitCREWjr will NOT cause route and address header words received from the RACEway to be written to the output FIFO. External logic would be required to place address and/or route words in the output FIFO.

Master Function

The master function of PitCREWjr is accessed whenever the "MGO" PitCREWjr input is asserted. The assertion of "MGO" launches the PitCREWjr master state machine. This state machine is clocked by the RACEway data clock "XCLKI". Two clocks after "MGO" is sampled asserted, PitCREWjr asserts its "ROUTE" output. Local board hardware should use "ROUTE" to enable a route word onto the FIFO data bus. PitCREWjr asserts its "MASTER" output when it drives this route word onto the RACEway and then drives the "shifted route" prescribed by the RACEway protocol. "MGO" should be deasserted once PitCREWjr's "MASTER" output is true. This is because "MGO" will cause a slave in progress to issue a kill over the RACEway. When "change to address" reply is received from the RACEway, "ROUTE" is deasserted, and one clock later "ADDR" is asserted. Local board hardware should use "ADDR" to enable an address word onto the FIFO data bus. PitCREWjr relays the address word to the RACEway and waits for a "$\overline{\text{DSE}}$" reply from the RACEway. When the reply is received, PitCREWjr deasserts the "ADDR" signal.

The RACEway protocol communicates data direction in bit 1 of the address word. PitCREWjr's master state machine branches on this bit value. If the direction of the data is from the local FIFO to the RACEway (a master write, bit 1 of address word is false), then data is read from the local input FIFO, registered inside the PitCREWjr, and driven onto the RACEway XBIO bus. The PitCREWjr FIFO data bus pins remain three-stated and PitCREWjr asserts the signals "$\overline{\text{IFRE}}$" and "$\overline{\text{IFOE}}$" to enable the input FIFO data onto the FIFO data bus. PitCREWjr asserts this signal pair each time a new word is required from the FIFO. If the input FIFO becomes empty, as signaled by the "$\overline{\text{IFAE}}$" PitCREWjr input, PitCREWjr stops reading the input FIFO and ends the RACEway transaction.

If the direction of data is from the RACEway to the local FIFO (a master read, bit 1 of address word is true), then as data arrives from the RACEway, it is registered inside the PitCREWjr and driven onto the FIFO data bus. The PitCREWjr writes the data received from the RACEway to the output FIFO by asserting "OFWEN" each time a valid word is ready on the FIFO data bus. A PitCREWjr input called "$\overline{\text{OFAF}}$" is used to indicate to PitCREWjr that the output FIFO is full. Assertion of "$\overline{\text{OFAF}}$" causes PitCREWjr to suspend transfer requests to the RACEway slave, effectively stalling the RACEway transaction until the signal is deasserted. "$\overline{\text{OFAF}}$" would typically be connected to the output FIFO programmable almost full flag. On completion of the RACEway data transfer as indicated by the deassertion of "MASTER," PitCREWjr three-states the FIFO data bus.

Additional Features

A slave read enable input "$\overline{\text{SRE}}$" is provided to lock out slave access from the RACEway side of the interface. This signal may be used to "protect" data in the input FIFO when that FIFO is being used for both master and slave data. Slave read can be disallowed when data is being queued up in the input FIFO for a master write.

The "MR_ERR" output of the PitCREWjr is an indicator that a master read operation received an error response from its target slave. The signal is a "one-shot", pulsing HIGH for one XCLKI clock period at the end of a master read access for which the RACEway slave signaled a read error.

The "COUNT" output signal strobes each time an 8-byte data beat occurs on the raceway when PitCREWjr is master. For writes, "COUNT" is as-

serted for each 8 bytes sent. For reads, "COUNT" is asserted for each 8 bytes requested.

The "PFIFO" input is used to assist in loading the output FIFO almost full flag. When "PFIFO" is asserted, PitCREWjr three-states its FIFIO data bus drivers, and asserts "$\overline{\text{OFWE}}$." The signal that connects to "PFIFO" can also be used to enable the "almost empty" value onto the FIFIO data bus.

PitCREWjr Operation

Figure 4 illustrates master write behavior. The "MGO" PitCREWjr input is asserted to start RACEway master (read or write) function. It should be deasserted when PitCREWjr asserts "MASTER". Master write is stopped by asserting "$\overline{\text{IFAE}}$" to the PitCREWjr. Notice that two data words are read after "$\overline{\text{IFAE}}$" is asserted. "ROUTE" and "ADDR" are shown enabling route and address information respectively onto the FIFIO data bus from external hardware. The "COUNT" PitCREWjr output pulses once for each 8 bytes sent over the RACEway.

Figure 5 illustrates master read behavior. Data arriving from the RACEway is to be taken from the FIFIO data bus on the rising edge of the RACEway data clock "CLK". Again "COUNT" pulses once for each 8 bytes requested from the RACEway. Master read is stopped by asserting the PitCREWjr input "$\overline{\text{OFAF}}$." Note that eight data values are delivered after "$\overline{\text{OFAF}}$" is signalled. This figure shows the timing when data traverses one RACEway crossbar. Latency will increase by two for each additional crossbar in the data path.

Figures 6 and *7* illustrate slave timing. "ROUTE" and "ADDR" PitCREWjr outputs mark the timing of valid route and address information on the FIFIO data bus. Bit 1 of the RACEway address field is captured by PitCREWjr, causing the appropriate FIFO control signalling for the data direction. For writes, "$\overline{\text{OFWE}}$" is asserted as data is driven by PitCREWjr onto the FIFIO data bus. For reads, "IFOEN" and "$\overline{\text{IFRE}}$" are asserted as shown and data is sampled from the FIFIO data bus on the rising edge of the RACEway data clock, "CLK".

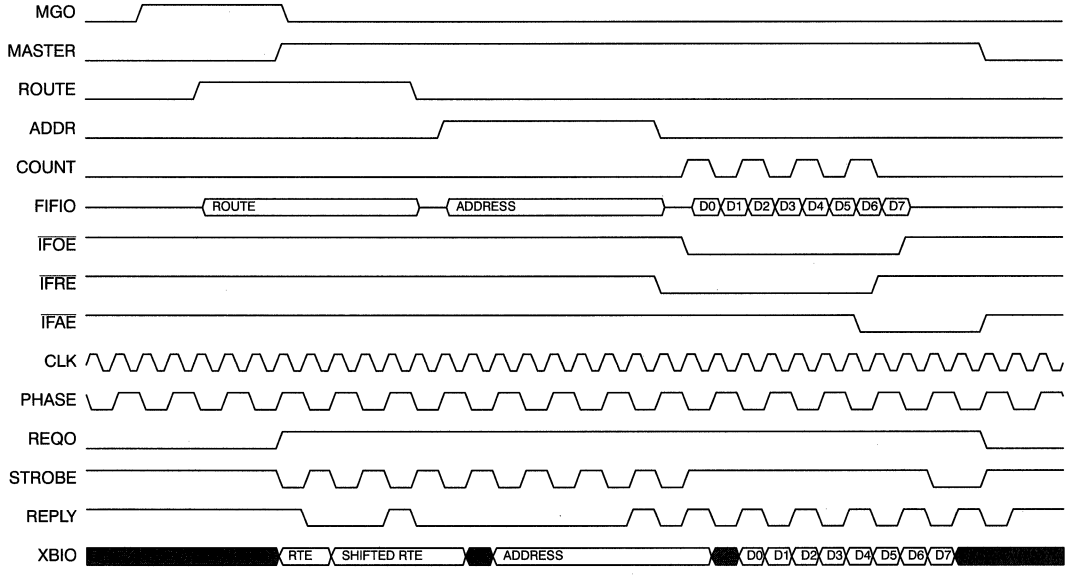


Figure 4. Master Write

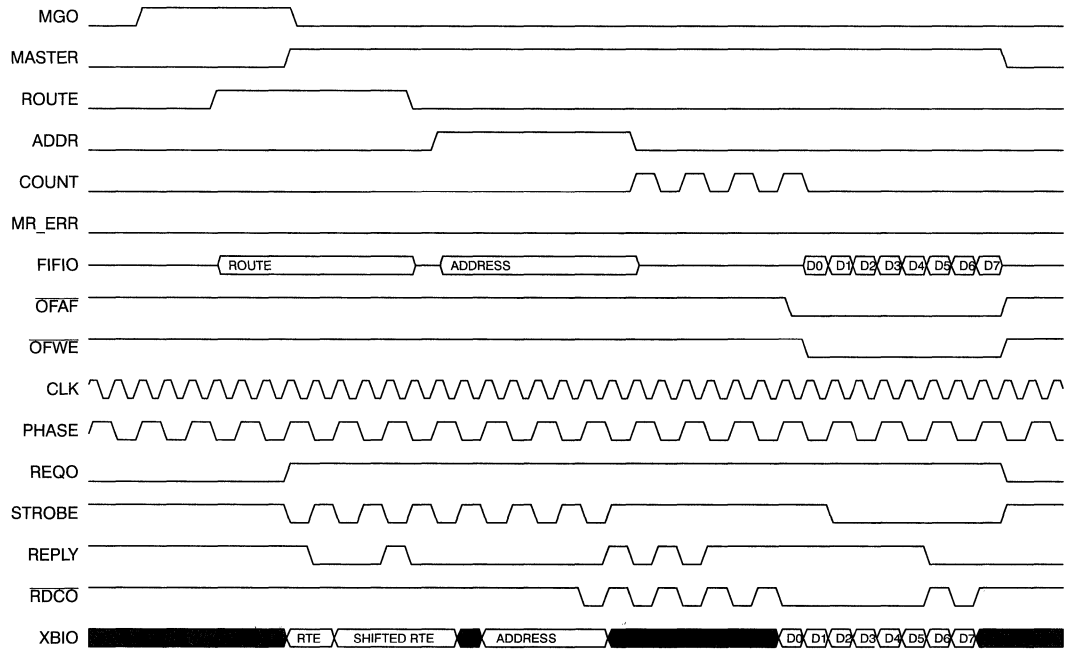


Figure 5. Master Read

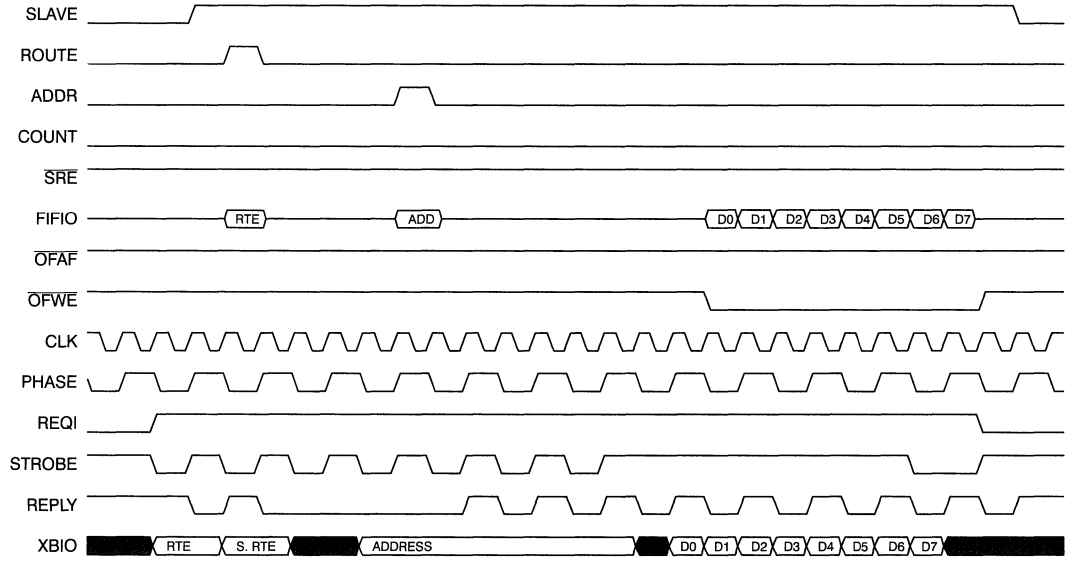


Figure 6. Slave Write

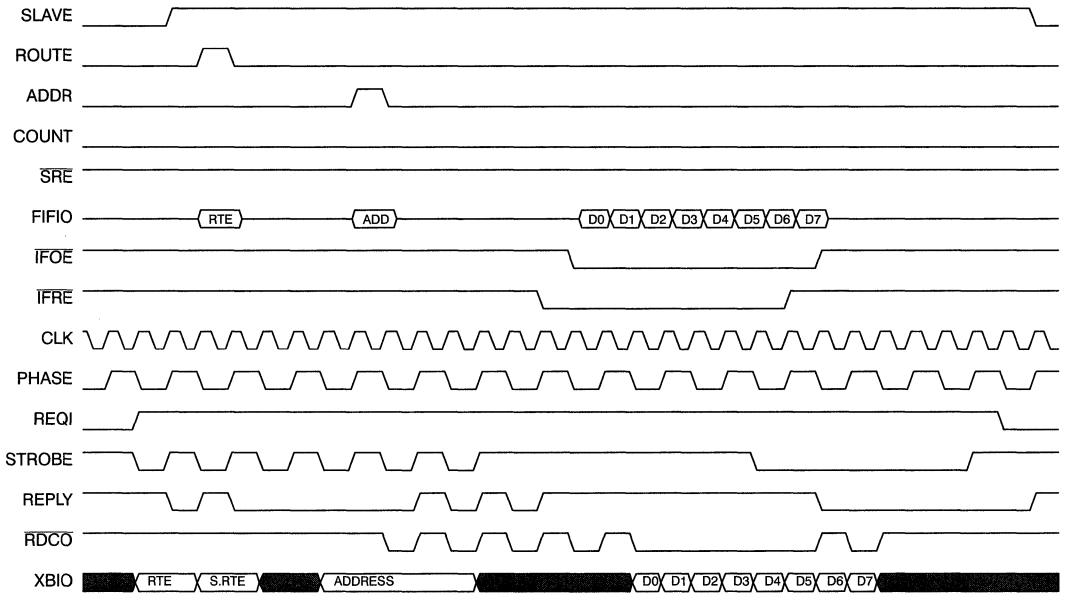


Figure 7. Slave Read

Figure 8 shows a PitCREWjr master writing to a PitCREWjr slave across one RACEway crossbar. The slave signals have an (S) suffix. In this example, the slave PitCREWjr input "$\overline{\text{OFAF}}$" signals that the slave is "almost full". The slave PitCREWjr signals "REQO" (a RACEway protocol kill). This kill propagates through the intervening RACEway crossbar to the PitCREWjr master, terminating the master transaction. The amount of data the slave must absorb after "$\overline{\text{OFAF}}$" is signalled is shown for a single intervening crossbar. Two additional FIFO write cycles will be required for each additional "crossbar hop".

Figure 9 shows the utility of the "$\overline{\text{SRE}}$" PitCREWjr input. It can be used to block PitCREWjr's response to a slave read from the RACEway. This feature allows the input FIFO facility to be multiplexed between master write and slave read without coordinating with the remote master across the RACEway. Master data being queued up in the input FIFO can be "protected" from a slave read operation as shown. The timing of "MGO" assertion with respect to slave arrival from the RACEway is arbitrary. "$\overline{\text{SRE}}$" may be deasserted any time after the assertion of "MASTER" by the PitCREWjr.

Figure 10 shows a PitCREWjr master reading from a PitCREWjr slave across one RACEway crossbar. The slave signals have an (S) suffix. In this example, the slave PitCREWjr input "$\overline{\text{IFAE}}$" signals that the slave is "almost empty." The slave PitCREWjr signals "REQO" (a RACEway protocol kill). This kill propagates through the intervening RACEway crossbar to the PitCREWjr master, terminating the master transaction. The slave PitCREWjr stops reading from its input FIFO two clocks after "$\overline{\text{IFAE}}$" is asserted; however, the RACEway protocol compels the slave to send until the RACEway master stops. By the time the PitCREWjr master responds to the kill, several long words of bad data have been written to the PitCREWjr master's output FIFO. The PitCREWjr output "MR_ERR" pulses HIGH for one data clock to signal that this error has occurred.

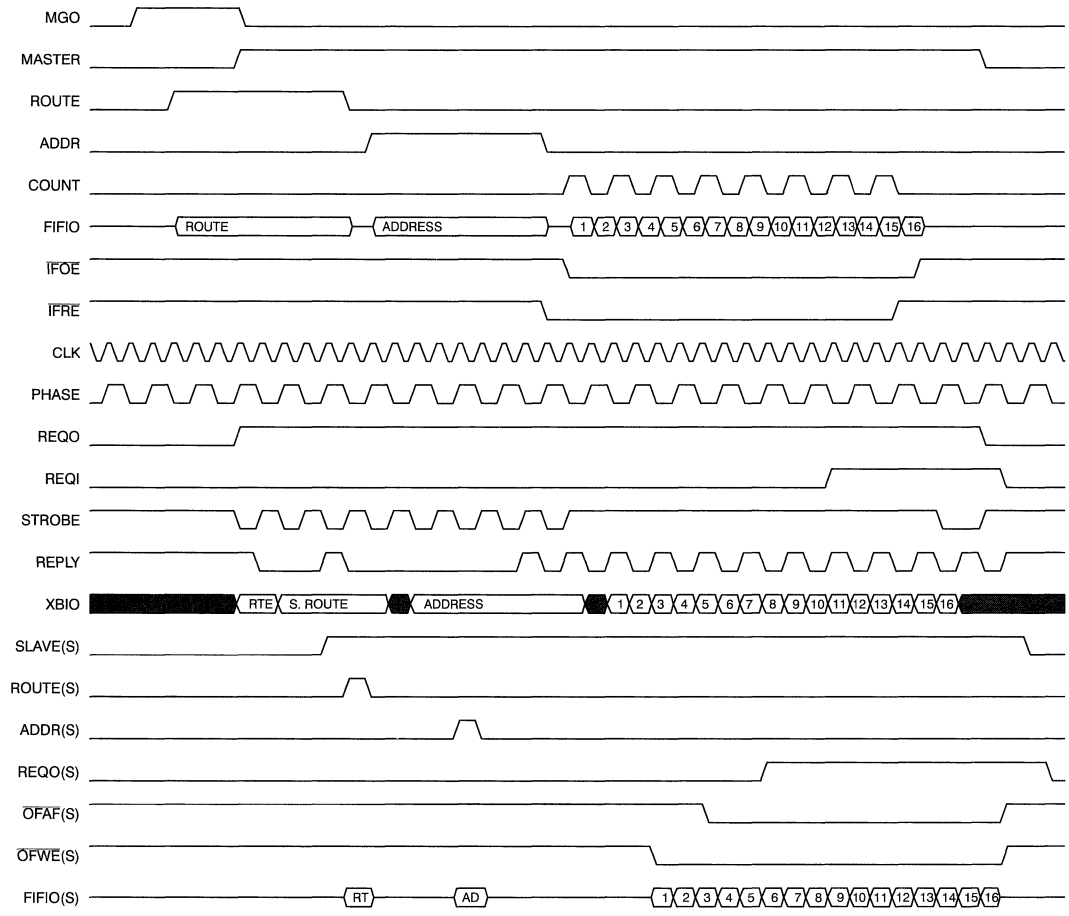


Figure 8. Master Write Overflow

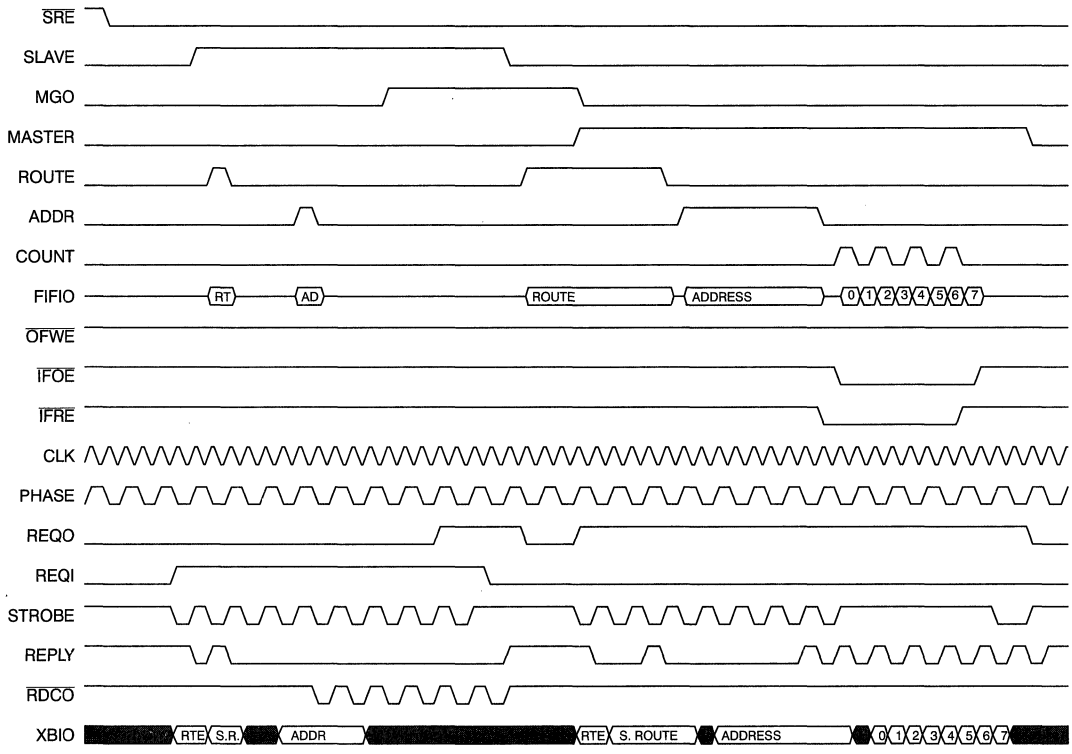


Figure 9. $\overline{\text{SRE}}$ Function

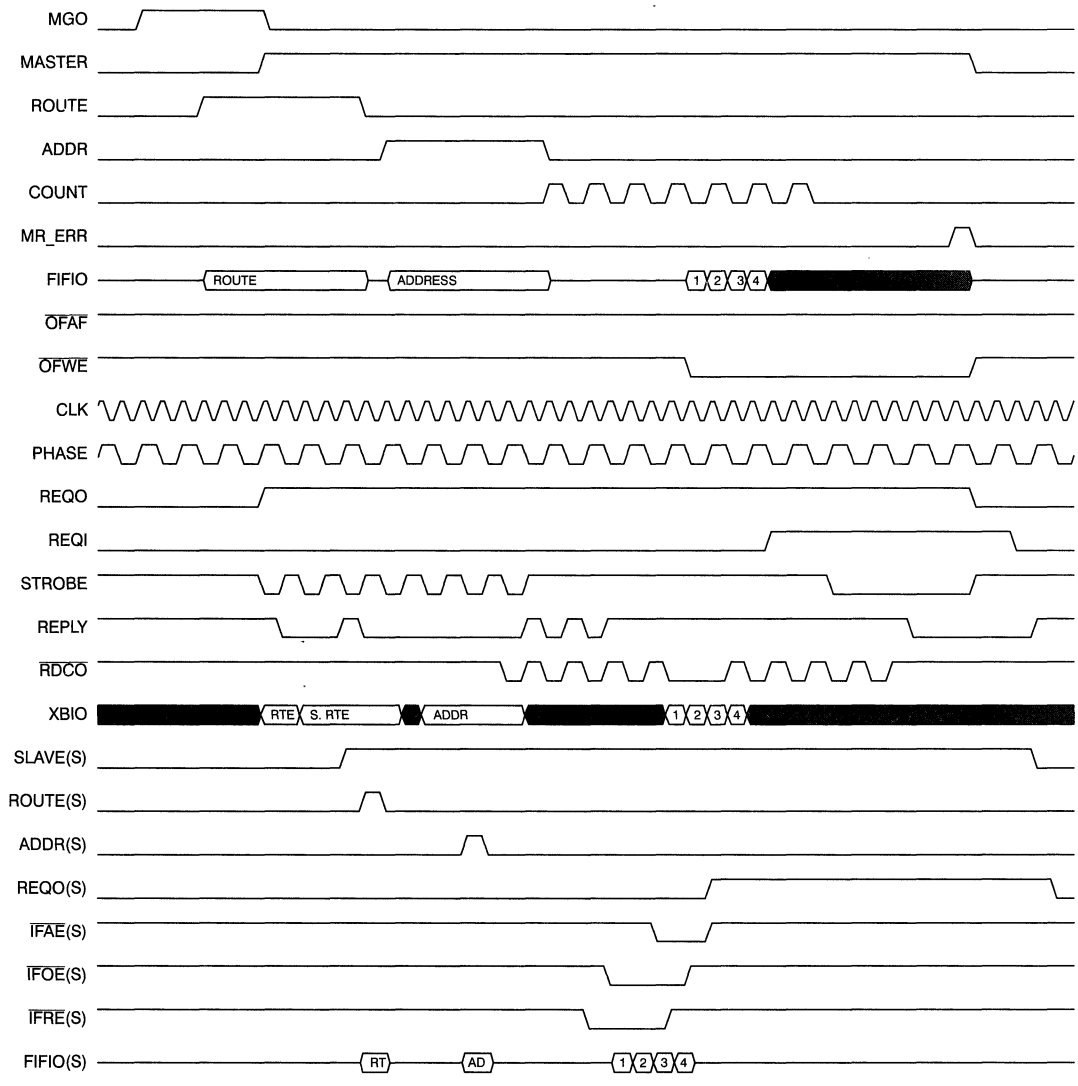


Figure 10. Master Read Error

CY7C384A Pin Table

Signal	Type	Pin No.
FIFIO_16	INOUT	1
XBIO_19	INOUT	2
FIFIO_19	INOUT	3
FIFIO_22	INOUT	4
XBIO_22	INOUT	5
FIFIO_25	INOUT	6
XBIO_25	INOUT	7
FIFIO_24	INOUT	8
VSS	-----	9
XBIO_24	INOUT	10
MGO	INPUT	11
CLK	INPUT	12
VCC	-----	13
UNUSED	INPUT	14
OFAF	INPUT	15
VCC	-----	16
XBIO_23	INOUT	17
FIFIO_23	INOUT	18
FIFIO_26	INOUT	19
XBIO_26	INOUT	20
XBIO_31	INOUT	21
FIFIO_31	INOUT	22
FIFIO_30	INOUT	23
XBIO_30	INOUT	24
FIFIO_27	INOUT	25
XBIO_27	INOUT	26
FIFIO_28	INOUT	27
XBIO_28	INOUT	28
FIFIO_21	INOUT	29
FIFIO_29	INOUT	30
XBIO_29	INOUT	31
XBIO_21	INOUT	32
PFIFO	INOUT	33
OFWE	OUTPUT	34
VSS	-----	35
SYNC	INOUT	36
REQO	OUTPUT	37
VSS	-----	38
MR_ERR	OUTPUT	39
STROBE	INOUT	40
RDCO	INOUT	41
VCC	-----	42
ROUTE	OUTPUT	43
COUNT	OUTPUT	44
REPLY	INOUT	45
MASTER	OUTPUT	46
SRE	INOUT	47
IFRE	OUTPUT	48
IFOE	OUTPUT	49
XBIO_2	INOUT	50

Signal	Type	Pin No.
FIFIO_2	INOUT	51
FIFIO_4	INOUT	52
XBIO_4	INOUT	53
XBIO_5	INOUT	54
FIFIO_5	INOUT	55
XBIO_3	INOUT	56
FIFIO_3	INOUT	57
FIFIO_1	INOUT	58
VSS	-----	59
XBIO_1	INOUT	60
IFAE	INPUT	61
RESET	INPUT	62
VCC	-----	63
REQI	INPUT	64
UNUSED	INPUT	65
VCC	-----	66
XBIO_7	INOUT	67
FIFIO_7	INOUT	68
XBIO_0	INOUT	69
FIFIO_0	INOUT	70
XBIO_8	INOUT	71
FIFIO_8	INOUT	72
FIFIO_9	INOUT	73
XBIO_9	INOUT	74
FIFIO_6	INOUT	75
XBIO_6	INOUT	76
XBIO_12	INOUT	77
ADDR	OUTPUT	78
XBIO_11	INOUT	79
FIFIO_11	INOUT	80
FIFIO_12	INOUT	81
XBIO_10	INOUT	82
XBIO_13	INOUT	83
FIFIO_10	INOUT	84
VSS	-----	85
XBIO_15	INOUT	86
FIFIO_13	INOUT	87
VSS	-----	88
XBIO_14	INOUT	89
FIFIO_15	INOUT	90
SLAVE	OUTPUT	91
VCC	-----	92
XBIO_18	INOUT	93
FIFIO_14	INOUT	94
FIFIO_18	INOUT	95
XBIO_17	INOUT	96
FIFIO_17	INOUT	97
XBIO_20	INOUT	98
XBIO_16	INOUT	99
FIFIO_20	INOUT	100

pASIC is a trademark of Quicklogic.

Glossary

10BASE-T: An IEEE 802.3 Standard for 10-Mb/s communication over 2 pair of twisted pair cable.

100BASE-T4: An IEEE 802.3 Standard for 100-Mb/s communication over four pair unshielded twisted pair cable running at 25 MHz.

4B/5B: An encoding method that takes four-bit data characters and maps each to a specific five-bit symbol. 4B/5B encoding also allows for transmission of command symbols outside the data space of 16 characters.

8B/10B code (8 bits to 10 bits): A patented coding method that converts "raw-data" to a form more suitable for transmission over a high speed serial interconnect link. This particular code insures a high transition density with a perfect DC balance.

Abel-HDL: Proprietary Hardware Description Language (HDL) from Data I/O Corp. Created as text input design language for their PLD/FPGA development software.

address phase: In PCI, the first part of a transaction in which the initiator sends out an address and command and waits for the addressed target to claim ownership of the transaction.

AHDL (Advanced Hardware Description Language): A high-level, modular language used to create logic designs for MAX EPLDs.

Alias SYNC: An unintentional SYNC character that occurs when transmission errors corrupt the serial data stream. It is possible to create a bit pattern that matches the SYNC character, but which is not correctly aligned with the serial byte boundaries. This Alias SYNC can make it impossible to correctly recover the data.

AND: The "and" logic gate.

ANSI (American National Standards Institute): A committee of numerous commercial, governmental, and educational constituents which conceive, formalize, and document standards for various applications, including information transport technologies such as Fibre Channel.

ANSI X3T9.3: The name of a data communication standard, sponsored by the American National Standards Institute, describing Fibre Channel.

arbitration: The process of deciding which one of two or more competing entities will be allocated a resource.

arbitrator: In PCI, the device that grants bus control requests to requesting initiator agents.

architecture: As pertains to VHDL logic synthesis, the declaration that specifies the behavior or structure of an entity. Entities and Architectures are always paired in VHDL descriptions.

artwork: The graphic materials generated for use in production of printed circuit boards, containing a representation of the copper circuit patterns in computer, mylar, or glass form.

associativity: The number of lines per set in a cache.

asynchronous: Referring to an operation that does not occur simultaneously with a specific time interval; i.e., the rising or falling edge of a clock pulse is not used as a timing reference signal.

Asynchronous Bus Protocol: A method of transferring data between a processor and peripheral that does not derive or rely on any timing parameters linked to a synchronous clock.

ATM: Asynchronous Transfer Mode. A circuit switched network protocol utilizing 53 byte cells, which promises to interface Local Area Networks (LANs) and Wide Area Networks (WANs) seamlessly. ATM was designed as a network that can provide services to multiple traffic types including isochronous (or time sensitive data like voice and video), as well as bursty traffic such as traditional data transfer.

ATM Forum: A committee of numerous commercial, government, and educational constituents which conceive, formalize, and document standards in support of ATM technology.

auto-negotiation: An IEEE 802.3 Standard for automatic configuration of a twisted-pair link without user intervention.

bad symbol: A special character that is transmitted when a receive error is detected in the physical layer.

bandwidth: (1) the absolute difference between the upper and lower frequency limits of operation. (2) The points in a spectrum where the circuit response is 3 dB down from nominal.

base address register: The register in the configuration space of PCI that holds the assigned address space values.

baseline wander: A low frequency variation in the relative threshold position at the receive end of a transmission line.

baud: The encoded bit rate per second. For binary communication channels, using Non-Return to Zero (NRZ) coding, 1 baud = 1 bit per second. In general, 1 baud = 1 symbol per second.

Behavioral VHDL: A description of how a design should operate or its behavior, as opposed to its structure. This is the highest level of abstraction for a VHDL description.

BER: Bit Error Rate, the ratio of corrupted data to correctly received data. This ratio is typically small and expressed as an exponent (i.e., $1X10^{-12}$, one error in 10^{12} bits). BER may be expressed in either bits in error or Bytes in error.

bias: The DC component of an AC signal or the DC level of a signal dictated by a resistor divider.

BiCMOS: Bipolar Complementary Metal Oxide Semiconductor. An advanced silicon process technology that combines the best features of bipolar technology (high speed) and CMOS technology (low power, high density), but at a cost penalty relative to pure CMOS.

bidirectional: Allowing data to flow in either direction (but not both simultaneously).

BIFO (Bidirectional FIFO): A FIFO, e.g., CY7C439, whose two sets of data pins can both be configured as either inputs or outputs, allowing transmission of data in both directions (though not simultaneously).

bipolar: A widely commercialized, silicon integrated circuit (IC) technology. Bipolar technologies create the highest performance silicon integrated circuits, but at the expense of high power consumption and the inability to make very large chips economically.

BIST: Built-In Self-Test, logic included in the chip that allows it to generate patterns and test them without external hardware or software intervention.

bit-cell: The nominal time period of a single bit in a serial data stream.

bridge: A device that connects two independent peripheral buses together, allowing them to communicate. It will perform the necessary bus protocol translations.

burst sequence: The sequence of addresses followed when multiple locations in a memory are being consecutively accessed in a single operation.

bus contention: A period in time when a common data or address bus may have more than one active driver on the bus at a given time.

bus switch: A device that can be used to isolate a device from a bus.

cache: A small, fast memory located between the CPU and main memory. A cache's purpose is to store copies of the instructions and/or data the CPU is most likely to need in the near future so that the

CPU can access them more quickly than if they were stored only in main memory.

cache hit: An access to main memory that is found in, and serviced by, the cache memory.

cache lock: A method, e.g. using a status bit, for ensuring that specific lines in the cache do not get replaced. Users can lock critical programs in the cache to ensure that performance on these programs is high and deterministic.

cache miss: An access to the main memory that is not found in the cache memory and therefore must be serviced by the main memory.

cache tag: A table of the current contents of a cache. The tag itself is made up of a varying number of address bits that uniquely identify each line in the cache as coming from a specific main memory line.

carrier: A signal whose presence is necessary to allow communications.

CAS (Column Address Strobe): In dynamic RAMs, the signal asserted to strobe the column address of the current access into the device after the row has been input.

cascading: Connecting several smaller parts, usually SRAMs, Dual-Ports, or FIFOs, together in such a way as to create an effective memory that is deeper or wider.

CELP: An industry standard reference for L2 cache module sockets.

chipset: One or more highly integrated chips that add features to a processor board.

clean: The status of a cache line when it contains the same data as the copy in main memory. Compare dirty.

clock generator: A circuit that is used to generate a clock signal used to trigger digital logic circuits.

clock stability: The stability of a clock signal with respect to frequency, pulse width, and amplitude.

CMOS levels: There are two sets of CMOS specifications: HC and HCT. The older HC devices are generally not TTL compatible, and the newer HCT (also FACT, FCT, etc.) are TTL compatible. Because the minimum V_{OH} level for TTL is 2.4V, TTL is not guaranteed to drive an HC input high. A 4Ω 10,000Ω pull-up resistor to V_{CC} at the TTL device's output enables the device to achieve the HC V_{IH} level of 3.5V.

coax (coaxial cable): A cable consisting of a single central conductor surrounded by a dielectric that spaces an overall cable shield from the central conductor.

collision: The condition caused by two or more Ethernet nodes transmitting at the same time.

combinatorial: A logic function that does not involve any synchronous elements.

component: A component is a VHDL design unit that may be instantiated in other VHDL design units. Before it can be instantiated, it must be declared using the COMPONENT declaration, which specifies the name of the component and lists its local signal names.

Concurrent Statement: As pertains to VHDL logic synthesis, a statement in an Architecture that executes or is modeled concurrently (simultaneously) with all other statements in the architecture.

converting ABEL: A technique whereby ABEL hardware descriptions are converted to VHDL.

CPLD: Complex programmable logic device.

CRC: Acronym for Cyclic Redundancy Check or Cyclic Redundancy Code. Used for error detection on serial data communication channels.

crosstalk: Coupling of electrical signals between conductors in a circuit. Often undesirable, crosstalk can corrupt data transfer by changing voltage levels to a level other than the intended value.

crystal oscillator: An oscillator with a crystal as the frequency setting element.

coherency (consistency): Agreement between shared contents of members of the memory system.

crosstalk: The temporal change in either the magnetic field or the electric field of a signal on one conductor that results in an unwanted signal being coupled to other conductors.

CSMA/CD (Carrier Sense Multiple Access with Collision Detection): The access method used by

the Ethernet MAC. This scheme detects when there is activity on the medium in order to avoid transmission on a shared medium. If the medium is clear, the MAC can transmit data. If the medium is busy, the MAC will wait to transmit data. If two or more MACs on the medium attempt to access the medium at the same time, a collision is detected and all MACs will stop transmitting and retry at a random time.

CY7C971: Cypress's 10BASE-T/100BASE-T4 Ethernet Transceiver.

cycle-cycle jitter: The change in a clock's output transition from its corresponding position in the previous cycle.

Cyclic Redundancy Check (CRC): An error control mechanism based on use of an error-detecting code. The code can be described as follows. Given a k-bit message, the transmitter generates an n-bit sequence (known as a check sequence) so that the resulting message, consisting of $k + n$ bits, is exactly divisible by some predetermined number. The receiver then divides the incoming message by the same number and, if the remainder is zero, assumes there is no error. The CRC codes are often expressed as polynomials.

daisy chain: A method of making connections serially, from some point to each next point in one continuous sequence (as in PCB layout).

data bus latency: The amount of time the data bus is driven after a given bus cycle terminates.

DCD (Duty Cycle Distortion): A deterministic jitter that is typically caused by mismatches within the serial transmission line interface. It causes rising edges to be misplaced in one direction and falling edges to be misplaced in the opposite direction (typically an identical offset).

DDJ (Data Dependent Jitter): A deterministic jitter that is a function of the characteristics of a particular serial interconnect media and the content of the serial data stream. It causes edges (either rising or falling) to be misplaced by a distance that varies as a function of their distance from preceding transitions. Variations in pulse width inherent in the data stream cause variations in the magnitude of the misplacement of the data transitions.

deadlock: The condition in which two or more processes that share resources halt because no process can obtain all the resources it needs to continue.

Dhrystone: A measurement of PC or microprocessor performance taken while running a benchmark program that consists of a loop of simple integer operations.

differential: Mode of communication in which two complementary signals are compared to each other to determine the logical state of the signal. Also known as a balanced connection.

DIMM: A dual-readout SIMM socket. Every pin on a DIMM socket can be a separate signal for a high pin count interconnection. See SIMM.

dirty: The status of a cache line that has been modified and now contains data different than the copy in main memory.

dispersion: Widening of a pulse as it travels down a transmission line due to characteristics of both the pulse and the media.

DMA (Direct Memory Access): A design technique that offloads some of the I/O processing from the CPU. A DMA controller allows the CPU to continue operation while the controller controls block transfers between I/O and memory or between separate memories in a multiprocessor system.

double oven oscillator: An oscillator that contains two ovens, with the crystal encased in the inner oven, and the temperature control circuitry and the inner oven encased in the outer oven.

DRAM (Dynamic Random Access Memory): The main (read/write) memory in almost all computers. Compare *SRAM*.

dual-port RAM: An SRAM that can process two different accesses simultaneously.

DUART (Dual Universal Asynchronous Receiver Transmitter): A pair of serial interfaces integrated into one chip.

duplex (also, full duplex): Capable of simultaneous bidirectional operation and having multiple sources and destinations.

duty cycle: The relationship of a clock pulse HIGH time to its LOW time—expressed as a percent.

ECC: Error Correction Code, used to ensure that data is correctly stored or transmitted.

ECL (Emitter-Coupled Logic): A convention for "one" and "zero" voltage reference levels in one integrated circuit family. ECL "one" and "zero" voltage levels are very small and, therefore, are able to be sent into and out of integrated circuit packages very quickly. ECL logic is used in the fastest available computers, such as those from Cray and Convex. ECL circuits are fabricated in Bipolar or BiCMOS technology. Compare TTL.

EDC: Error detection/correction. Hardware or software used to generate/check ECC bits. Upon single-bit error detection, the EDC will also correct the faulty bit.

EEPROM (Electrically Erasable Programmable Read Only Memory): A PROM that can be erased and reprogrammed electrically. See PROM.

effective access time: A cache performance metric giving the average time required to service a memory reference.

emulation: In circuit verification, using a separate piece of hardware which takes the place of an IC or subsystem in the circuit under test.

Entity: As pertains to VHDL logic synthesis, the declaration that lists or describes the ports (the interfaces to the outside) of the design. An Entity describes the names, directions, and data types of each port.

EPROM (Erasable Programmable Read Only Memory): A PROM that can be erased and reprogrammed. See PROM.

equalization: The application of frequency selective gain or attenuation to compensate for distortion. Equalization is often used to increase the distance over which a communication channel can operate. Usually, a system is equalized for a given distance, and must be re-equalized if that distance is increased or decreased or if the data rate is changed.

error-free window: The widest possible area within which a transition can occur and be correctly interpreted by the receiving circuit; a measure of jitter tolerance.

ESCON: A protocol used to interconnect IBM compatible computers at data rates of 20 MByte/sec.

Ethernet: The physical layer and control standards that are encompassed in the IEEE 802.3 Standard. Ethernet uses a shared network topology with an access method known as CSMA/CD (Carrier Sense Multiple Access with Collision Detection).

expanders: Extra product terms in MAX EPLDs that are available to be used and shared by all macrocells in a Logic Array Block (see LAB).

eye pattern: Method of examining a data stream that compares the stable versus unstable portions of a bit-cell.

fall time: The amount of time it takes a digital logic signal to transition from a logic HIGH to a logic LOW.

FDDI: Acronym for Fiber Data Distribution Interface. A high-speed, local-area network, using a pair of fiber-optic links in a dual token-ring topology. Data rates of 10 Mbytes per second are supported.

FFT (Fast Fourier Transform): A mathematical method for determining the frequency spectrum of a waveform.

fiber-optic: A reference to components whose primary mode of operation is through the use of optical rather than electrical energy.

Fibre Channel: An ANSI-standard data communications interface for computers and peripherals. A high-performance computer interconnect standard that describes a method of interconnecting computers and peripherals at specified data rates between 13 and 100 MByte/sec.

FIFO – First-In First-Out Memory: A memory device in which data is accessed from memory in the order that it was written into memory.

finite state machine: A synchronous sequential circuit, the outputs and next state of which are functional logic functions of the inputs and current state.

Fourier transform: A mathematical operation used to convert time-domain expressions into equivalent frequency domain expressions.

FPGA: A field programmable gate array.

framer: The internal logic included in the HOTLink Receiver that examines the serial bit stream and looks for the SYNC character. When it is found, the framer logic aligns the deserializer with the transmitted byte boundaries.

framing: (As it applies to the CY7B933 HOTLink Receiver) The process of determining what the proper byte boundaries are in a serial bit data stream.

frequency synthesizer: A device that uses PLLs to generate one or more output frequencies from a reference frequency. Also called clock generator or clock synthesizer.

full duplex: See duplex. Compare simplex, half duplex.

glue logic: Either 74 series or programmable logic (PLD or CPLD) that implement a function that was not integrated into the chipset. These are usually high-current buffers that improve the drive capability of the chipset.

Green PC: Refers to a PC that, when idle, does not consume more than a specified maximum power, as defined by the U.S. Environmental Protection Agency (EPA).

ground bounce: When many outputs of a device change from HIGH to LOW there is a rush of current into the output drivers. If the inductance to ground is sufficient, the virtual ground level is raised due to this inductance. The voltage spike caused by this phenomenon is called ground bounce.

HBM (Human Body Model): A model of Electro Static Discharge (or ESD) hazards based on static discharges observed between humans and electronic devices. Semiconductor manufacturers use the HBM to design ESD protection circuits into their products.

half duplex: A device or system that can transmit information in two directions, but not simultaneously. The Cypress CY7C439 BIFO is a half-duplex device. Compare simplex, duplex.

HIPPI: Acronym for HIgh Performance Peripheral Interface. A standard way of interconnecting high performance peripheral devices to medium- and large-scale computers. The interface is characterized by a parallel bus using ECL logic levels and is capable of data transfer rates of several hundred Mbytes per second over relatively limited distances.

HOTLink: The name for Cypress's High-Speed Optical Transceiver Link chip set.

Hysteresis: In general, the failure of a property that has been changed by an external agent to return to its original state when the cause of the change is removed.

idle: The state in which the 10/100BASE-T Ethernet transceiver is not transmitting frames.

initiator: The agent in PCI that has the current control and operation of the bus.

instantiate: The use of a previously designed module in a schematic or computer program (such as a VHDL model).

IPI: Acronym for Intelligent Peripheral Interface. Originally defined in IPI1 as a controller interface for high-performance disk drives, the standard has evolved in IPI3 to a relatively complete channel interface intended for general-purpose high-speed I/O in medium- to large-scale computer systems.

jam: A special pattern that is transmitted when a collision is detected.

jabber: The condition caused by a node that is continually transmitting.

jitter: A typical form of corruption that occurs on serial data streams. It is a displacement of the timing of a transition from its ideal position. The two basic types of jitter are Random and Deterministic. Deterministic jitter is further divided into DCD and DDJ.

jitter tolerance: The ability of the deserializer to recover data from a corrupted serial data stream. This specification indicates tolerance to displaced transitions within the expected bit window. This tolerance

may be expressed in time (i.e., nanoseconds) or as a percentage of a bit time (i.e., ±45% of a bit time). See error-free window.

K28.5: A special character that is defined in the 8B/10B code. This character is typically used as an idle or fill character when no data is to be transmitted on the serial media. Sometimes referred to as a Sync Character. See SYNC.

LAB (Logic Array Block): In Cypress MAX PLD devices, the LAB represents a separate functional block in the device. Each type of MAX PLD has a different number of LABs.

latch-up: A regenerative phenomenon that occurs when the voltage at an input pin or an output pin is either raised above the power-supply voltage potential or lowered below the substrate voltage potential, which is usually ground.

level one cache (L1): The cache that is integrated into the processor. The L1 cache improves performance by reducing the volume of data transferred between the processor and external memory.

level two cache (L2): The cache between the L1 cache and main memory. The L2 cache improves performance by reducing the volume of data transferred between the L1 cache and main memory.

LFSR: Linear Feedback Shift Register, used to generate a pseudorandom sequence of characters. The LFSR in the HOTLink is used to generate and check the BIST sequence.

library: A logical storage facility for design units. Before a component can be instantiated in a higher-level design unit, its package must be compiled into a library that is visible to that design unit, usually the current work library.

line (block): The basic unit of information exchange between a cache and main memory or between a parent cache and its child(ren) cache(s).

line size: The number of bytes or words in one cache/main memory line. In a cache system, a line is the quantum of data identified by the cache tag and is the smallest quantum of data that can be transferred between the cache and main memory. Whenever a new entry is placed into the cache, one line is transferred. Common line sizes are 16 and 32 bytes.

Linear Feedback Shift Register (LFSR): A shift register using XOR gates and feedback to implement cyclic redundancy check polynomials.

link pass state: The condition entered when a an operational link is established between two nodes.

local bus: The peripheral bus connected directly or "local" to the CPU itself. This bus will usually have better performance than a nonlocal bus.

logic cell: A replicated element within an FPGA typically containing a register and additional combinatorial logic. It is the basic building block used for implementing circuits in the FPGA.

long-term jitter: Measures the maximum change in a clock's output transition from its ideal position over many cycles.

MAC (Media Access Control): The MAC is the control structure that governs access to a communication medium. It also governs how data is encapsulated or framed on medium and usually includes a basic form of error detection.

MACH: The trademark for Advanced Micro Devices' family of complex programmable logic devices.

macrocell: A low-level block of logic in programmable logic devices. This block can include one or more registers along with configurable feedback and/or output paths.

master device: A device that controls the timing for data exchanges between two devices. When devices are cascaded in width, the master device is the one that controls the timing for data exchanges between the cascaded devices and an external interface. The controlled device is called the slave device.

MAX7000: the trademark for Altera's family of complex programmable logic devices

Mealy machine: A state machine in which outputs depend on the present state and the previous value of the inputs.

Media Independent Interface (MII): An IEEE 802.3 standard interface between MAC devices and

physical layer devices. The MII supports operation at 10 Mb/s and 100 Mb/s.

metastable: A condition in which neither a logic zero nor a logic one can be guaranteed, due to a timing violation to a synchronous logic element.

Moore machine: A state machine in which the outputs depend only on the current state.

MTBF (Mean Time Before Failure): The average length of time a system or component will continuously operate between failures, given a defined set of operating conditions.

multimode: Fiber-optic communication where light propagates in one or more modes through the optical media.

multiprocessing: A computer architecture in which two or more processing units are coupled together to run different programs simultaneously while sharing the same computer frame and memory.

Non-Return-to-Zero-Invert (NRZI): A method of encoding a serial bit stream. A transition indicates a 1 and no transition indicates a 0, hence the term non-return-to-zero. The waveform doesn't return to zero to indicate a bit value.

NTSC (National Television Systems Committee): The standard video format used in the USA.

Number Representations: Required VHDL syntax for binary, octal, decimal, and hexadecimal numbers.

OLC (Optical Link Card): The OLC is a LED/laser-based data-communications adapter card based on the Fibre Channel standard.

optical module: A device capable of bidirectional conversion of electrical signals to optical signals for use in communicating over fiber-optic cables.

OR: The "or" logic gate.

oscillator: A circuit that is generally crystal controlled and is used to generate a clock frequency.

oven controlled oscillator: An oscillator that encases its crystals in a temperature-controlled oven, in order to maintain a precise operating temperature at the crystal.

overshoot: The amount by which the amplitude of a signal exceeds its final value on a LOW-to-HIGH transition.

package: A package is a collection of VHDL declarations that can be used by other VHDL descriptions. For the purpose of creating hierarchical designs, a package consists of one or more components. However, a package may also include other types of declarations.

PAL (Phase Alienation by Line): A standard video format used in Europe and the Far East.

parallel-resonant crystal: A piezoelectric device that exhibits a maximum-impedance resonance. Because the operation of such a crystal depends on the load it "sees," the capacitive loading of a parallel-resonant crystal must be specified when the crystal is ordered.

parity: An error detection scheme in which a status flag is saved, indicating that the number of "on" bits is even or odd.

partition: The disabling of an Ethernet port.

PCB (Printed Circuit Board): A system building block that allows connecting integrated circuits together.

PCI bus (Peripheral Component Interconnect bus): A high-bandwidth, processor-independent peripheral bus (32 bits, expandable to 64; 33 MHz, expandable to 66 MHz) that has a potential data transfer rate of up to 132 MBytes/sec.

PECL: A variation of ECL often referred to as Positive-ECL or Pseudo-ECL in which the devices are operating from a positive power supply instead of 0 volts to −5.0 volts.

period jitter: Measures the maximum change in a clock's output transition from its ideal position.

photodiode: Optoelectric device capable of converting changes in received light amplitude into changes in current.

Physical Coding Sublayer (PCS): The PCS is a sublayer contained within the Ethernet Physical layer standard. This sublayer is responsible for digital functions such as data encoding and serial to parallel conversion.

Physical Layer: The devices and components that attach directly to the physical communication media. These include drivers, shifters, filters, etc. that are needed to implement the physical requirements of the communication protocol. The Physical Layer is usually the lowest layer of a communication protocol stack.

PIA (Programmable Interconnect Array): In Cypress MAX devices, the PIA is the routing path between separate logic array blocks (LABs). The PIA routes automatically and provides uniform timing throughout the devices.

PIM (Programmable Interconnect Matrix): In Cypress FLASH370 devices, the PIM is the routing path between separate logic array blocks (LABs). The PIM routes automatically and provides uniform timing throughout the devices.

PLD (Programmable Logic Device): An integrated circuit that is shipped blank to customers and can be field programmed into a custom logic circuit, such as a counter, an adder, or a state machine.

PLL (Phase-Locked Loop): A circuit used to minimize clock skews by keeping them in phase with respect to a reference clock. Also used to generate a clock that is a multiple frequency of the reference clock.

plug and play: The concept of the ability of a product to be easily installed into a system with minimal or no user configuration.

PMA (Physical Medium Attachment): The portion of the transceiver that interfaces with the shared medium.

Polarity Conventions: Rule of thumb for assigning and interpreting polarity in VHDL.

PQFP (Plastic Quad Flat Pack): A plastic package with flat-pack style pins on all four sides of the part.

preamble: The first 8 bytes of an Ethernet frame.

process: As pertains to VHDL logic synthesis, a collection of Sequential Statements appearing in a design Architecture. The Process itself is evaluated concurrently within the Architecture.

product term: A Boolean AND of all the inputs to a PLD array.

PROM (Programmable Read-Only Memory): Memory in which the data is fixed even when the power is turned off. Programmable ROMs are shipped blank to customers and customized in their facilities.

protocol: A set of rules that govern network communications. Low-level protocols define transmission rates, data encoding schemes, physical interfaces, network addressing schemes, and the method by which nodes contend for the chance to transmit data over the network. High-level protocols define functions such as printing and file sharing.

QuietBus: A technique in which a bus is not driven unless the address is decoded to be within the requested address space.

RACEway Interlink: The official name of the ANSI standard, which describes how to make a crossbar-based communication system including electrical specifications and logical protocols for the data transmission. The word "Interlink" conveys that the standard is communication oriented and covers more than one participating device. Although the RACEway Interlink standard does not specifically mention it, it is a perfect description of the way the Cypress CY7C965 works.

random jitter: Random jitter is a measure of edge displacement that is uncorrelated with either the interconnect media or the serial data stream. It is usually caused by random effects in the interconnect system or by thermal effects in the high gain amplifiers used to translate between optical and electrical information.

RAS (Row Address Strobe): In dynamic RAMs, this signal is asserted to strobe the row address into the device; the address inputs are time-multiplexed.

recursion: see *recursion*.

reference: A request by the processor to read or write a memory location.

reframe: To determine and align the deserialization logic with correct byte boundaries, so that the data can be decoded correctly.

refresh: The periodic replenishment of the charge on storage capacitors used in DRAM cells.

rise time: The amount of time it takes a digital logic signal to transition from a logic LOW to a logic HIGH.

RTC (Real Time Clock): A peripheral clock chip that operates from an integrated battery when the system power is off.

RTL (Register Transfer Level): A level of description in hardware design languages that consists of operations being described in terms of register- and gate-level structures.

run length: Run length can be either the distance between transitions (i.e., the maximum number of adjacent ones or the maximum number of adjacent zeros) in a serial data stream; or the length of time that an error will propagate after an error event. In the first case, the 8B/10B code rules allow a run-length of five (5) bits. In the second case, a single error event can occur within a single byte, and be terminated at the next one, or in the case of a running disparity error (or a framing error) the effect of the error can continue for an indeterminate time.

running disparity: Running disparity is a concept included in the 8B/10B code that allows it to ensure a perfect DC balance. It is a measure of difference between the number of 1s (high-bits) and number of 0s (low-bits) and is automatically managed by logic that selects alternative codes from the possible code tables to assure a perfect match.

running disparity error: A type of error in a serial data bit stream in which there are too many consecutive bits at a single logic level for the data received to be valid.

SBCCS (Single Byte Command Code Set): A command set defined as a Fibre Channel level 4 protocol. The set is characterized by having, in all cases to command defined in the first byte, and all subsequent bytes providing only parametric information relating to the command.

SCSI (Small Computer Systems Interface): A standard way of interconnecting peripheral devices, such as disk and tape to small to medium sized computers. It is specified in a document from the ANSI committee X3.31. Up to seven storage devices can be attached to a single computer using a single SCSI network.

SECAM (*Systeme Sequentiel a Memoire*): A standard video format used in France and Europe.

semaphore: A software technique for providing explicit mutual synchronization of parallel sequential (software) processes. Semaphores are initialized with the value zero or one before the processes are started. After initialization, the processes access the semaphores only via two specific operations—the so-called synchronizing primitives. The operations carried out on semaphores are referred to as P and V, which are the first letters of the Dutch words corresponding to WAIT and SIGNAL, respectively.

Sequential Statement: As pertains to VHDL logic synthesis, it is a statement appearing within a Process. All statements within a Process are executed or modeled in order, similar to programming languages such as C or Pascal.

set: A collection of cache locations in which a line may reside.

set associativity: A property that allows a cache to be divided into sets, each of which contains one or more lines. This property enables a line of main memory to map to more than one line in the cache; the line of main memory can map to one line in each of the sets. When searching the cache, the tags of one line from each of the sets are compared to the reference tag concurrently, to determine to which set, if any, the main memory line was mapped.

shielded twisted pair: Copper cable consisting of two insulated conductors twisted together in a controlled fashion, having an overall cable shield that is isolated from both conductors.

skew: The variation in time of two signals specified to occur at the same time.

SIMM (Single Inline Memory Module): A memory packaging option commonly used for DRAMs.

simplex: A device or system that can transmit data in only one direction. Compare half duplex, duplex.

simulation: In circuit design, the modeling of an electronic circuit's function using a computer software.

single-ended: Mode of communication in which a received signal is compared to an internal or external fixed reference to determine the logical state of the signal. Also known as an unbalanced connection.

single mode: Fiber-optic communication in which light propagates in only one mode through the optical media.

slave device: A device that allows another device to control the timing for data exchanges between them. Also, when devices are cascaded in width, the slave device is the one that allows another device to control the timing for data exchanges between the cascaded devices and an external interface. The controlling device is called the master device.

slew: The rate of change of voltage or frequency with time.

snooping: A method used in multimaster applications in which one or more of the masters contain data or instruction cache. Cache coherency and maintenance operations occur when the active master requests an operation on data that happens to be contained in a non-active master's cache. The non-active master can intervene and, depending on the type of transfer, maintain its cache accordingly and possibly supply its cached data to the active master. The act of monitoring the bus address and data by the non-active master is considered "bus snooping."

SONET (Synchronous Optical NETwork): A standardized frame format used by telecommunication carriers to encapsulate data and transmit that data over a WAN.

spectrum analyzer: A frequency domain oscilloscope.

SRAM (Static Random Access Memory): A Random Access Memory allows the user to store and retrieve data at a high rate of speed. The term "static" means that so long as the power is on, the memory will retain its data. This feature contrasts with Dynamic Random Access Memories (DRAMs) that store data in a temporary medium, which allows the data to fade away every few milliseconds. DRAMs must have their data refreshed continuously, even when the power is on, but they provide greater density at lower costs than SRAMs, although they may be slower.

starvation: The condition in which one process that shares resources with other processes halts due to the fact that it can not obtain the resource(s) it needs to continue.

STP (shielded twisted pair): Similar to UTP but surrounded by a metal shield.

Structural VHDL: A description of how the various components that make up a design are connected; the lowest level of abstraction for a VHDL description.

sum-of-products: A Boolean algebra construct in which inputs are logic ANDed and the outputs of the AND gates are ORed together. This is how most PLDs are constructed.

SVIC: Slave VME interface card.

SYNC: The special character included in the 8B/10B code that allows the serial data stream to be properly decoded. This character (K28.5) contains a unique sequence of bits that can never occur with any combination of legal data bytes in an undamaged data stream.

synchronous: Said of a system or signal when the rising edge of a clock pulse is used as a reference signal.

target: The agent in PCI with which the initiating agent is involved in a transaction.

temperature compensating oscillator (TXCO): An oscillator that contains circuitry that compensates for temperature changes and hence combats frequency variations.

terminate: To match the impedance of a driver to a line or a line to a load.

Test pin: A pin on the CY7C971 that is only used for factory testing. This pin should be tied LOW to permanently disable the test mode.

Thévenin: A type of circuit used to terminate a transmission line.

three-state: A signal that can be at a HIGH or LOW logic level, or in a high-impedance state.

token passing: (as applied to state machines) A design methodology in which an n-bit state machine is built with n 1-bit registers, instead of with $\lceil \log_2(n) \rceil$ registers. In a token-passing state machine, the state is indicated by the specific 1-bit register that contains the only "1," and state transitions are accomplished by passing the "1" (i.e., the token) from one register to another.

transaction: In PCI, the process of establishing a communication link between two device agents (i.e., CPU and peripheral) and transferring data.

transformed transaction: A transaction that is changed from its original intent, e.g., a read becomes a write and a write becomes a read.

transformer: Electrically isolates the Ethernet transceiver from the media.

transimpedance amplifier: Amplifier designed to convert a small change in current into a large change in voltage.

translation: Conversion from one standard to another.

translator: A device that converts from one standard to another.

transparent write: A write in which the data appears at the outputs as the data is written into the array. Possible only on separate I/O RAMs.

transmitter: A circuit used to send information.

TTL (Transistor-Transistor Logic): The dominant convention for "one" and "zero" voltage reference levels in integrated circuits. TTL circuits are pervasive in most electronics applications, including personal computers, workstations, and consumer electronics. See ECL.

twinax (twinaxial cable): Copper cable consisting of two insulated conductors assembled parallel to each other and having an overall cable shield that is isolated from both conductors.

UART (Universal Asynchronous Receiver Transmitter): A device that provides serial communication capabilities for a system.

uniprocessing: A computer architecture in which one processing unit runs all programs.

UTP (unshielded twisted pair): Telephone type cable in which two wires are twisted together to form a pair. As the name implies, there is no metal shielding around the cables.

UVEPROM (Ultraviolet Electrically Programmable Read Only Memory): An EPROM that can be erased using an ultraviolet light. See PROM, EPROM.

VAC: VMEbus Address Controller.

VCO: Voltage controlled oscillator; e.g., a clock generator that uses input voltage levels to vary the clock frequency.

VESA bus: A local bus standard that extended the existing ISA bus to increase throughput.

VHDL (VHSIC–Very High Speed Integrated Circuit Hardware Description Language): A standard (IEEE 1076) software language for describing and simulating hardware designs, from transistor level up to full-system level. It is the language used in Cypress's *Warp* PLD design tools.

ViaLink: The programmable antifuse element used to connect wires in a pASIC FPGA.

VIC: VME Interface Controller.

VITA: VME International Trade Association.

VME: VERSAModule Eurocard.

VSO: VITA Standards Organization.

watchdog timer: A watchdog timer limits the amount of time a system will wait for a bus cycle termination signal (e.g., $\overline{\text{RDY}}$). If the watchdog timer completes, the system assumes that an error has occurred and responds appropriately.

XOR: The "exclusive-or" logic gate.

Index

An italicized page number means the reference is to a figure or table.

Symbols

Numbers

A

B

C

D

E

F

G

H

M

S

T

U

V

W

X

Z

Domestic Direct Sales Offices

Corporate Headquarters

Cypress Semiconductor
3901 N. First Street
San Jose, CA 95134
(408) 943–2600
Telex: 821032 CYPRESS SNJ UD
TWX: 910 997 0753
FAX: (408) 943–2741

IC Designs Division
12020–113th Ave. N.E.
Kirkland, WA 98034
(206) 821–9202
FAX: (206) 820–8959

Alabama

Cypress Semiconductor
4940B Corporate Drive
Huntsville, AL 35805
(205) 721–9500
FAX: (205) 721–0230

California

Northwest Sales Office
Cypress Semiconductor
100 Century Center Court
Suite 340
San Jose, CA 95112
(408) 437–2600
FAX: (408) 437–2699

Cypress Semiconductor
23586 Calabasas Rd., Ste. 201
Calabasas, CA 91302
(818) 222–3800
FAX: (818) 222–3810

Cypress Semiconductor
2 Venture Plaza, Suite 460
Irvine, CA 92718
(714) 753–5800
FAX: (714) 753–5808

Cypress Semiconductor
12526 High Bluff Dr., Ste. 300
San Diego, CA 92130
(619) 755–1976
FAX: (619) 755–1969

Canada

Cypress Semiconductor
701 Evans Avenue
Suite 312
Toronto, Ontario M9C 1A3
(416) 620–7276
FAX: (416) 620–7279

Colorado

Cypress Semiconductor
4704 Harlan St., Suite 360
Denver, CO 80212
(303) 433–4889
FAX: (303) 433–0398

Florida

Cypress Semiconductor
13535 Feather Sound Drive
Suite 130
Clearwater, FL 34622
(813) 968–1504

Cypress Semiconductor
255 South Orange Avenue
Suite 1255
Orlando, FL 32801
(407) 422–0734
FAX: (407) 422–1976

Cypress Semiconductor
1000 W. McNab Road
Pompano Beach, FL 33069
(954) 943–9295
FAX: (954) 943–4057

Georgia

Cypress Semiconductor
1080 Holcomb Bridge Rd.
Building 200, Ste. 265
Roswell, GA 30076
(770) 998–0491
FAX (770) 998–2172

Illinois

Cypress Semiconductor
1530 E. Dundee Rd., Ste. 190
Palatine, IL 60067
(708) 934–3144
FAX: (708) 934–7364

Maryland

Cypress Semiconductor
8850 Stanford Blvd., Suite 1600
Columbia, MD 21045
(410) 312–2911
FAX: (410) 290–1808

Minnesota

Cypress Semiconductor
14525 Hwy. 7, Ste. 360
Minnetonka, MN 55345
(612) 935–7747
FAX: (612) 935–6982

New Hampshire

Cypress Semiconductor
61 Spit Brook Road, Ste. 550
Nashua, NH 03060
(603) 891–2655
FAX: (603) 891–2676

New Jersey

Cypress Semiconductor
100 Metro Park South
3rd Floor
Laurence Harbor, NJ 08878
(908) 583–9008
FAX (908) 583–8810

New York

Cypress Semiconductor
22 IBM Road
Suite 103B
Poughkeepsie, NY 1260
(914) 463–3218
FAX: (914) 463–3220

North Carolina

Cypress Semiconductor
7500 Six Forks Rd., Suite G
Raleigh, NC 27615
(919) 870–0880
FAX: (919) 870–0881

Oregon

Cypress Semiconductor
8196 S.W. Hall Blvd. Suite 100
Beaverton, OR 97005
(503) 626–6622
FAX: (503) 626–6688

Pennsylvania

Cypress Semiconductor
Two Neshaminy Interplex, Ste. 206
Trevose, PA 19053
(215) 639–6663
FAX: (215) 639–9024

Texas

Cypress Semiconductor
101 W. Renner Rd, Suite 155
Richardson, TX 75082–2002
(214) 437–0496
FAX: (214) 644–4839

Cypress Semiconductor
8834 Capital of Texas Highway North
Suite 220
Austin, TX 78759
(512) 418–4205
FAX: (512) 418–4201

Cypress Semiconductor
20405 SH 249, Ste. 215
Houston, TX 77070
(713) 370–0221
FAX: (713) 370–0222

Domestic Sales Representatives

Alabama

Giesting & Associates
4835 University Square
Suite 15
Huntsville, AL 35816
(205) 830–4554
FAX: (205) 830–4699

Arizona

Thom Luke Sales, Inc.
9700 North 91st St., Suite A–200
Scottsdale, AZ 85258
(602) 451–5400
FAX: (602) 451–0172

California

TAARCOM
451 N. Shoreline Blvd.
Mountain View, CA 94043
(415) 960–1550
FAX: (415) 960–1999

TAARCOM
735 Sunrise Ave., Suite 200–4
Roseville, CA 95661
(916) 782–1776
FAX: (916) 782–1786

Technology Solutions Company
5525 Oakdale Ave., Suite 275
Woodland Hills, CA 91364
(818) 704–1693
FAX: (818) 704–6165

Technology Solutions Company
10 Hughes, Suite A201
Irvine, CA 92718
(714) 707–4565
FAX: (714) 707–4510

Canada

bbd Electronics, Inc.
6685–1 Millcreek Dr.
Mississauga, Ontario L5N 5M5
(905) 821–7800
FAX: (905) 821–4541

bbd Electronics, Inc.
298 Lakeshore Rd., Ste. 203
Pointe Claire, Quebec H9S 4L3
(514) 697–0801
FAX: (514) 697–0277

bbd Electronics, Inc. — Ottawa
(613) 564–0014
FAX: (416) 821–4092

bbd Electronics, Inc. — Winnipeg
(204) 942–2977
FAX: (416) 821–4092

Western Canada

Microwe Electronics Corporation
Site #7, Box 40 R.R.1
Dewinton, Alberta, Canada T0L 0X0
(403) 254–4180
FAX: (403) 256–0942

Colorado

Lange Sales
1500 W. Canal Court, Bldg. A
Suite 100
Littleton, CO 80120
(303) 795–3600
FAX: (303) 795–0373

Georgia

Giesting & Associates
2434 Highway 120
Suite 108
Duluth, GA 30155
(770) 476–0025
FAX: (770) 476–2405

Idaho

Sierra Technical Sales
10378 Fairview
Suite 246
Boise, ID 83704
(208) 378–8981
FAX: (208) 378–0228

Illinois

Micro Sales Inc.
901 W. Hawthorn Drive
Itasca, IL 60143
(708) 285–1000
FAX: (708) 285–1008

Indiana

Technology Mktg. Corp.
1526 East Greyhound Pass
Carmel, IN 46032
(317) 844–8462
FAX: (317) 573–5472

Technology Mktg. Corp.
4630–10 W. Jefferson Blvd.
Ft. Wayne, IN 46804
(219) 432–5553
FAX: (219) 432–5555

Technology Marketing Corp.
1214 Appletree Lane
Kokomo, IN 46902
(317) 459–5152
FAX: (317) 457–3822

Iowa

Midwest Technical Sales
463 Northland Ave., N.E.
Suite 101
Cedar Rapids, IA 52402
(319) 377–1688
FAX: (319) 377–2029

Kansas

Midwest Technical Sales
13 Woodland Dr.
Augusta, KS 67010
(316) 775–2565
FAX: (316) 775–3577

Midwest Technical Sales
10,000 College Blvd.
Suite 240
Overland Park, KS 66210
(913) 338–2400
FAX: (913) 338–0404

Kentucky

Technology Marketing Corp.
100 Trade Street, Suite 1A
Lexington, KY 40510–1007
(606) 253–1808
FAX: (606) 253–1662

Maryland

Tri-Mark, Inc.
1410 Crain Highway, N.W.
Suite 4B
Glen Burnie, MD 21061
(410) 761–6000
FAX: (410) 761–6006

Massachusetts

The Nashoba Group
321 Billerica Rd.
Chelmsford, MA 01824
(508) 256–9900
FAX: (508) 256–1142

Mexico

Ciber Electronica, S.A. de C.V.
Prolongacion Arbol No. 33
Col. Chapalita Sur
45000 Guadalajara, Jal.
Mexico
Tel: (52) 3–647–5217
Tel: (52) 3–647–1998
FAX: (52) 3–121–3331

Ciber Electronica, S.A. de C.V.
Monrovia No. 410
Col. Portales
03300 Mexico, D.F.
Tel & FAX: (52) 5–539–7832

Ciber Electronica, S.A. de C.V.
Missouri No. 202 OTE.
Col. del Valle
66220 Garza Garcia, N.L.
Mexico
Tel & FAX: (52) 8–356–842

Michigan

Techrep
2200 North Canton Center Rd.
Suite 110
Canton, MI 48187
(313) 981–1950
FAX: (313) 981–2006

Minnesota

Matrix Marketing, Inc.
5001 West 80th Street, Suite 375
Bloomington, MN 55437
(612) 835–6977
FAX: (612) 835–6822

Missouri

Midwest Technical Sales
4203 Earth City Expwy., #149
Earth City, MO 63045
(314) 298–8787
FAX: (314) 298–9843

Nevada

TAARCOM
735 Sunrise Ave.
Suite 200–4
Roseville, CA 95661
(916) 782–1776
FAX: (916) 782–1786

Domestic Sales Representatives (continued)

New Jersey

GroupTec
111 Howard Blvd.
Suite 212
Mt. Arlington, NJ 07856
(201) 398–1200
FAX: (201) 398–3344

New Mexico

Thom Luke Sales
(719) 661–8795
FAX: (602) 451–0172

New York

Reagan/Compar
815 Montrose Turnpike
Owego, NY 13827
(716) 271–2230
FAX: (716) 381–2840

Reagan/Compar
44 Riverferry Way
Rochester, NY 14608
(716) 454–3350
FAX: (716) 454–4230

Reagan/Compar
532 Benton Street
Rochester, NY 14620
(716) 473–6070
FAX: (716) 473–6075

Reagan/Compar
3301 Country Club Road
Ste. 2211
P.O. Box 135
Endwell, NY 13760
(607) 754–2171
FAX: (607) 754–4270

North Carolina

Quantum Marketing
6604 Six Forks Rd., Ste. 102
Raleigh, NC 27615
(919) 846–5728
FAX: (919) 847–8271

Quantum Marketing
4801 E. Independent Blvd.
Ste. 1000
Charlotte, NC 28212
(704) 536–8558
FAX: (704) 536–8768

Ohio

KW Electronic Sales, Inc.
8514 North Main Street
Dayton, OH 45415
(513) 890–2150
FAX: (513) 890–5408

KW Electronic Sales, Inc.
3645 Warrensville Center Rd. #244
Shaker Heights, OH 44122
(216) 491–9177
FAX: (216) 491–9102

Oregon

Northwest Marketing Associates
4905 SW Griffith Drive
Suite 106
Beaverton, OR 97005
(503) 644–4840
FAX: (503) 644–9519

Pennsylvania

KW Electronic Sales, Inc.
4068 Mt. Royal Blvd., Ste. 110
Allison Park, PA 15101
(412) 492–0777
FAX: (412) 492–0780

Omega Electronic Sales, Inc.
Four Neshaminy Interplex, Ste. 101
Trevose, PA 19053
(215) 244–4000
FAX: 244–4104

Puerto Rico

Electronic Technical Sales
P.O. Box 10758
Caparra Heights Station
San Juan, P.R. 00922
(809) 781–1313
FAX: (809) 781–2020

Tenessee

Giesting & Associates
475 Arrowhead Springs Lane
Versailles, KY 40383
(606) 873–2330

Utah

Sierra Technical Sales
1192 E. Draper Parkway
Suite 103
Draper, UT 84020
(801) 571–8195
FAX: (801) 571–8194

Washington

Northwest Marketing Associates
12835 Bellevue-Redmond, Ste. 330N
Bellevue, WA 98005
(206) 455–5846
FAX: (206) 451–1130

Wisconsin

Micro Sales Inc.
210 Regency Court
Suite 100
Brookfield, WI 53045
(414) 786–1403
FAX: (414) 786–1813

International Direct Sales Offices

Cypress Semiconductor International—Europe

Avenue Ernest Solvay, 7
B–1310 La Hulpe, Belgium
Tel: (32) 2–652–0270
Telex: 64677 CYPINT B
FAX: (32) 2–652–1504

France

Cypress Semiconductor France
Miniparc Bât. no 8
Avenue des Andes, 6
Z.A. de Courtaboeuf
91952 Les Ulis Cedex, France
Tel: (33) 1–69–29–88–90
FAX: (33) 1–69–07–55–71

Germany

Cypress Semiconductor GmbH
Munchner Str. 15A
W-8011, Zorneding, Germany
Tel: (49) 81–06–2855
FAX: (49) 81–06–20087

Cypress Semiconductor GmbH
Büro Nord
Matthias-Claudius-Str. 17
W–2359 Henstedt-Ulzburg, Germany
Tel: (49) 4193-77217
FAX: (49) 4193-78259

Italy

Cypress Semiconductor
Interporto di Torino
Prima Strada n. 5/B
10043 Orbassano, Italy
Tel: (39) 11–397–57–98
or (39) 11–397–57–57
FAX: (39) 11–397–58–10

Cypress Semiconductor
Via Gallarana 4
20052 Monza, Milano
Tel: (39) 39–202–7099
FAX: (39) 202–7101

Japan

Cypress Semiconductor Japan K.K.
Shinjuku-Marune Bldg.
1–23–1 Shinjuku
Shinjuku-ku, Tokyo, Japan 160
Tel: (81) 3–5269–0781
FAX: (81) 3–5269–0788

Singapore

Cypress Semiconductor Singapore
583 Orchard Road, #11–03 Forum
Singapore 0923
Tel: (65) 735–0338
FAX: (65) 735–0228

Sweden

Cypress Semiconductor Scandinavia AB
Täby Centrum, Ingang S
S–18311 Täby, Sweden
Tel: (46) 8 638 0100
FAX: (46) 8 792 1560

Taiwan, R.O.C.

Cypress Semiconductor Taiwan
11F, RM 1102, No. 333
Section 1, Keelung Rd.,
Taipei, Taiwan, R.O.C.
Tel: (886) 2–757–6898
FAX: (886) 2–757–6892

United Kingdom

Cypress Semiconductor U.K., Ltd.
Gate House
Fretherne Road
Welwyn Garden City
Herts., U.K. AL8 6NS
Tel: (44) 707–33–88–88
FAX: (44) 707–33–88–11

Cypress Semiconductor Manchester
27 Saville Rd. Cheadle
Gatley, Cheshire, U.K.
Tel: (44) 614–28–22–08
FAX: (44) 614–28–0746

International Sales Representatives

Australia

Braemac Pty. Ltd.
1/59–61 Burrows Road
Alexandria, Sydney 2015, Australia
Tel: (61) 2–550–6600
FAX: (61) 2–550–6377

Braemac Pty. Ltd.
6/417 Ferntree Gully Rd.
Mt. Waverly, Victoria 3149, Australia
Tel: (61) 3–540–0100
FAX: (61) 3–540–0122

Braemac Pty. Ltd.
300 Gilles Street
Adelaide, SA 5000, Australia
Tel: (61) 8–232–5550
FAX: (61) 8–232–5551

Braemac Pty Ltd.
345 Harborne Street
Herdsman W.A. 6017, Australia
Tel: (61) 9–443–5122
FAX: (61) 9–443–5262

Austria

Eurodis Electronics GmbH
Lamenzanstrasse 10
A-1232 Wien
Austria
Tel: (43) 1–610–62–128
FAX: (43) 1–610–62–151

Belgium

N.V. Memec Benelux
Sint-Lambertusstraat 135
1200 Brussels, Belgium
Tel: (32) 2–772–8008
FAX: (32) 2–460–1200

Belgium (continued)

Sonetech
Limburgstriumlaan 243, B–2
B–1810 Wemmel, Belgium
Tel: (32) 2–460–0707
FAX: (32) 2–460–1200

Denmark

Tech-Partner A/S
Tomsagervej 18
DK–8250 Aabyhoj (Aarhus)
Denmark
Tel: (45) 86–25–00–55
FAX: (45) 86–25–28–55

Team Tech
Bygstubben 3
DK–2950 Vedbaek
Denmark
Tel: (45) 45–66–25–00
FAX: (45) 45–66–02–44

Finland

ScandComp Finland OY
Asemakuja 2 A
SF-02 770 Espoo
Finland
Tel: (358) 0 61352695
FAX: (358) 0 61352620

France

Arrow Electronics
73/79, Rue des Solets
Silic 585
94653 Rungis Cedex
Tel: (33) 1 49 78 49 00
FAX: (33) 1 49 78 05 99

France (continued)

Arrow Electronics
Les Jardins d'Entreprises
Betiment B3
213, Rue Gerland
69007 Lyon
Tel: (33) 78 72 79 42
FAX: (33) 78 72 80 24

Arrow Electronics
Centreda
Avenue Didier Daurat
31700 Blagnac
Tel: (33) 61 15 75 18
FAX: (33) 61 30 01 93

Arrow Electronics
Immeuble St. Christophe
Rue de la Frebardiere
Zi Sud Est
35135 Chantepie
Tel: (33) 99 41 70 44
FAX: (33) 99 50 11 28

Newtek
Rue de L'Esterel, 8, Silic 583
F–94663 Rungis Cedex, France
Tel: (33) 1–46–87–22–00
FAX: (33) 1–46–87–80–49

Newtek
Rue de l'Europe, 4
Zac Font–Ratel
F–38640 Claix, France
Tel: (33) 76–98–56–01
FAX: (33) 76–98–16–04

Scaib, SA
6 Rue Ambroise Croizat
91127 Palaiseau Cedex, France
Tel: (33) 1–69–19–89–00
FAX: (33) 1–69–19–89–20

International Sales Representatives (continued)

Germany

AktiveRep Electronic GmbH
Kennedy Strasse 5
D–75438 Knittlingen, Germany
Tel: (49) 70–43–94 00 12
FAX: (40) 70–43–334 92

CED Ditronic GmbH
Feldkirchner Str. 12A
D–85551 Kirchheim, Germany
Tel: (49) 89–903 8551
FAX: (49) 89–903 0944

CED Ditronic GmbH
Julius-Hoelder Str. 42
D–70597 Stuttgart, Germany
Tel: (49) 711–72001–0
FAX: (49) 711–7289780

CED Ditronic GmbH
Laatzener–Str. 19
D–30539 Hannover, Germany
Tel: (49) 511–8764–0
FAX: (49) 511–8764–160

Metronik GmbH
Leonhardsweg 2, Postfach 1328
D–82008 Unterhaching, Germany
Tel: (49) 89–61108–0
FAX: (49) 89–6116468

Metronik GmbH
Liessauer Pfad 17
D–13503 Berlin, Germany
Tel: (49) 30–4361219
FAX: (49) 30–4315956

Metronik GmbH
Zum Lonnenhohl 38
D–44319 Dortmund, Germany
Tel: (49) 231–217041
FAX: (49) 231–210799

Metronik GmbH
Osmiastrasse 9
D–69221 Dossenhem, Germany
Tel: (49) 6203–4701
FAX: (49) 6203–45543

Metronik GmbH
Schoenauer Str. 113
D–04207 Leipzig, Germany
Tel: (49) 341–4891413
FAX: (49) 341–4891424

Metronik GmbH
Pilotystrasse 27/29
D–90408 Nürnberg, Germany
Tel: (49) 911–363536
FAX: (49) 911–353986

Germany (continued)

Metronik GmbH
Carl Zeiss-Strasse 37
D–25451 Quickborn, Germany
Tel: (49) 41–06–77 30 50
FAX: (49) 41–06–77 30 52

Metronik GmbH
Löewenstrasse 37
D–70597 Stuttgart, Germany
Tel: (49) 711–764033
FAX: (49) 711–7655181

Metronik GmbH
Bahnstrasse 9
D–65205 Wiesbaden, Germany
Tel: (49) 611–70 20 83
FAX: (49) 611–70 28 86

SASCO–HED GmbH
Hermann-Oberth-Strasse 16
D–85640 Putzbrunn, Germany
Tel: (49) 89–4611–211
FAX: (49) 89–4611–271

SASCO–HED GmbH
Huttenstrasse 31
D–10552 Berlin, Germany
Tel: (49) 30–349–92 40
FAX: (49) 30–349–52 36

SASCO–HED GmbH
Beratgerstr. 36
D–44149 Dortmund, Germany
Tel: (49) 231–17 97 91
FAX: (49) 231–17 29 91

SASCO–HED GmbH
Hainer Weg 48
D–60599 Frankfurt, Germany
Tel: (49) 69–61 03 91
FAX: (49) 69–61 88 24

SASCO–HED GmbH
Europaallee 3
D–22850 Norderstedt, Germany
Tel: (49) 4052–3 20 13
FAX: (49) 4052–3 23 78

SASCO–HED GmbH
Stafflenbergstrasse 21
D–70184 Stuttgart, Germany
Tel: (49) 711–21 07 10
FAX: (49) 711–23 39 63

SASCO–HED GmbH
Am Gansacker 26
D–79224 Umkirch bei Freiburg
Germany
Tel: (49) 7665–70 18
FAX: (49) 7665–87 78

Greece

Peter Caritato & Associates S.A.
31 Ilia Iliou
Athens 11743, Greece
Tel: (30) 1–9020–115
FAX: (30) 1–9017–024

Hong Kong

Tekcomp Electronics, Ltd.
Rm. 913–914 Bank Centre
636, Nathan Road, Mongkok
Kowloon, Hong Kong
Tel: (852) 2–710–8121
Telex: 38513 TEKHL
FAX: (852) 2–710–9220

India

Spectra Innovations Inc.
Manipal Centre, Unit No. S–822
47, Dickenson Rd.
Bangalore–560,042
Karnataka, India
Tel: (91) 80–558–8323/3977
FAX: (91) 80–558–6872

Israel

Talviton Electronics
P.O. Box 21104, 9 Biltmore Street
Tel Aviv 61 210, Israel
Tel: (972) 3–544–2430
Telex: 33400 VITKO
FAX: (972) 3–544–2085

Italy

Silverstar Ltd. SPA
Viale Fulvio Testi, 280
20126 Milano, Italy
Tel: (39) 2 661251
Telex: 33 2189 SIL 7I
FAX: (39) 2 66101359

CED Italy
Via Volta 54
20090 Cusago (MI)
Italy
Tel: (39) 2 9039 0684

ECC Electronica S.P.A.
Via C. Goldoni 29
20090 Trezzano Sul Naviglo (Milano)
Italy
Tel: (39) 2 48401547
FAX: (39) 2 48401599

International Sales Representatives (continued)

Japan

Tomen Electronics Corp.
2-1-1 Uchisaiwai-cho, Chiyoda-ku
Tokyo, 100 Japan
Tel: (81) 3-3506-3673
Telex: 23548 TMELCA
FAX: (81) 3-3506-3497

Fuji Electronics Co., Ltd.
Ochanomizu Center Bldg.
3-2-12 Hongo, Bunkyo-ku
Tokyo, 113 Japan
Tel: (81) 3-3814-1416
Telex: J28603 FUJITRON
FAX: (81) 3-3814-1414

Ryoyo Electro Corporation
Konwa Bldg., 1-12-22 Tsukiji,
Chuo-ku, Tokyo 104 Japan
Tel: (81) 3-3546-5088
FAX: (81) 3-3546-5044

Korea

Logicom Inc.
5th Floor, Haesung Bldg.
2-46 Yangjae-Dong
Seocho-ku
Seoul, Korea 137-131
Tel: (82) 2-575-3211
FAX: (82) 2-576-7040

Netherlands

Memec Benelux B.V.
Insulindelaan 134
5613 BT Eindhoven
The Netherlands
Tel: (31) 40-65-9399
FAX: (31) 40 65-9393

Sonetec Nederland B.V.
Gulberg 33
NL-5674 TE Nuenen
The Netherlands
Tel: (31) 40-2-635-635
Telex: 59418 INTRA NL
FAX: (31) 40-2-832-300

Norway

ScandComp Norway A/S
PO Box 274
N-2020 Skedsmokorset
Norway
Tel: (47) 22-50-06-50
FAX: (47) 22-50-27-77

Portugal

ATD Electronica S.A.
Quinta Grande Lote 20
Alfragide
2700 Amadora (Lisboa)
Portugal
Tel: (351) 1-4714182
FAX: (351) 1-4715886

Singapore

Electec PTE Ltd.
Block 50, Kallang Bahru
#04-21, Singapore 1233
Tel: (65) 294-8389
FAX: (65) 294-7623

South Africa

Electronic Bldg. Elements
P.O. Box 912-1222
Silverton 0127
178 Erasmus St., Meyers Park
Pretoria 0184, South Africa
Tel: (27) 12 803-8294
FAX: (27) 12 803-7680

Spain

ATD Electronica S.A.
Albasanz, 75
28037 Madrid, Spain
Tel: (34) 1-304-1534
FAX: (34) 1-327-2778

ATD Electronica S.A.
Conchita Suprevia 9
08028 Barcelona, Spain
Tel: (34) 3-4907344
FAX: (34) 3-4901723

SELCO
Ctra. de La Coruna, Km 18.200
28230 Las Rozas (Madrid), Spain
Tel: (34) 1-637-1333
FAX: (34) 1-637-5114

Sweden

ScandComp Sweden AB
Box 8303 Domnarvsgatan 33
S-163 08 Spanga
Sweden
Tel: (46) 8-761-73-00
FAX: (46) 8-760-46-69

Switzerland

Basix für Elektronik A. G.
Hardturmstrasse 181
CH-8010 Zurich, Switzerland
Tel: (41) 1-276-11-11
Telex: 822762 BAEZ CH
FAX: (41) 1-276-14-48

Taiwan R.O.C.

Prospect Technology Corp.
5F, No. 348, Section 7
Cheng-Teh Rd.
Taipei, Taiwan
Tel: (886) 2-820-5353
Telex: 14391 PROSTECH
FAX: (886) 2-820-5731

Turkey

Inter Electronik Sanayi ve Ticaret A.S.
Kadlkoy Hasircibasi Caddesi no. 55
81310 Istanbul
Turkey
Tel: (0216) 349-94-00
Telex: 29245 Inmd tr
FAX: (0216) 349-94-30
FAX: (0216) 349-94-34

United Kingdom

2001 Electronic Components Ltd.
Stevenage Business Park
Pin Green
Stevenage, Herts
SG1 4SU U. K.

Ambar Components Ltd.
17 Thame Park Road
Thame, Oxfordshire
England, OX9 3XD
Tel: (44) 844-26-11-44
Telex: 837427
FAX: (44) 844-26-17-89

Arrow Electronics (UK) Ltd.
St. Martins Business Centre
Cambridge Road
Bedford MK42 OLF, U.K.
Tel: (44) 234 270272
FAX: (44) 234 214674

Pronto Electronic System Ltd.
City Gate House
Eastern Avenue, 399-425
Gants Hill, Ilford,
Essex, U. K. IG2 6LR
Tel: (44) 81-554-62-22
Telex: 8954213 PRONTO G
FAX: (44) 81-518-32-22

Spectrum
2 Grange Mews
Station Road
Launton
Bicester
Oxon, U.K. OX6 0DX
Tel: (44) 1-869-325-174
FAX: (44) 1-869-325-175

Distributors

Anthem Electronics, Inc.:

Huntsville, AL 35805
(205) 890–0302

Tempe, AZ 85281
(602) 966–6600

Chatsworth, CA 91311
(818) 775–1333

Irvine, CA 92718
(714) 768–4444

Rocklin, CA 95677
(916) 624–9744

San Jose, CA 95131
(408) 453–1200

San Diego, CA 92121
(619) 453–9005

Englewood, CO 80112
(303) 790–4500

Waterbury, CT 06705
(203) 575–1575

Altamonte Springs, FL 32701
(407) 831–0007

Fort Lauderdale, FL 33309
(305) 484–0990

Duluth, GA 30136
(404) 931–3900

Schaumburg, IL 60173
(708) 884–0200

Wilmington, MA 01887
(508) 657–5170

Columbia, MD 21046
(301) 995–6640

Eden Prairie, MN 55344
(612) 944–5454

Pine Brook, NJ 07058
(201) 227–7960

Commack, NY 11725
(516) 864–6600

Raleigh, NC 27604
(919) 871–6200

Beaverton, OR 97005
(503) 643–1114

Horsham, PA 19044
(215) 443–5150

Austin, TX 78728
(512) 388–0049

Richardson, TX 75081
(214) 238–7100

Salt Lake City, UT 84119
(801) 973–8555

Bothel, WA 98011
(206) 483–1700

Arrow Electronics:

Alabama
Huntsville, AL 35816
(205) 837–6955

Arizona
Tempe, AZ 85282
(602) 431–0030

California
Calabasas, CA 91302
(818) 880–9686

Irvine, CA 92718
(714) 587–0404

San Diego, CA 92123
(619) 565–4800

San Jose, CA 95131
(408) 441–9700

San Jose, CA 95134

Canada
Mississauga, Ontario L5T 1MA
(416) 670–7769

Dorval, Quebec H9P 2T5
(514) 421–7411

Neapean, Ontario K2E 7W5
(613) 226–6903

Quebec City, Quebec G2E 5RN
(418) 871–7500

Burnaby, British Columbia V5A 4T8
(604) 421–2333

Colorado
Englewood, CO 80112
(303) 799–0258

Connecticut
Wallingford, CT 06492
(203) 265–7741

Florida
Deerfield Beach, FL 33441
(305) 429–8200

Lake Mary, FL 32746
(407) 333–9300

Georgia
Deluth, GA 30071
(404) 497–1300

Illinois
Itasca, IL 60143
(708) 250–0500

Indiana
Indianapolis, IN 46268
(317) 299–2071

Kansas
Lenexa, KS 66214
(913) 541–9542

Maryland
Columbia, MD 21046
(410) 596–7800

Gathersburg, MD
(301) 596–7800

Arrow Electronics: (cont.)

Massachusetts
Wilmington, MA 01887
(617) 658–0900

Michigan
Livonia, MI 48152
(313) 462–2290

Minnesota
Eden Prairie, MS 55344
(612) 941–5280

Missouri
St. Louis, MO 63146
(314) 567–6888

New Jersey
Marlton, NJ 08053
(609) 596–8000

Pinebrook, NJ 07058
(201) 227–7880

New York
Rochester, NY 14623
(716) 427–0300

Hauppauge, NY 11788
(516) 231–1000

North Carolina
Raleigh, NC 27604
(919) 876–3132

Ohio
Centerville, OH 45458
(513) 435–5563

Solon, OH 44139
(216) 248–3990

Oklahoma
Tulsa, OK 74146
(918) 252–7537

Oregon
Beaverton, OR 97006–7312
(503) 629–8090

Pennsylvania
Pittsburgh, PA 15238
(412) 963–6807

Texas
Austin, TX 78758
(512) 835–4180

Carrollton, TX 75006
(214) 380–6464

Houston, TX 77099
(713) 530–4700

Washington
Bellevue, WA 98007
(206) 643–9992

Wisconsin
Brookfield, WI 53045
(414) 792–0150

Distributors (continued)

Axis Components

Corporate Headquarters
SanDiego, CA 92121
(619) 677–7950
(800) 556–0225

Irvine, CA 92714
(714) 442–8325

Westlake Village, CA 91362
(818) 706–0166

Sunnyvale, CA 94086
(408) 522–9599

Westminster, CO 80234
(303) 469–8186

Bell Microproducts:

Irvine, CA 92718
(714) 470–2900

San Jose, CA 94131
(408) 451–9400

Altamonte Springs, FL 32714
(407) 682–1199

Deerfield Beach, FL 33441
(305) 429–1001

Billerica, MA 01882
(508) 667–2400

Columbia, MD 21045
(410) 720–5100

Edina, MN 55435
(612) 933–3236

Clifton, NJ 07013
(201) 777–4100

Smithtown, NY 11787
(516) 543–2000

Ambler, PA 19002
(215) 540–4148

Austin, TX 78759
(512) 258–0725

Richardson, TX 75081
(214) 783–4191

Chantilly, VA 22021
(703) 803–1020

Redmond, WA 98052
(206) 861–7510

Marshall Industries:

Alabama
Huntsville, AL 35801
(205) 881–9235

Arizona
Phoenix, AZ 85044
(602) 496–0290

California
Marshall Industries, Corp. Headquarters
El Monte, CA 91731–3004
(818) 307–6000

Irvine, CA 92718
(714) 458–5301

Calabasas, CA 91302
(818) 878–7000

Rancho Cordova, CA 95670
(916) 635–9700

San Diego, CA 92123
(619) 627–4140

Milpitas, CA 95035
(408) 942–4600

Canada
Mississauga, Ontario L4V 1X5
(416) 458–8046

Pointe Claire, Quebec H9R 5P9
(514) 694–8142

Colorado
Colorado Springs, CO 80915
(719) 573–0904

Thornton, CO 80241
(303) 451–8383

Connecticut
Wallingford, CT 06492–0200
(203) 265–3822

Florida
Ft. Lauderdale, FL 33309
(305) 977–4880

Florida (continued)
Altamonte Springs, FL 32701
(407) 767–8585

St. Petersburg, FL 33716
(813) 573–1399

Georgia
Norcross, GA 30093
(404) 923–5750

Illinois
Schaumbrug, IL 60173
(708) 490–0155

Indiana
Carmel, IN 46032
(317) 431–6554

Kansas
Lenexa, KS 66214
(913) 492–3121

Maryland
Columbia, MD 21046
(410) 880–3030

Distributors (continued)

Marshall Industries:

Massachusetts
Wilmington, MA 01887
(508) 658–0810

Michigan
Livonia, MI 48150
(313) 525–5850

Minnesota
Plymouth, MN 55447
(612) 559–2211

Missouri
Bridgeton, MO 63044
(314) 291–4650

New Jersey
Fairfield, NJ 07006
(201) 882–0320

Mt. Laurel, NJ 08054
(609) 234–9100

New York
Endicott, NY 13760
(607) 785–2345

Rochester, NY 14624
(716) 235–7620

Ronkonkoma, NY 11779
(516) 737–9300

North Carolina
Raleigh, NC 27604
(919) 878–9882

Ohio
Solon, OH 44139
(216) 248–1788

Dayton, OH 45414
(513) 898–4480

Oregon
Beaverton, OR 97005
(503) 644–5050

Pennsylvania
Mt. Laurel, NJ 08054
(609) 234–9100

Texas
Austin, TX 78754
(512) 837–1991

Richardson, TX 75081
(214) 705–0600

Houston, TX 77043
(713) 467–1666

Utah
Salt Lake City, UT 84119
(801) 973–2288

Washington
Bothell, WA 98011
(206) 486–5747

Wisconsin
Waukesha, WI 53186
(414) 797–8400

Semad:

Calgary
Calgary, Alberta T2E 7H7
(403) 252–5664
FAX: (800) 565–9779

Montreal
Pointe Claire, Quebec H9R 427
(514) 694–0860
1–800–361–6558
FAX: (514) 694–0965

Ottawa
Ottawa, Ontario K1B 1A7
(613) 526–4866
FAX: (613) 523–4372

Toronto
Markham, Ontario L3R 4Z4
(905) 475–3922
FAX: (905) 475–4158

Vancouver
Burnaby, British Columbia V5G 1H1
(604) 451–3444
1–800–663–8956
FAX: (604) 451–3445

Zeus Electronics:

Yorba Linda, CA 92686
(714) 921–9000

San Jose, CA 95131
(408) 629–4789

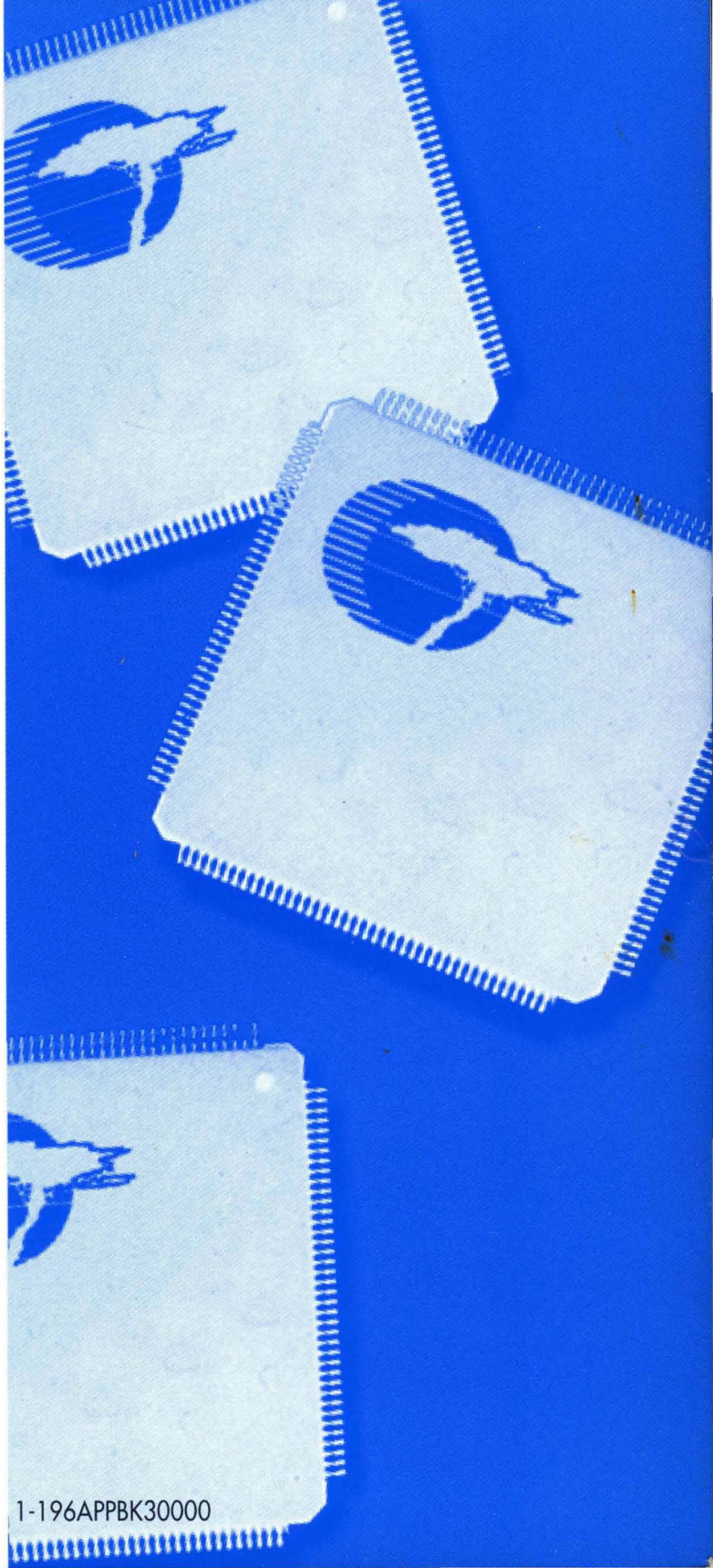
Lake Mary, FL 32746
(407) 333-3055

Itasca, IL 60143
(708) 595–9730

Wilmington, MA 01887
(508) 658–4776

Port Chester, NY 10573
(914) 937–7400

Carrollton, TX 75006
(214) 380–4330



Cypress Semiconductor
3901 North First Street
San Jose, CA 95134
Tel: 1 (800) 293-2311
Fax: (408) 943-2741
Fax on demand: 1 (800) 213-5120
http://www.cypress.com

1-196APPBK30000