Texas Instruments

The proLogic™ Compiler

User's Guide

1989

The proLogic™ Compiler

User's Guide

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) provides this software **AS IS** without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. TI may discontinue, make improvements to and or change the program(s) described herein at any time and without notice. TI does not warrant that this software package will function properly in every software/hardware environment.

Users are authorized to copy, duplicate, and distribute this product for internal use only without further permission from TI provided that the copyright is maintained on each copy. Selling of this product without prior consent in writing from TI is prohibited.

proLogic™ Development Software, Release 1.97

This logic compiler was prepared for distribution by Texas Instruments, Incorporated, expressly to support devices manufactured by them. A complete universal compiler, including Texas Instruments devices, is also available directly from Inlab, Inc. Contact Inlab Customer Service at (303) 460–0103 and ask for information or assistance regarding proLogic.

Assistance on programming Texas Instruments Programmable Logic is also available, upon request, from the nearest TI field sales office, local authorized TI distributor, or by calling the Texas Instruments PLD Hotline (214) 997–5666.

TRADEMARKS

IBM is a registered trademark of International Business Machines Corporation.
IMPACT is a trademark of Texas Instruments Incorporated.
proLogic is a trademark of Inlab Incorporated.

Preface

Read This First

This Preface contains a brief description of the contents of *The proLogic Compiler User's Guide*.

How to Use this Manual

Preface

The proLogic compiler is a software development design tool used to program Texas Instrument Programmable Logic Devices (PLD). This development software package quickly converts your logic design to a JEDEC fuse map that can be downloaded to a device programmer. proLogic allows you to describe your logic design in any of the following formats:

- ❏ Truth Table
- ❏ Boolean Equations
- ❏ State Diagrams

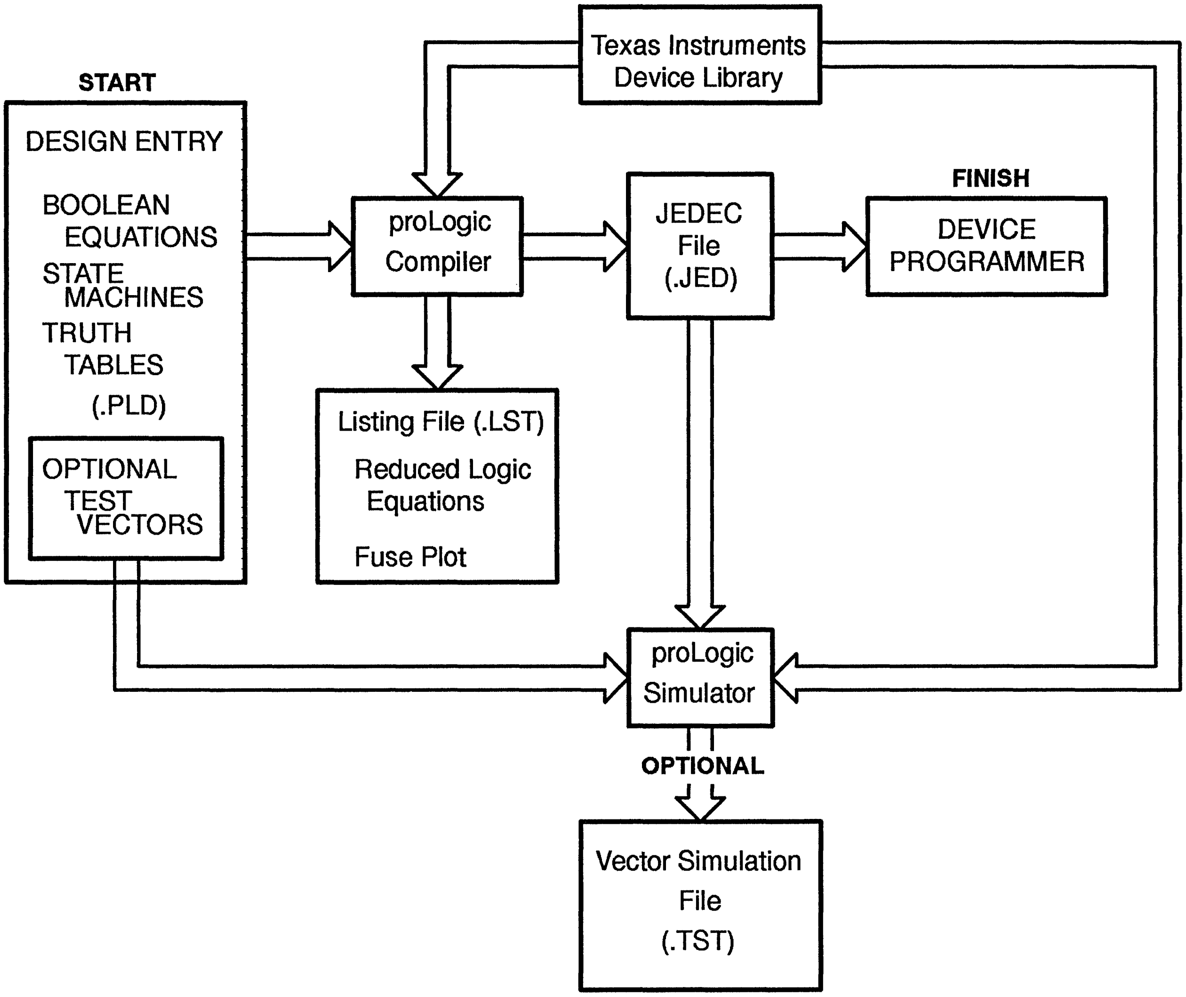
From your logic design, the proLogic compiler will create a standard JEDEC fuse map. The JEDEC file can be downloaded to a device programmer to produce a functional programmed device. Many Texas Instruments authorized distributors provide these programming services.

The proLogic compiler also serves as a functional test vector simulator. The simulator uses the fuse list portion from the JEDEC file to create a functional device model. It can then execute the simulation vectors against this model. The results are automatically placed in a file for evaluation.

The proLogic compiler, when combined with a device programmer, will allow you to create integrated circuits to your own specifications. The advantages to this new design methodology are numerous:

- ❏ Lower chip count
- ❏ Reduced board space
- ❏ Lower power requirements
- ❏ Higher reliability (fewer interconnects)
- ❏ Proprietary design protection (fuse protection)
- ❏ Fewer parts in inventory
- ❏ Greater design flexibility

The proLogic compiler is complete and ready for installation on your IBM® compatible PC. The following block diagram shows the steps from design to proLogic to a programmed device.



proLogic Block Diagram

Section 1

How To Begin

The proLogic Compiler Package contains:

❏ a proLogic Compiler User's Guide

❏ three proLogic Diskettes

The proLogic Diskette contains:

❏ the executable files

❏ a README file

❏ Header files (.H) which apply to multiple Programmable Logic Devices

❏ the Texas Instruments PLD Architecture files

Installation

Step 1: On your IBM or IBM compatible PC, make a new directory on the hard disk and call it PROLOGIC.

Step 2: Change directories into your new PROLOGIC directory and copy all of the files on the diskettes into the new directory.

Step 3: The file, NAND3.PLD is a sample Application PLD (sample program source file) needed to describe the PLDs.

Compile the sample device specification by entering the command.

```
LC NAND3
```

Compiling NAND3 produces the output files NAND3.JED and NAND3.LST. The JEDEC file can be downloaded to your device programmer. The listing file is a fuse plot showing the programmed device.

Step 4: Simulate device programming and testing by entering the command

```
LS NAND3
```

Logic Simulation uses NAND3.JED to produce the output file NAND3.TST. All of these files including the source file, are text files so you can print, type, or edit them to see the results.

The files on the Architecture Description diskettes may all be copied to your PROLOGIC directory if you have enough disk space. If not, you can selectively copy the files needed for your programmable devices.

This is the time to look at the README file on the Installation Diskette. This file contains last–minute information that may not be in the User's Guide.

Section 2

Programmable Logic

A Programmable Logic Device (PLD) is a type of integrated circuit whose function is field–configured. These devices allow you to create integrated circuits to your own specifications at a reasonable cost, thus reducing the chip count.

Programmable Logic Devices have the same basic kinds of digital building blocks that Small Scale Integration (SSI) devices have. The difference between SSI devices and PLDs is primarily the higher levels of integration in PLDs.

There are more input signals per gate. Even a common PLD like the TIBPAL6R4 has 32–input AND gates.

There are more gates per device. The 16R4 has sixty–four 32–input AND gates. The size of these devices has required a new form of logic notation. This notation permits the semiconductor manufacturer to specify the function of a PLD in a concise, tabular format.

Conventional schematics tend to flow from left to right, with input signals on the left and output signals on the right. Thus a three input AND gate is drawn.

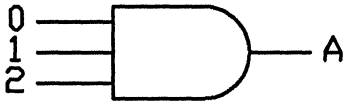


Figure 1. SSI Three Input AND Gate

The new PLD logic notation retains the convention that output signals are on the right. The change is that the input signals flow into a gate vertically from the top or bottom. A PLD AND gate is drawn

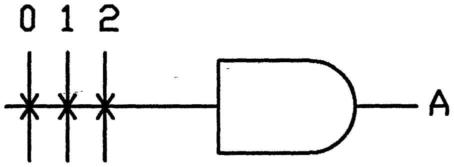


Figure 2. PLD Three Input AND Gate

These PLD AND gates are also called product terms. A product term refers to any n–input AND gate. The main reason for vertical inputs is that PLDs have a very regular structure. In the 16R4, all 32 of the input signals are input to each of the 64 AND gates. This permits the whole logic structure of the device to be concisely noted in tabular format.

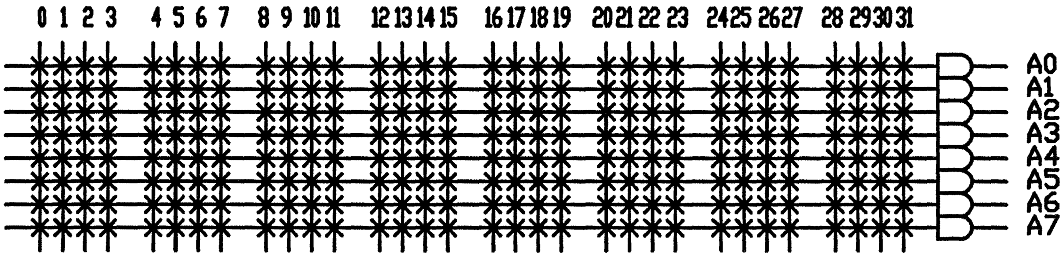


Figure 3. Eight 32-Input AND Gates

The 16R4 PLD is a 20 pin device. When the power, ground, clock, and output enable pins are substracted, the total I/O pin count is 16. Figure 4 shows how the sixty–four 32–input AND gates are interfaced to the I/O pins.

Figure 4 shows that the device in its unprogrammed state doesn't do anything. The output of all 64 AND gates is logic LOW because the input buffers feed both the true and the complement of all inputs into all the AND gates. In Boolean Logic notation:

```
a & !a = 0
```

for all AND gate output.

The basic difference between building a circuit using SSI devices and building a circuit with PLDs is the way in which the gates form a circuit. With SSI devices the gates are connected with wires or traces. Within a PLD, the gates are connected by a process called programming.

A circuit is built within a PLD by disconnecting inputs from gates. In this respect, building a PLD circuit is the exact opposite of building an SSI circuit. A PLD in its initial state has all possible gate connections already made. Programming consists of removing the connections that are not needed for your design. The connections that remain define function of the programmed device .

The notation used to show a programmed device is to draw Xs where connections remain, and draw nothing where connections have been removed. (In most programmed PLDs the number of connections remaining after programming is a lot less than the ones removed, so this convention results in a less cluttered diagram.) To further reduce the number of Xs, a gate with all connections intact is drawn with an X in the gate symbol itself rather than drawing individual Xs at each gate input.

Semiconductor manufacturers ignore this convention when printing the logic diagrams in PLD data books. It is assumed you know that an X is implied at all intersections. Figure 5 shows the part of a 16R4 which has been programmed to implement a 3–input NAND gate. The output is on pin19. The inputs are on pin2, pin3, and pin4.

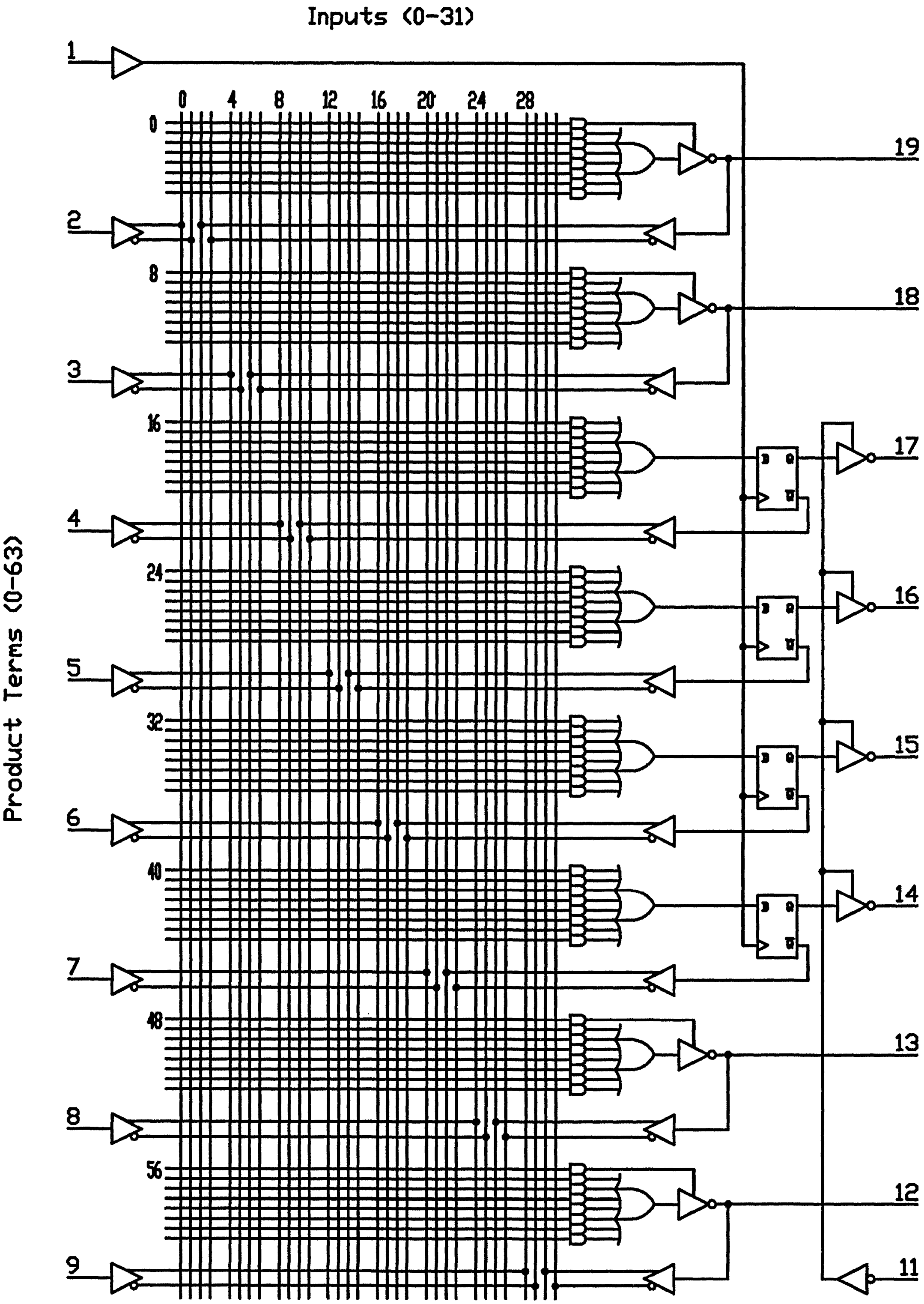


Figure 4. 16R4 Logic Diagram

Figure 5 also illustrates one other PLD feature which you should know. The PLD gates are implemented so that when all gate inputs are disconnected, the gate output is asserted. Figure 5 shows the output buffer for pin 19 as always enabled.

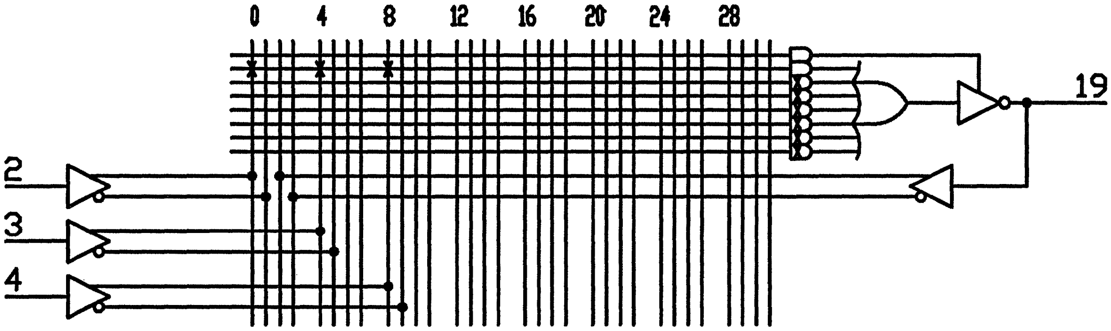


Figure 5. A 3-input NAND Gate

Section 3

PLD Design Implementation

Implementing a digital logic design using PLDs is about the same as implementing a design with SSI devices except you need a development tool called a device programmer to remove the unwanted connections from the internal gates of the PLD.

Device Programmers

A device programmer is an electronic machine which programs the specified cells of a programmable device. PLDs are available in many technologies. A programming algorithm is defined for each type of device. These algorithms involve voltages and currents not used during normal device operation. As might be expected, the number of different programming algorithms is relatively large because algorithms can vary by both product technology and manufacturer.

Texas Instruments continually evaluates new programming equipment. TI Programmable Logic data books are a good reference source for device programmer vendors.

JEDEC Files

The device programmer vendors have made it as easy as possible for you to specify your requirements. All device programmers accept an input file in a standard format. This file is called a JEDEC file because its format was developed by the Joint Electron Device Engineering Council (JEDEC). JEDEC files, being comprised of ASCII characters, are also readable as text files. This example of a fuse list line of a JEDEC–standard file

```
L040 10011001010101101111*
```

is interpreted by the device programmer to mean:

1. program cell 40. The string of 1s and 0s which follow the L040 define the value of consecutive cells after programming. 1 means program the cell. The L040 gives the decimal address of the first cell in the string defined by the line.

2. do not program cell 41. The second digit in the string is a 0 which means do not program the cell.

3. program cells 43, 44, 47, 49, 51, 53, 54 and 56 through 59.

Another kind of JEDEC–standard line is this example of a test vector:

```
V01 NNN010110N10LLLHHNNN*
```

These lines are used by the device programmer to functionally test the programmed device. The V01 is a sequence number. The other characters are called test conditions. The first applies to device pin 1, the second to pin2, and so forth. This sample shows a 20–pin device. Test condition N means do nothing, so the first 0 means apply logic low voltage to pin 4. 1 applies logic high and L and H test for output level low and high.

The proLogic Compiler

The proLogic Compiler is an interface to a device programmer. It accepts your specifications in symbolic form, manages the task of specifying each of the thousands of cell states, and produces a JEDEC file to instruct the device programmer.

To see how it works, let's take the example at the end of Section 2. It shows a 16R4 programmed to implement a 3–input NAND gate. The output is to be pin19. The inputs are on pin2, pin3 and pin4. To make the JEDEC file control the device programmer, we must prepare a PLD specification file.

The PLD specification is a simple text file which tells proLogic which cells to program in the PLD. proLogic takes its input from a source file in the same way that a compiler for microprocessors does. In order to program a 16R4 so that pin19 is a 3–input NAND gate, we need a source file called NAND3.PLD which has the following three lines:

```
include p16r4;
!pin19 = pin2 & pin3 & pin4;
pin19.oe = 1;
```

The first line specifies the PLD to be programmed. The other lines specify the two signals required to implement the circuit.

Once NAND3.PLD is available, entering the DOS command

```
LC NAND3
```

executes the proLogic Compiler. The execution result is a JEDEC file named NAND3.JED which specifies the 61 unwanted connections to be programed by the device programmer. The following is the part of the file which has the fuse list lines.

```
proLogic Compiler (JEDEC Object A100) V2.00
Serial bxav0
Copyright (C) 1988 INLAB, Inc.

p16r4 revision 89.2.11

*N_csidp16r4
*QP20
*QF2048
*F0
*L0000 11111111111111111111111111111111
*L0032 01110111011111111111111111111111
*C07E6
```

While preparing the JEDEC file, the compiler may find errors in your program. When it does, it describes what seems to be wrong on the computer display. The error text completely describes the problem, so there is no need for you to look up error codes in the manual. Another file called NAND.LST is also created which allows you to read the NAND3.JED file. The part of the file which describes pin19 of the 16R4 looks like this:

```
     proLogic Compiler (Fuse Plot A100) V2.00
     Serial bxaw0
     Copyright (C) 1988 INLAB, Inc.

     p16r4 revision 89.2.11

                     11 1111 1111 2222 2222 2233
         0123 4567 8901 2345 6789 0123 4567 8901

       0 ---- ---- ---- ---- ---- ---- ---- ----   OE
       1 X--- X--- X--- ---- ---- ---- ---- ----   +
       2 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   +
       3 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   +  !pin19
       4 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   +
       5 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   +
       6 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   +
       7 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX   +

         | |  | |  | |  | |  | |  | |  | |  | |
      pin2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 |
           |    |    |    |    |    |    |    |
         pin1   18   17   16   15   14   13   11

      Legend:

   X  : Cell intact     (JEDEC 0)
   -  : Cell programmed (JEDEC 1)

   X- : True input term
   -X : Complement input term
    XX : Any XX pair in a product term yields product term
LOW.
    -- : No input term (don't care).  A product term com-
prised
        entirely of -- yields product term HIGH.
```

Compare the NAND3.LST file to Figure 5. This format, called a fuse plot, is just a printer oriented version of the logic diagram. It has some new words in it – an input term is another word for a signal; a product term is another word for AND gate. Other than that, the fuse plot is about the same as a logic diagram.

The proLogic Simulator

The proLogic Simulator is a software version of a device programmer. To load it, enter the DOS command

```
LS NAND3
```

The simulator uses the fuse list lines from the JEDEC file (NAND3.JED) to create a functional device model. It then executes the test vectors against this model. The test results are placed in file NAND3.TST:

```
proLogic Simulator V2.00
Copyright (C) 1988 INLAB, Inc.

Architecture Description: p16r4.lxa
JEDEC Fuse Information:   nand3.jed
JEDEC Test Vectors:       nand3.jed

V1     N000 NNNN NNNN NNNN NNHN
V2     N001 NNNN NNNN NNNN NNHN
V3     N010 NNNN NNNN NNNN NNHN
V4     N011 NNNN NNNN NNNN NNHN
V5     N100 NNNN NNNN NNNN NNHN
V6     N101 NNNN NNNN NNNN NNHN
V7     N110 NNNN NNNN NNNN NNHN
V8     N111 NNNN NNNN NNNN NNLN

No errors detected with 8 Test Vectors.
```

You will find the same test vectors in NAND3.JED. The more readable source which created them is in NAND3.PLD.

In the example, no errors were found by the simulator. When errors exist, the simulator notes them with a single character code. At the end of the test listing, it appends a legend to explain the meaning of the error codes. This puts all the information you need in one place. There is no need to refer to the manual for a description of the error codes.

proLogic Simulator Input Files

When you run the simulator, you will need to supply the name of the Test Vectors file. It is a text file in JEDEC 3–A format which contains the test vectors (JEDEC field V) to be executed against the compiled device model. The DOS path name of the test vector file is the first parameter of the DOS command as shown in these examples:

```
LS VECTORS
LS TEST.JED
LS \TEST\VECTORS.VF1
```

When no file extension is specified, proLogic defaults to (.JED).

If your Test Vectors were created by the proLogic Compiler, that's the only parameter required. If not, the following are other options which can be used.

The **–j** Option

This option specifies the DOS path name of the Fuse Information file. It is also a text file in JEDEC 3–A format. It contains the programming information (JEDEC field F) which specifies the device function. Example:

```
LS VECTORS -JFUSES.JED
```

When no file extension is specified, proLogic defaults to (.JED). When the –j option is not supplied, proLogic assumes the fuse information is contained in the Test Vectors file.

The **–a** Option

This option specifies the DOS path name of the Architecture Description file. This file contains a description of the programmable device to be simulated. proLogic uses this file and the Fuse Information file to compile an optimized device model prior to beginning actual simulation. Example:

```
LS NAND3 -A\LXA\P16R4
```

Architecture Description files always have a (.LXA) file extension. When the –a option is not specified, proLogic examines the Fuse Information file for a field beginning with the characters N_csid. If found, the remainder of the field characters identify the Architecture Description file.

The proLogic Simulation Algorithm

The simulator behaves just like an electronic device tester. The next few paragraphs describes the device tester as a piece of hardware. These are the details you need to know to test your program.

Before executing any test vectors, the simulator removes voltage from all device pins. It next applies test condition 0 to the device ground pin and 1 to the device V_{CC} pin to simulate power–on. All memory elements power–on clear unless the device manufacturer specifies that they power–on set. Note that power–on or –off may be simulated by a test vector which applies test conditions to the power and ground pins.

After power–on, the simulator executes each test vector in the order encountered in the Test Vectors file. Execution consists of performing these four steps in order:

1. Remove voltage from test pins (L,H,Z) in pin number sequence.
2. Apply input and I/O pin voltage (0..9,F,X) in pin number sequence excepting clock (C,K).
3. Apply clock pin voltage (C,K) in pin number sequence.
4. Read test pin voltage (L,H,Z).

Input pins with no voltage applied are assumed to float logic high. Applied voltages persist from test vector to test vector when neither driven nor tested (N).

Section 4

proLogic Diagrams

A 3–input NAND Gate

The source file NAND3.PLD which programs 16R4 pin 19 to be a 3–input NAND gate has these three lines:

```
include p16r4;
!pin19 = pin2 & pin3 & pin4;
pin19.oe = 1;
```

The first line names the the header file (.H) for the PLD. The other two lines are called assignment statements.

An assignment has an expression made up of signal names and gate operators. Signal names identify signals internal to the PLD to be programmed. The gate operators define signal relationships. Because each PLD is different, the signal names and gate operators also are different from one PLD to another. For example, a 24–pin PLD might have a signal named pin23, but a 20–pin PLD would not. The signal names for each PLD are documented by the prologic diagrams.

Look at the proLogic diagram for the 16R4. You'll find it in the Logic Diagram section. Each signal is labeled with its proLogic signal name.

When you write an assignment statement, it must begin with an output signal name. Output signal names always end with the "=" character. They name gate output signals.

The remainder of the assignment is an expression comprised of any of the signal names which you want as circuit inputs. The signal names which directly feed an AND gate are written using an "&" gate operator as a separator, as in the expression

```
pin2 & pin3 & pin4
```

Similarly, signal names directly feeding an OR gate are written using the "|" gate operator as a separator.

```
pt47 | pt46                    /* device a167 */
```

When there's no signal name on a gate output, use the gate's input expression in place of the missing signal name. The expression

```
!pin19 = ( pin2 & pin3 & pin4 )
```

represents an OR gate which has an output signal name in terms of one of it's unnamed AND gate inputs.

To program pin19 as an AND instead of a NAND, write

```
! pin19 = ( ! pin2 ) | ( ! pin3 ) | ( ! pin4 )
```

where the parenthesized expressions denote the outputs of three of the AND gates feeding the OR gate. The parenthesis aren't required. The expression

```
! pin19 =  ! pin2  |  ! pin3 |  ! pin4
```

means the same thing.

Assignments allow a separation of the punctuation parts of a signal name from the rest using spaces, tabs or even new lines. Also, pairs of parenthesis can be used to group expressions. You can also use the symbols "0" and "1" as fixed gate inputs, as in

```
pin19 . oe = 1
```

Brackets

When you find a name like

```
[!]pin16=
```

in the prologic diagrams, the brackets indicate an optional part. In this case, either

```
 pin16=
```

or

```
!pin16=
```

may be used as the output signal name.

Section 5

Symbols

Signal Names

The prologic diagrams for each Programmable Logic Device specify the signal names you are required to use in your programs. These names are the ones used in the examples in the earlier sections.

If you've looked at the Texas Instruments application briefs, or at the sample Application PLD files, you've probably noticed that it's standard practice to assign application specific names to the various signals. There are a number of good reasons for doing this:

1. PLD specifications can be complex. Appropriate mnemonics are a big help to comprehension, not only during initial PLD specification writing, but also when trying to comprehend an existing specification.
2. Certain signal names have conventional meaning, OE for Output Enable, CS for Chip Select, A7 for address line seven, and so forth. Also, some stan dard circuits have conventional names. To illustrate, if part of your PLD circuit is an SR latch, the inputs should be named R and S.
3. It saves extra documentation. If your PLD specification file uses the pin17.d signal name, then somewhere you must document that signal's function. On the other hand, if you call it D3 and the programmed PLD is titled "An Up–down Counter", the name itself implies counter output bit three.
4. Your programmed PLD is part of a circuit. It's convenient if the names used in the PLD specification are similar to those used by your schematic capture package.

Application signal names are specified by define statements. The statement looks like this

```
define cs = pin2;          /* chip select */
```

The define replaces the symbol to the left of the = with the symbols to the right of the = (except the ;). This example means "wherever cs is found, replace it with pin2". Thus the expression

```
!pin19=cs
```

becomes

```
!pin19=pin2
```

after define replacement.

The define is a powerful tool. With it you can create application specific signal names such as

```
define STATE2 = (!pin17 & pin16 & !pin15);
define STATE3 = (!pin17 & pin16 &  pin15);
define ERROR  = ! pin19;
```

so that an expression for PLD pin 19 can be written as

```
ERROR = STATE2 | STATE3
```

Define statements don't take effect until they are encountered. For this reason, it's a good idea to group them all together near the start of the program.

Symbols

proLogic recognizes three kinds of symbols:

1. A group of alphanumeric characters, such as

```
cs        define
pin2      DEFINE
3725      8259_CS
```

2. A group of characters, such as

```
=         ==        &!
;         %         =!
```

except that

```
( ) { }
```

are always single character symbols and

```
/*
```

always begins a comment.

3. Any characters within quotation marks

```
"this is an unusual symbol"
"/* and so is this */"
```

A comment is any text enclosed by /* and */. Since characters like newline are text, you can write very large comment blocks. Comments can be written anywhere you can use a space or a tab, that is, they are symbol separators.

Do not use any of these reserved symbols as signal names.

```
define          else            if
include         repeat          signal
state           state_diagram   test_vectors
title           truth_table
```

Do not use any symbol beginning with an underscore _ character as a signal name.

Upper and lower case letters are different. That is, ″define″ is not the same as ″DEFINE″, which is different from ″Define″. The define statement does not replace a variable's extension (a symbol after the dot). That is, if one of your variables is

```
define d = pin19;
```

the expression

```
pin17.d = d
```

becomes

```
pin17.d = pin19
```

The define statement can be used to tailor the operator symbols to your preference. proLogic uses the ″&″ for AND, ″|″ for OR and ″!″ as the negation attribute. But they can be changed to *, + and / by these defines:

```
define * = & ;       define + = | ;       define / = ! ;
```

Notice that the spaces between the symbols are required. If we had written

```
define *=&;
```

proLogic would see this as the symbol ″define″ followed by the symbol ″*=&;″. If you want to change the operator symbols you also need

```
define =/ = = ! ;
define */ = & ! ;
define +/ = | ! ;
```

to be able to write a natural expression such as

```
a=/b*/c+/d
```

without being forced to put in spaces to separate symbols formed from punctuation characters, as in

```
a= /b* /c+ /d
```

The define statement rescans after it does a replacement. For instance, if you just finished inputing a program and realized you had the polarity of one of the signals wrong, you might be tempted to make a ”temporary” fix with a define like

```
define cs = (!cs);       /* wrong */
```

This define might find an expression such as

```
x=cs
```

and replace it with

```
x=(!cs)
```

but then it rescans yielding

```
x=(!(!cs))
x=(!(!(!cs)))
x=(!(!(!(!cs))))
.
.
.
and so forth...
```

Things like

```
define a=b;    define b=a;
```

present similar opportunities.

In practice, these don't come up very often. When they do, you'll find that proLogic tells you what symbol it is seeing. A number of instances can be fixed just by adding parentheses. As a further aid, the first part of the listing (.LST) file shows you the signal specifications after everything's completed.

Section 6

Expressions

Operators

In Section 4 we wrote a 3–input AND gate as

```
! pin19 = ! pin2 | ! pin3 | ! pin4
```

to make the point that an AND gate on an active low output cell requires the consumption of three product terms. That's also the form required to map onto the proLogic Diagram. We loosened up the strict version of signal names by writing

```
!pin19=
```

as

```
! pin19 =
```

Now we're going to take the next step away from the proLogic Diagram and write

```
! pin19 = !(pin2 & pin3 & pin4)
```

When proLogic sees this kind of an expression, it uses arithmetic rules to transform it internally back into the sum–of–products form which maps to the diagram. If you compile

```
! pin19 = !(pin2 & pin3 & pin4)
```

and look at the (.LST) file, you'll see that it has become

```
!pin19= !pin2 | !pin3 | !pin4
```

Expressions are evaluated according to priority. The ″|″ operator has a lower priority than the ″&″ operator. Both ″|″ and ″&″ have a lower priority than the ″!″ attribute. In the expression

```
!a | b & c
```

the highest priority things get evaluated first: first the ″!″

```
(!a) & b | c
```

then the ″&″

```
((!) & b) | c
```

leaving the ″|″ for last.

When the normal priorities work against you, there's another rule which says that parenthetical expressions have a higher priority than operators and attributes so that the expression

```
!((a | b) & c)
```

becomes

```
!a & !b | !c
```

Similarly,

```
(a | b) & (c | d)
```

becomes

```
a & c | a & d | b & c | b & d
```

You can even write expressions like

```
! ( a = b & ! c )
```

which turns out to be

```
!a = !b | c
```

The assignment operator = has a lower priority than most, so that in the expression

```
a = b & c
```

the

```
b & c
```

expression is evaluated before it is assigned to a. Conversely the . (dot) operator has a very high priority so that it applies even before the "!" as in

```
! pin27 . D = 1
```

which proLogic sees as

```
! ( pin27 .d ) = 1
```

The dot operator changes upper case letters in the extension to lower case. The equality operators "==" and "!=" provide a way of writing the equivalent of the logical Exclusive NOR/OR functions. That is,

```
a == b
```

is the same as the expression

```
a & b | !a & !b
```

The "!=" operator is defined to be the inverse. This means

```
a != b
```

is the same as

```
! ( a == b )
```

or

```
a & !b | !a & b
```

As a general note, all operators are binary. They require an expression both before and after the operator. For example a attributes like "!" apply to a single expression.

Statements

An assignment expression such as

```
a = b & c
```

becomes a statement when it is followed by a semicolon, as in

```
a = b & c;
```

You've already seen the define statement. It is also terminated by the semicolon. Syntactically, your program is a sequence of statements.

There is another kind of statement called a block which is terminated by a "}". The title block is the subject of the next section.

Section 7

The Title Block

A title block

```
title {   Function: Special Barrel Shifter.
          Designer: Acme Products.
          Date:     4 July 1988.
          .
          .
}
```

defines text to be copied to the JEDEC output file as documentation. The title block is a statement. It may appear almost anywhere in your program, but you can only have one. The text enclosed by the braces { and } is copied to the JEDEC output file. Because the JEDEC file format uses an * character to delimit the documentation, your text may not contain this character.

The semicolon is not required to terminate the title block (or any other block). Note that our style of block which has a separate line for the closing } and begins the text on the same line as the opening { may not be to your taste. Feel free to experiment to find a style that suits you.

The text is also copied to the fuse map portion of the (.LST) file.

Section 8

Truth Table Blocks

The Classic Seven-Segment Display

The following is the truth table to program a PLD as a hex decoder driver for a seven-segment display.

```
truth_table {  /*             aa
                          +--------+
                          |        |
                        bb|        |ff
                          |   gg   |
                          +--------+
                          |        |
                        cc|        |ee
                          |   dd   |
                          +--------+
               */

               q3  q2  q1  q0  :  aa  bb  cc  dd  ee  ff  gg;

   /*  0  */    0   0   0   0  :   0   0   0   0   0   0   1;
   /*  1  */    0   0   0   1  :   1   0   0   1   1   1   1;
   /*  2  */    0   0   1   0  :   0   1   0   0   1   0   0;
   /*  3  */    0   0   1   1  :   0   1   1   0   0   0   0;
   /*  4  */    0   1   0   0  :   1   0   1   1   0   0   0;
   /*  5  */    0   1   0   1  :   0   0   1   0   0   1   0;
   /*  6  */    0   1   1   0  :   0   0   0   0   0   1   0;
   /*  7  */    0   1   1   1  :   0   1   1   1   0   0   1;
   /*  8  */    1   0   0   0  :   0   0   0   0   0   0   0;
   /*  9  */    1   0   0   1  :   0   0   1   1   0   0   0;
   /*  A  */    1   0   1   0  :   0   0   0   1   0   0   0;
   /*  B  */    1   0   1   1  :   1   0   0   0   0   1   0;
   /*  C  */    1   1   0   0  :   0   0   0   0   1   1   1;
   /*  D  */    1   1   0   1  :   1   1   0   0   0   0   0;
   /*  E  */    1   1   1   0  :   0   0   0   0   1   1   0;
   /*  F  */    1   1   1   1  :   0   0   0   1   1   1   0
}
```

Each truth_table block creates a block of assignment statements, but is set up in tabular form to save you writing time and to help you visualize all the cases.

The table heading line

```
     q3   q2   q1   q0   :   aa   bb   cc   dd   ee   ff   gg   ;
```

lists the input and output expressions. The input expressions are listed first and are separated from the output expressions by the colon. The semicolon ends the heading line.

Each other line of the table is a detail line. Detail lines follow the format set up by the heading line with respect to number of input and output expressions. These lines create assignment statements. If your truth table had only this single detail line

```
 /* 0 */      0    0    0    0    :    0    0    0    0    0    0    1 ;
```

then only this one assignment statement would be created:

```
   gg = q3==0 & q2==0 & q1==0 & q0==0;
```

These two detail lines

```
/*  0 */      0    0    0    0    :    0    0    0    0    0    0    1 ;
/*  1 */      0    0    0    1    :    1    0    0    1    1    1    1 ;
```

would create all these:

```
     aa = q3==0 & q2==0 & q1==0 & q0==1;
     dd = q3==0 & q2==0 & q1==0 & q0==1;
     ee = q3==0 & q2==0 & q1==0 & q0==1;
     ff = q3==0 & q2==0 & q1==0 & q0==1;
     gg = q3==0 & q2==0 & q1==0 & q0==0
        | q3==0 & q2==0 & q1==0 & q0==1;
```

And these three

```
/* 0 */      0    0    0    0    :    0    0    0    0    0    0    1   ;
/* 1 */      0    0    0    1    :    1    0    0    1    1    1    1   ;
/* 2 */      0    0    1    0    :    0    1    0    0    1    0    0   ;
```

would create these:

```
     aa = q3==0 & q2==0 & q1==0 & q0==1;
     bb = q3==0 & q2==0 & q1==1 & q0==0;
     dd = q3==0 & q2==0 & q1==0 & q0==1;
     ee = q3==0 & q2==0 & q1==0 & q0==1
        | q3==0 & q2==0 & q1==1 & q0==0;
     ff = q3==0 & q2==0 & q1==0 & q0==1;
     gg = q3==0 & q2==0 & q1==0 & q0==0
        | q3==0 & q2==0 & q1==0 & q0==1;
```

All 16 of the detail lines would produce seven assignments and the aa assignment would be the sum of four products — one for each 1 in the aa column. Even though the final signal specifications which show up in the listing file can be reduced to their minimum logical equivalents, a truth table is certainly a way to generate a lot of logic with minimal work on your part.

The reason inputs use the equality operator == is so that you can have more complicated things than 0s and 1s. In

```
     truth_table { a   b   :   z   ;
                   c   d   :   1   ;
     }
```

the assignment is

```
z = a==c & b==d;
```

You'll notice that we used the phrase input expression at the start of this section. This means that not only can you negate inputs, as in

```
truth_table { !a  !b  :  z  ;
               1  !d  :  1  ;
}
```

which creates the assignment

```
z = (!a == 1) & (!b == !d);
```

You can also write almost any valid expression, such as

```
truth_table { !(a & b)  c   :  z  ;
                 c   d | e :  1  ;
}
```

which is

```
z = (!(a & b)) == c & c == (d | e);
```

The only expression that you have to stay away from in the detail lines is a signal named X (in either upper or lower case). That's because we picked this signal name to mean don't care. If you have to use it, put it in parenthesis. This table:

```
truth_table { a   b  :  x  y  ;
              1   x  :  1  0  ;
              x  (x) :  0  1  ;
}
```

yields

```
x = a==1;
y = b==x;
```

The output side of the truth tables is easier because there are only three characters permitted — 0, 1 and X (or x). These make menu selections from the output expressions in the header.

The X is again a don't care. It means that this detail line doesn't affect the assignment for the output. The 1 (0) specifies that the output is logic high (low) for the case defined by the input. The output characters can be written together. That is,

```
0x01
```

is the same as

```
0 X 0 1
```

Output Polarity

You should think of the polarity problem in exactly the same way that you do when you are about to write an assignment statement. Logic high (the 1s) produce assignments which are true relative to the signal specification names. If we remove the abstraction and show some real signal specification names for the 16R4 on this table:

```
   truth_table { pin2 pin3 pin4   : !pin19  ; /* 3-input NAND
*/
                    0    X    X    :    1    ;
                    1    0    X    :    1    ;
                    1    1    0    :    1    ;
                    1    1    1    :    0    ;
      }
```

it becomes clear that we've got it right.

Section 9

State Diagrams

A Module–3 Counter

proLogic can help you implement programmable logic state machines. Other names used for this general subject are finite state machines, sequencers, sequential circuits, counters, Mealy model/machine/circuit, Moore model/machine/circuit, state diagram, ect..

If you want to create a circuit using clocked flip–flops with feedback, then you can use state blocks instead of assignments.

Let's take a modulo–3 counter for an example. It uses two flip–flops as state variables. Each of the three permitted combinations of Os and 1s in the state variables is called a state. On power up it is in one of the states and on each clock a next state is established which depends, in part, on the current state. Figure 6 illustrates a modulo–3 counter.

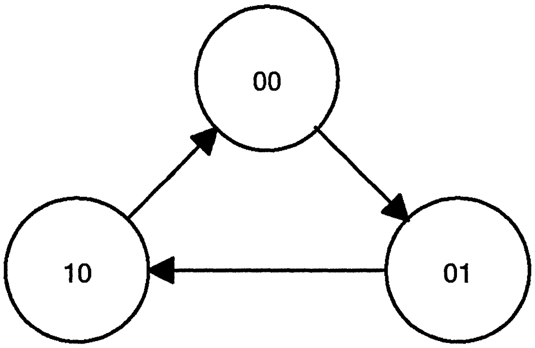


FIGURE 6. Modulo–3 Counter

The assignments for a D flip–flop implementation are determined by inspection to be

```
    v1.d = v0.q;
    v0.d = !v1.q & !v0.q;
```

The same counter written as a state diagram appears as follows:

```
state_diagram v1,v0 {
    /* modulo-3 counter */

    state s0=00
        s1;

    state s1=01
        s2;

    state s2=10
        s0;
}
```

A state diagram appears different from the assignments. This is because each assignment is a single statement but the state diagram has a hierarchical structure.

First while the outermost state_diagram statement.

```
state_diagram v1,v0 {
    . . .
}
```

It declares v1 and v0 to be state variables. Their values are defined in the body of the diagram. The body is all of the statements enclosed by the braces { and }.

In our counter example, the body consists of a state statement for each of the three states.

The first state statement.

```
state s0=00
    s1;
```

It declares the state name s0 and its value combination (v1.q,v0.q)=(0,0). The state statement also has a body. The body can be a single statement without braces, as written above or it can be one or more statements enclosed by braces, as in

```
state s0=00 {
    s1;
}
```

The transition statement

```
s1;
```

consists of a state name, as it appears in one of the state statements of this state diagram, followed by a semicolon. It specifies what state occurs on the next clock.

IF-ELSE

The if–else statement is used to condition other statements. The general form is

```
     if (expression)

          statement-1          /* then part */

  else

          statement-2          /* else part */
```

where the else part is optional.

To show its use, let's make our modulo–3 counter stay low (in state s0) until triggered by an external signal (count) as shown in Figure 7.

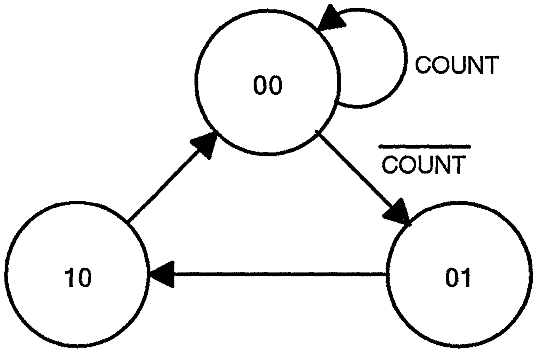


Figure 7. Modulo–3 Counter

This is coded

```
     state_diagram v1,v0 {
          state s0=00
               if (count)
                    s0;
               else
                    s1;
          state s1=01
               s2;
          state s2=10
               s0;
     }
```

The if–else statement conditions its then part with the expression in parenthesis and its else part with the expression's complement. Because its then part (or else part) can be another if–else statement or a block (one or more statements enclosed by braces { and }), things can get as complex as necessary.

The if–else and state statements can also condition assignments.

You can add an output to state s2 by writing

```
state s2=10 {
     overflow=1;
     s0;
}
```

or make it an up–down counter with

```
state s2=10
     if (up) {
          overflow=1;
          s0;
     }
     else
          s1;
```

Note that the style of one statement per line is Texas Instruments style. State statements can also be written as follows:

```
state s2=10    if (up) {overflow=1; s0;}
               else s1;
```

Choose a style that suits you. Also, an if–else can be used anywhere in your program– not just within a state diagram.

Global Transitions

Extend our modulo–3 counter so that instead of being triggered to count, it stays in state 00 unless enabled.

```
state_diagram v1,v0 {
     state s0=00
          if (count)
               s0;
          else
               s1;
     state s1=01
          if (count)
               s0;
          else
               s2;
     state s2=10
          s0;
}
```

State machines often have some condition such as this which applies to every state. These global transitions can be factored out and written before the first state.

The diagram

```
state_diagram v1,v0 {
     if (count)
          s0;
     state s0=00
          s1;
     state s1=01
          s2;
     state s2=10
          s0;
}
```

looks more like what it really is – a modulo–3 counter with an active high reset.

Illegal States

If the modulo–3 counter (shown in Figure 8) is implemented on flip–flops which power up to a random state the counter may be in trouble. It counts

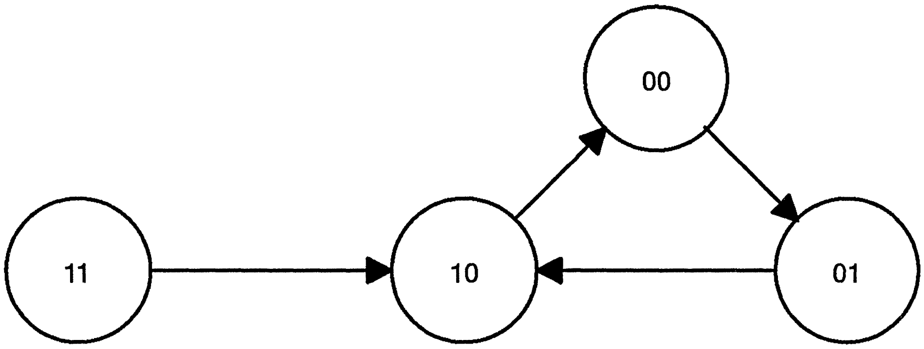


FIGURE 8. Modulo–3 Counter

This may be acceptable. If it isn't, you can bring it into line by adding an additional

```
state s3=11    /* illegal */
     s0;
```

If your machine is something other than a counter you can also specify that a state has don't cares for some, but not all, of its values. Instead of adding a new state, the existing state s2 could be redefined as

```
state s2=1x
     s0;
```

The x, which can be either upper or lower case, means that either a 10 or a 11 identifies state s2.

State Variables

The state variables declared by the state_diagram statement must name a flip–flop with internal feedback. The extension may be omitted. For example

```
state_diagram pin17,pin16 {
          . . .
}
```

declares that the pin17.q signal holds the most significant state value for a 16R4 implementation. On this PLD, proLogic synthesizes assignment statements for pin17.d= and pin16.d=.

You can also precede a state variable with a "!" which inverts all logic for that flip–flop. The modulo–3 counter example, if written for a 16R4 with active low outputs would be

```
state_diagram !v1.q, !v0.q {
     /* p16r4 modulo-3 counter */

     state s0=00
          s1;

     state s1=01
          s2;

     state s2=1x
          s0;
}
```

which is equivalent to

```
state_diagram (v1,v0) {
     /* modulo-3 counter */

     state s0=11
          s1;

     state s1=10
          s2;

     state s2=0x
          s0;
}
```

Conditioning

The statements which form the body of the state block are conditioned by the state. The following is a state which is the highest state of a modulo–5 counter

```
state s4=1XX {
     Y=1;
     s0;
}
```

Because the expression Y=1 is contained within the body of state s4, it is conditioned by s4. This means the resulting expression is extended to be true only when state variable v2 (the most significant variable) is logic high and yields

```
Y = v2.q;
```

If Y is a combinatorial output, it becomes high shortly after clocking a 1 into v2. If Y is a registered output, it becomes high on the next clock which simultaneously causes v2 to go low as it transitions to state s0. What happens to Y after that depends on what is written in the state s0 block. Conversely, if you wanted register Y high on the clock of 1 into v2 to enter state s4, Y would need to be set up in the prior state. For an up–counter,

```
state s3=000 {
     Y=1;
     s4;
}
```

would do it, but in an up–down counter, both preceding states are required to set up Y.

```
state s0=000
     if (up)
          s1;
     else {
          Y=1;
          s4;
     }

state s3=011
     if (up) {
          Y=1;
          s4;
     }
     else
          s2;
```

Default Transitions

An unconditioned transition statement is called a default transition. In

```
state s0=00
     s1;
```

the s1 is a default transition because it is not conditioned by an if–else statement. The state

```
state s0=00
     if (count)
          s0;
     else
          s1;
```

could also have been written

```
state s0=00 {
     if (count)
          s0;
     s1;
}
```

because the default transition specifies the state change when none of the other transitions apply.

States which have no default transition remain in their current state when none of the other transitions apply. Thus

```
state s0=00
     if (!count)
          s1;
```

also holds in s0 while count is logic high.

Each state block can have only one default transition statement.

Section 10

Test Vector Blocks

Test vectors are used by device programmers or logic simulators to verify that the logic functions defined for a PLD are correct. These vectors describe the inputs to the programmed PLD and specify the outputs expected after applying each set of inputs.

Each test_vectors block defines a sequence of vectors to be applied to a set of test condition variables. These blocks are written in tabular form with semicolons separating each line of the table.

```
test_vectors {

          pin3   !pin1   pin18  pin19;
           1       1       H      L;
           1       0       L      L;
           0       0       L      H;
}
```

The first line of the table

```
          pin3   !pin1   pin18  pin19;
```

names the test condition variables. In your program you probably have some define statements to make the names more meaningful. Because test conditions are applied to the pins of the device, not internal signals, you don't necessarily need to adhere strictly to the proLogic Diagram signal names. Anything close will usually work.

Each other line of the table is a test vector. The first vector

```
           1       1       H      L;
```

specifies the test conditions 1, 1, H and L to be applied respectively to the test condition variables pin3, !pin1, pin18 and pin19. Spaces aren't required. You can write

```
           1       1       H      H ;
```

or

```
           11HH;
```

as long as there is one test condition for each test condition variable. Lower case letters can be used if you prefer. This table lists the allowable test conditions.

0	drive input LOW	1
1	drive input HIGH	0
2..9	drive input to supervoltage #2-9	2..9
C	drive input LOW-HIGH-LOW	K
F	float input or output	F
H	test output HIGH	L
K	drive input HIGH-LOW-HIGH	C
L	test output LOW	H
N	power pins and outputs not tested	N
P	preload registers	P
X	output not tested, input default level	X
Z	test input or output for high impedance	Z

The last column lists the inverse condition which is used when a test condition variable is negated. The block

```
test_vectors { !pin1;
               1;
               0;
               N         }
```

is equivalent to

```
test_vectors { pin1;
               0;
               1;
               N         }
```

There is also a special kind of a test vector which is used to create one or more JEDEC buried register vectors. This one begins with the symbol internal and is followed by as many 0s, 1s, Ls and Hs as there are internal registers in the PLD.

Note:

If you're new to PLD design, be aware that we've left out some very important practical considerations. When a device programmer executes preload vectors, or vectors which have supervoltages, it may apply high voltages to pins. Refer to the data sheets for your exact PLD. Your device programmer manual also provides important cautions to observe when testing.

The Repeat Block

A repeat block, written

```
    repeat N {
         . . .
    }
```

can be used anywhere in your program. Its function is to create N copies of the symbols contained in the body. The following is a sample application.

```
test_vectors {
     /* 4-bit counter partial test */

     pin1      pin17 pin16 pin15 pin14;

      P          0     0     0     0;    /* preload */

   repeat 15 {
      C          X     X     X     X;    /* clock 15 times */
  }

      0          H     H     H     H;    /* OK if all high */

      C          L     L     L     L;    /* wraparound */

}
```

The } which terminates the repeat block was placed on the line after

```
      C          X     X     X     X;    /* clock 15 times */
```

but could also have been placed on the same line after the semicolon.

Repeat blocks are not statements. This means they can be used almost anywhere. Nesting is permitted, such as

```
   repeat 15 { C repeat 4 { X }; }
```

Section 11

Include Files

Exactly one

```
include file-name;
```

which identifies the Architecture Description file (.LXA) for the target device must be present somewhere in each program. file–name is a symbol string such as

```
P16R4.LXA
```

or

```
\prologic\p22v10.lxa
```

When you don't specify a file extension of .LXA, the include has a different function. In this case, the include and its symbols are replaced by the contents of the identified file.

You might want to use include files to define commonly used symbols like

```
define LOW = 0;
```

to save time and assure consistency over all of your PLD programs.

Even though a semicolon terminates the file–name, the include is not a statement. Like the repeat block, it can be used almost anywhere. Included files may themselves have includes. A repeat block may contain includes (and vice versa).

When no file extension is supplied, the default is .H (for header). But if your file has spaces as an extension, make this clear.

```
include myfile. ;
include myfile.″   ″;
include ″myfile.   ″;
```

Header Files

proLogic provides a library of header files (.H). These provide extra levels of information about each device to make it easier for you to write a program. You might think of them as a subroutine library. Header files answer questions like:

```
how many pins?
is register preload available?
what pins have registers?
how do states map to D flip-flops?
to SR flip-flops?
does   if (c) x=a;   mean   x= (c & a)?
does it mean   x = (c & a) | (!c & x.q)?
does the compiler drive output enables when you don't?
what about logic minimization?
.
.
.
```

The only logical operators which the compiler recognizes are the AND and OR. Header files may offer other operators such as XOR.

When you select a header file such as

```
include p16r4;
```

at the beginning of your program, you do not need an

```
include p16r4.lxa;
```

because the header file will bring the Architecture Description in as part of itself.

If your preference is for as little "help" as possible, use an include such as

```
include p16r4.lxa;
```

at the start of your program. This requires you to be very explicit when you write the program. Most of the higher level statements can't be used in this mode. We recommend you include the STDSYN.H file in addition to the LXA file. It has define statements which relax the operator symbols to make your expressions more natural.

Section 12

Signal Specifications

Although proLogic has the ability to manipulate, optimize, and even synthesize Boolean equations. Everything gets put into the form which will map onto the proLogic Diagram signal names and gate operators. This form is called a signal specification.

In the example

```
!pin19 = pin2 & pin3 & pin4
```

The signal specification is

```
"!pin19=" pin2 & pin3 & pin4
```

Quotes are used to write the seven–character symbol

```
!pin19=
```

because that is the exact name on the proLogic Diagram. Names such as

```
!pin2
```

would also need to be quoted.

The rules for writing a signal specification are as follows:

1. A signal specification always begins with an output signal name. These names always end with an = (equal) character.
2. An input signal name must follow the output signal name. These names never end with an = character.
3. After the first input signal name, more input signal names can be listed if they are separated by gate operators.

Gate Operators

The prologic diagrams implicitly use these operators:

```
&    Logical AND Gate
|    Logical OR Gate
%    Logical Exclusive OR Gate
```

Other operators unique to the PLD may be specified by individual diagrams.

The characters

```
!    Logical Negation
=    Assignment
.    Dot
```

are not gate operators. They are required to ”spell” some signal names.

Gate Operators Function

There is always a conceptual ”current gate”. The importance of the current gate concept is that it provides the environment for interpreting the meaning of the input signal names. Input names always identify signals with respect to the current gate.

The current gate becomes important in multi–level logic. The output name usually identifies a hierarchical logic structure of one, two, or more levels. For example, in a 16R4, the output name pin19.oe= identifies a one level product term. The output name !pin19= identifies a two level sum of product terms. The current gate is always one of the gates at the lowest level of the hierarchy. In both of these examples, the current gates are AND gates.

The initial current gate is identified by the output name. It is always the first (or topmost) of all of the possible current gates. The operator which identifies the current gate (in these examples, the ″&″), has no effect on the current gate. It serves only as a separator between input names. An operator which identifies a gate at a higher level in the PLD logic structure (in these examples, the ″|″) changes the current gate to be the next of all possible current gates.

In a one level structure such as the 16R4 pin19.oe=, there is only one possible current gate – the 32–input AND which drives the output enable for pin 19. It is the initial current gate identified by the pin19.oe= output name. In the signal specification for this signal only ″&″ operators may appear.

In the two level structure identified by the !pin19= output name, there are seven AND gates at the lowest level. The output name identifies the first of these as the current gate. All input names identify connections to this AND gate until an ″|″ operator is encountered. After the ″|″, all input names identify connections to the second AND gate, until the next ″|″ when they apply to the third, and so forth.

0 And 1

There are two special signal names which apply to the current gate of all devices. They are 1, which means the output of the current gate is logically true, and 0, which means the the output of the current gate is logically false. Since these names apply to the gate output, no other input names may apply to that gate. For example,

```
    "pin19.oe=" 1 & pin2
```

is improper.

The 0 and 1 signal names used in the signal specification are different from the 0 and 1 symbols used in expressions. In an expression, they are gate input signals. That is,

```
    ( pin19.oe = 1 & pin2 )
```

is transformed into

```
    ( pin19.oe = pin2 )
```

In the first matrix of a 16R4 there are 256 programmable cells. These two signal specifications cause only the last cell to remain connected, while the first 255 are disconnected (programmed).

```
    "pin19.oe=" 1

    "!pin19="   1 | 1 | 1 | 1 | 1 | 1 | "!pin12"
```

The Signal Statement

Most PLD programs for most devices can be expressed entirely in terms of boolean equations using logical AND, OR and NOT. That's why the assignment statement such as

```
a = b & !c;
```

requires you to do the minimum amount of typing on your keyboard. But some PLDs have XOR gates or signature words.

For these instances the signal statement always provides the ability to program at the signal specification level. Another way of writing

```
a = b & !c;
```

is

```
signal "a=" b & "!c";
```

In this statement, everything between signal and the semicolon must be a signal specification.

You also have the option of letting proLogic do most of the work for you. In the signal statement, anything enclosed by parenthesis is an expression. Thus

```
signal "a=" b & (!c);
signal "a=" (b & !c);
signal (a = b & !c);
```

are all equivalent.

To make it even easier, you can also write the =, ., and ! symbols separately.

proLogic will translate

```
signal a = ! b . c;
```

into

```
signal "a=" "!b.c" ;
```

because it knows the rules for forming a signal name.

Appendix A

A Designer's Guide to the TIBPSG507

Robert K. Breuninger and Loren Schiele
with Contributions by
Joshua K. Peprah

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

List of Illustrations

INTRODUCTION

The term PSG stands for Programmable Sequence Generator. The PSG is the newest member of the programmable logic family. It combines the powerful benefits of programmable array logic (PALs) with the specialized world of Field Programmable Logic Sequencers (FPLSs).

Applications such as waveform generators, state machines, timers, and simple logic reduction are all possible with a PSG. By utilizing the built-in binary counter, the PSG is capable of generating complex timing controllers. In short, the PSG offers the system designer an extremely powerful building block.

The purpose of this application report is to describe the functional operation of the PSG507 and demonstrate how it can be applied in real-world applications. Three design examples that highlight the features and flexibility of the PSG will be discussed.

FUNCTIONAL DESCRIPTION

Figure 1 shows the architecture of the PSG507. Major features include 13 inputs, eight programmable registered or nonregistered outputs, eight S/R state registers, and a 6-bit binary counter with control logic. The clock input is fuse-programmable for selection of positive or negative edge triggering.

The binary counter, state registers, and output cells are synchronously clocked by the fuse-programmable clock input. The clock polarity fuse selects either positive or negative edge triggering. Negative edge triggering is selected by blowing the clock polarity fuse. Leaving this fuse intact selects positive edge triggering.

Each output cell on the PSG can be configured for registered or nonregistered operation through the output multiplexer fuse. Nonregistered operation is selected by blowing the output multiplexer fuse. Leaving this fuse intact selects registered operation.

The PSG507 has 13 inputs, each providing a true and complement input to the AND array. Pin 17 functions as either an input and/or an output enable. Blowing the output enable fuse lets pin 17 function as an output enable but does not disconnect pin 17 from the input array. When the output enable fuse is intact, pin 17 functions only as an input with the outputs being permanently enabled.

The 6-bit binary counter is controlled by a synchronous clear and a count/hold function. Each control function has a nonregistered and registered option. When either SCLR0 or SCLR1 is taken active high, the counter resets to zero on the next active clock edge. When either $\overline{\text{CNT}}$/HLD0 or $\overline{\text{CNT}}$/HLD1 is taken active high, the counter is held at the present count and is not allowed to advance on each active clock edge. The SCLR feature overrides the $\overline{\text{CNT}}$/HLD feature when both functions are simultaneously active high. The functional benefit of both these features will be further clarified in the examples shown later in this appliction report.

The eight internal state registers feed back into the AND array. These registers can be used to store input data, to keep track of binary count sequences, or they can be used as output registers when connected to a nonregistered output cell. The state registers differ from the output registers in that they feed back into the input array. They can also be used to override an operating sequence such as demonstrated in the designer notes located at the end of this application report. By using extra state registers, the 6-bit counter can be expanded as shown in the second example. Other uses of the internal state registers will become apparent upon reading the examples shown.

THEORY OF OPERATION

The PSG architecture is capable of operating in many different modes. When comparing the operation of a PSG to a PAL, the outputs in both devices can be configured as an AND/OR function of the inputs. One major difference between a PSG and a PAL is that a programmable OR array is used in the PSG. This allows a selected number of AND terms to be connected to each output as compared to a fixed number of AND terms assigned to each output on a PAL. The programmable OR array is the more efficient in that it lets the user assign the exact number of AND terms to each output as required by the application.

Another major difference between the PAL architecture and that of a PSG is that the output cells on a PSG are not fed back into the input array. Typically, output feedback is used for building a counter or for holding state information. Since the architecture of the PSG already includes state registers and a binary counter, the requirement for output feedback is eliminated in most applications. This is a benefit to the user because valuable output cells and AND terms are not wasted when generating these functions.

When a Field Programmable Logic Sequencer is compared to a PSG, the most obvious difference is the addition of a binary counter. Most state machine designs can be simplified by referencing all or part of each sequence to a binary count. This technique is highlighted in the third example shown in this application note. A comparison will also reveal that the output cells on a PSG can be configured

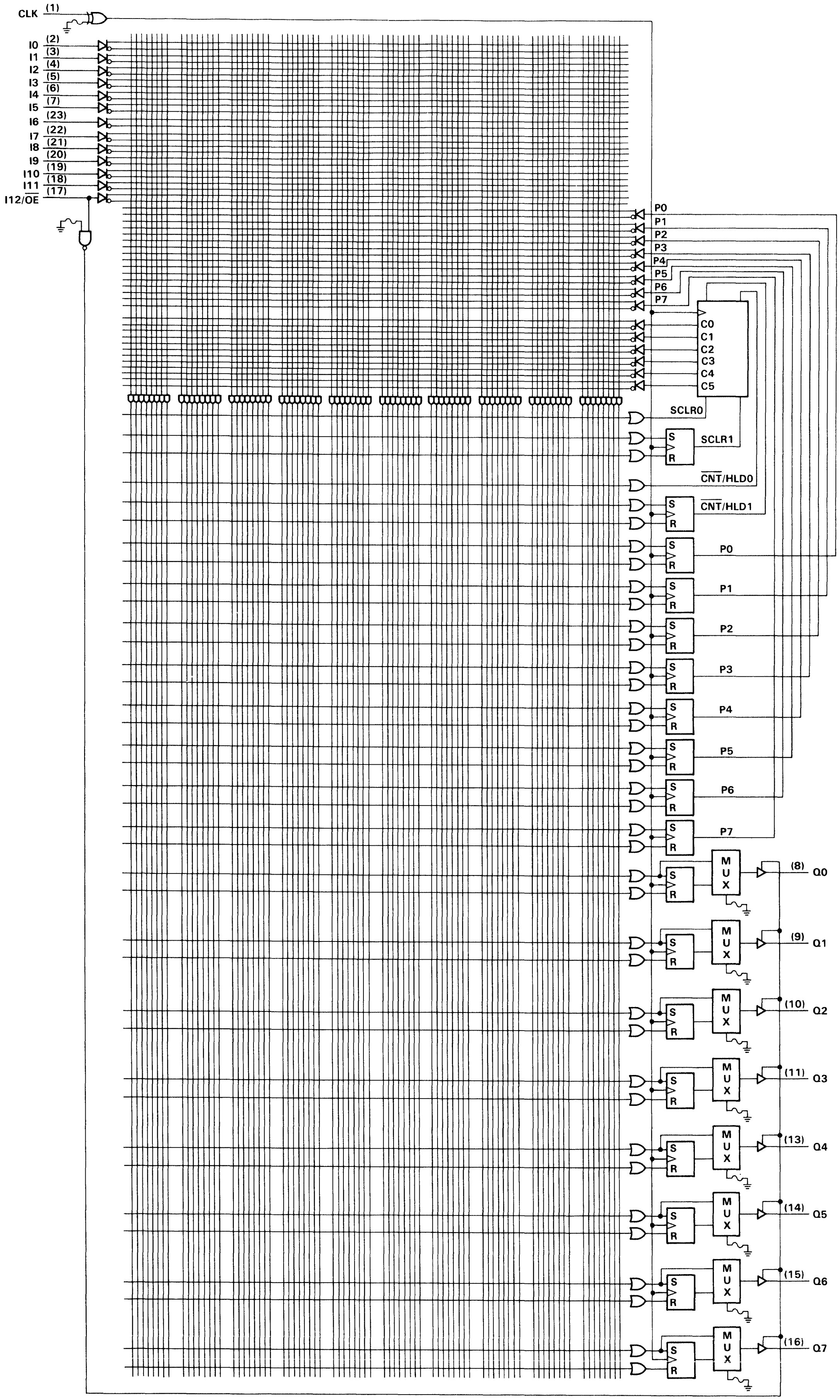


Figure 1. PSG507 Architecture

for nonregistered operation. This permits the outputs to be directly fed from the counter, AND/OR array, or state registers. Example 1 highlights this feature.

In short, the outputs of a PSG can be controlled by any or all of the following conditions:

- Present state of the inputs
- Present state of the binary counter
- Present state of the state holding registers

The key to understanding state machine design when using a PSG is to realize that different states can be assigned for each sequence. In other words, the assigned state determines which sequence is in operation. The length of each sequence is controlled by the SCLR function. Once the count sequence has been programmed to the desired length, each output can be easily decoded from the present state of the binary counter. The user will soon discover that complex state machines are easily developed when using this technique. This technique is demonstrated in Example 3.

Example 1: Waveform Generator

The first example demonstrates a design for a simple clock generator used for driving a microprocessor operating at 5 MHz (required duty cycle of 33.5% high, 66.5% low). In addition to the 5 MHz system clock (SYS CLK), a reference clock (REF CLK) operating at 15 MHz (50% duty cycle) and a peripheral clock (PCLK) operating at 2.5 MHz (50% duty cycle) are required for other timing controllers and peripherals throughout the system. Both clocks must be in close phase with the SYS CLK to guarantee synchronous operation within the system.

The above example demonstrates one of the many uses of the binary counter in the PSG. State registers are not used in this particular application, only the binary counter and three outputs. A 30 MHz clock, typically generated from a crystal, is used for driving the binary counter of the PSG. The three generated clock signals are decoded from the binary count. The unused inputs and outputs are still available for other sequential or combinational applications.

Figure 2 shows the timing diagram for the above application. For reference, a decimal count has been assigned to the master clock (PSG CLK) of the PSG. As shown in the timing diagram, at count 11 (1011_2) the sequence is repeated. By using the SCLR0 function, a logic equation can be defined to reset the counter at count 11. This concept is demonstrated in Figure 3.

With the binary counter programmed to clear at 11, it is a simple matter to decode the outputs from the binary count. With the REF CLK equal to the inverse of binary count zero (C0), REF CLK can be directly generated from the binary counter. A product term is required to connect C0 to the output cell. The output register is bypassed by blowing the output multiplexer fuse. Figure 4 shows how C0 can be connected.

SYS CLK and PCLK are decoded from the present state of the binary counter through the S/R outputs. Since the S/R register holds its present state until changed, product terms have to be used only during output transitions. For example, when the binary counter reaches one, a product term is used to reset the SYS CLK on the next clock transition. Below is a summary of the product terms required to control SYS CLK and PCLK. Note that the output transitions are set up in the previous clock cycle. Also note that only one product term is used regardless of how many output terms switch. This is demonstrated at count 5 and count 11. Figure 4 also shows how SYS CLK and PCLK are connected.

CNT 1:	Reset SYS CLK
CNT 5:	Set SYS CLK, reset PCLK
CNT 7:	Reset SYS CLK
CNT 11:	Set SYS CLK, set PCLK

This simple application demonstrates the basic concept of building a waveform generator using the PSG. This concept will be expanded further in Example 3 when a memory timing controller is developed. The basic rules for building a waveform generator are summarized below.

- Program the counter to reset to zero after the desired count length is reached.
- Generate the logic equations to control the outputs from the present state of the binary counter.

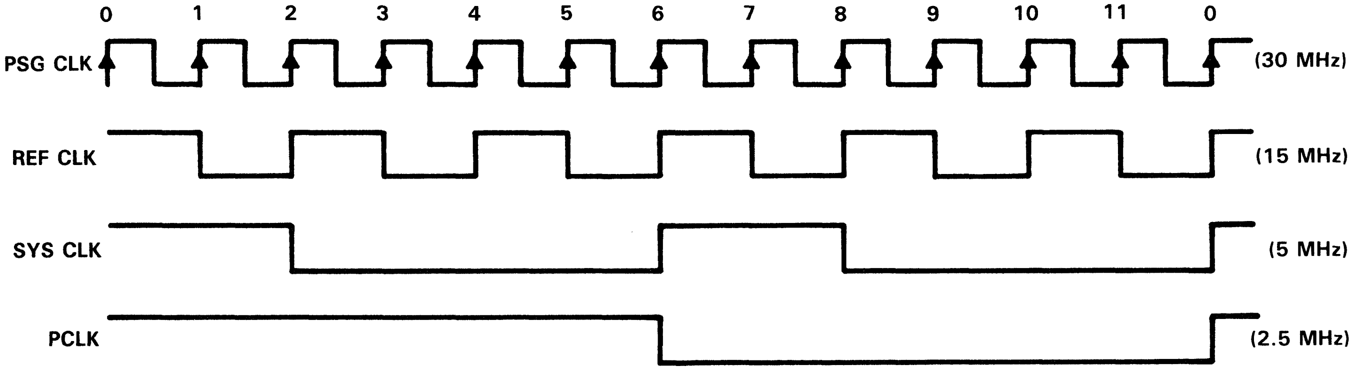


Figure 2. Clock Generator Timing Requirements (Example 1 — Waveform Generator)

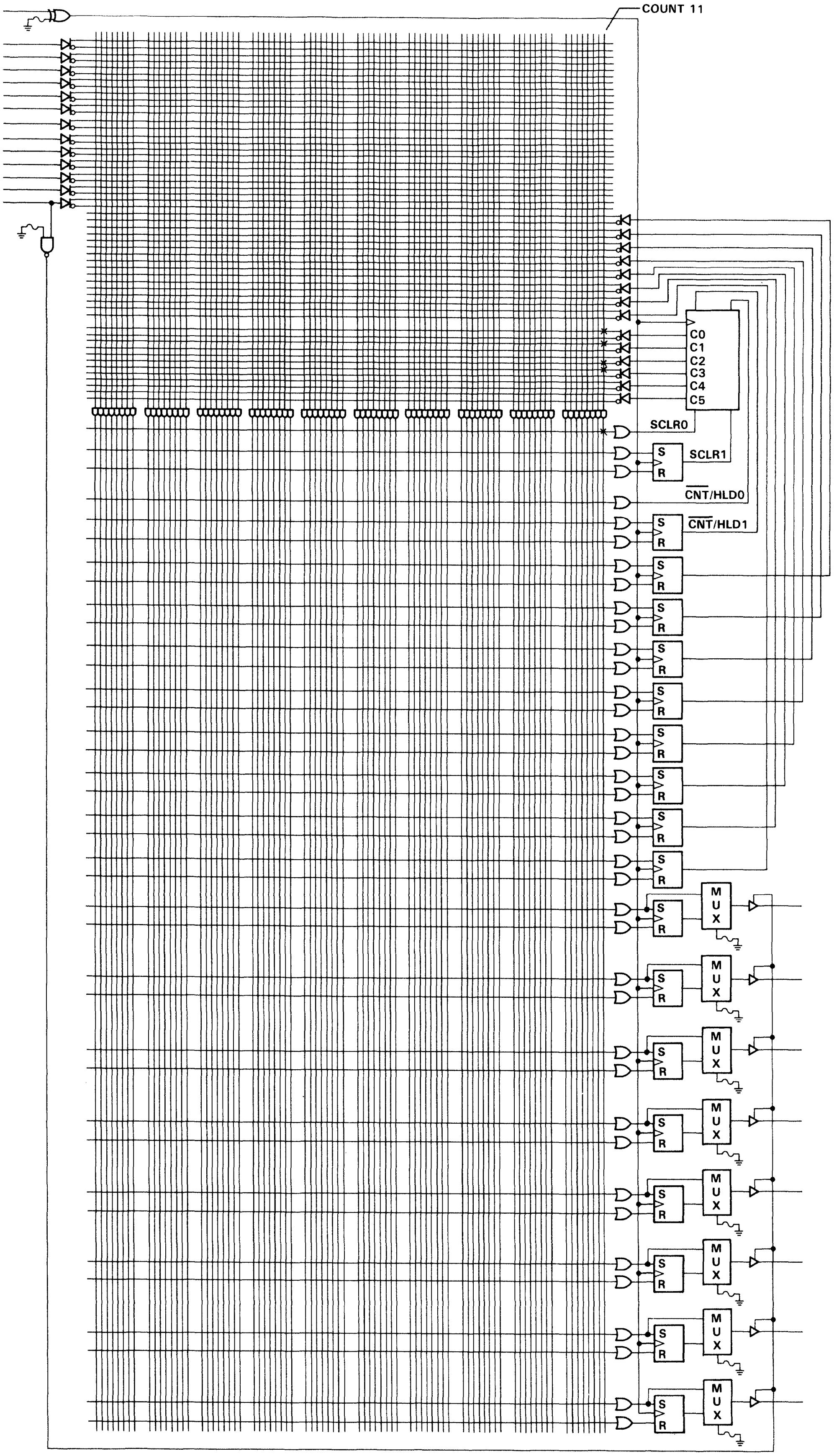


Figure 3. SCLR at COUNT 11
(Example 1 — Waveform Generator)

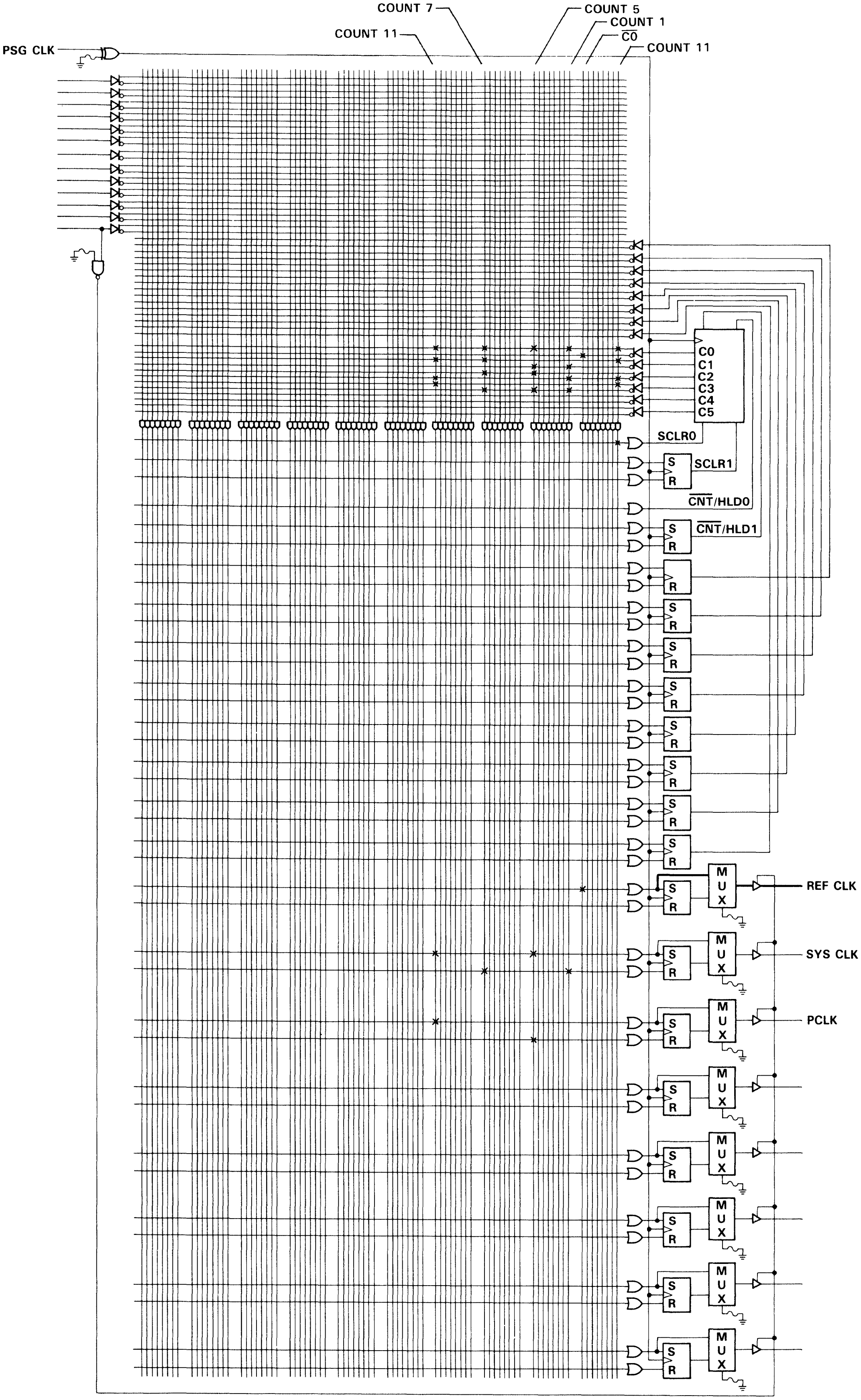


Figure 4. Waveform Generator
(Example 1)

Example 2: Refresh Timer

The second example demonstrates a design for a refresh timer used for signaling to a memory controller that it should execute a refresh cycle. As required by the dynamic memory, every row (256 on TMS4256) must be addressed once every 4 ms. One method used to guarantee that this requirement is met is to refresh one row at least once every 15.6 μs. With a 5 MHz system clock, the timer should be set for a division rate of approximately 77 clock cycles. This condition will generate a refresh request every 15.4 μs.

The memory controller executes the refresh request ($\overline{\text{REFREQ}}$) immediately if it is not involved in an access cycle. If the memory controller is executing an access cycle, then the refresh request will not be honored until the access cycle is completed. A refresh complete input (RFC) is required on the refresh timer to acknowledge when the refresh cycle has been completed by the memory controller. It is important that the timer does not stop, even though a refresh complete signal has not been received. This guarantees the refresh requirement is not violated. This also assumes the memory controller will complete the refresh request sometime in the next 77 clock cycles.

Figure 5 shows the timing diagram for the above application. A decimal count has been assigned to the PSG's master clock (PSG CLK) for reference. The counter is held at zero until the reset input is taken inactive low. Once the counter reaches 76 (equal to 77 clock cycles) the $\overline{\text{REFREQ}}$ output is driven active (low). The $\overline{\text{REFREQ}}$ output returns inactive high on the first positive clock edge after RFC goes active high. RFC is the signal from the memory controller that tells the refresh timer when the refresh operation has been completed. The $\overline{\text{REFREQ}}$ output remains low until the RFC signal has been received.

In order to generate a refresh request every 77 clock cycles, a 7-bit counter is required. Since the internal counter of the PSG is 6 bits, one of the state holding registers is required to expand the counter to 7 bits. As shown in Figure 6, only two product terms are required to expand to 7 bits; one product term to set the register when the 6-bit counter reaches its full count (63), and one product term to reset the register after count 76. Since both the binary counter and the added register need to be reset after count 76, a single product line can be used for both. (For additional details on expanding the 6-bit counter of the PSG, see the designer notes at the end of this application report.)

Figure 7 shows the fuse map for the entire refresh timer. The refresh timer is initialized by taking the RESET input high. When RESET is taken high, a single product line is activated and all other product lines are disabled. On the next active clock edge, the binary counter and C6 are cleared and the $\overline{\text{REFREQ}}$ output is set high. The refresh timer will begin counting when RESET returns low. When the 7-bit counter reaches 76, a product line goes active (high) and on the next clock edge forces C6 and the 6-bit counter to zero. Note that the output register holding $\overline{\text{REFREQ}}$ is also reset to zero. The RFC input is connected to a product line which in turn is connected to the set input of the $\overline{\text{REFREQ}}$ output register. On the next active clock edge after RFC is taken high the $\overline{\text{REFREQ}}$ output will return high.

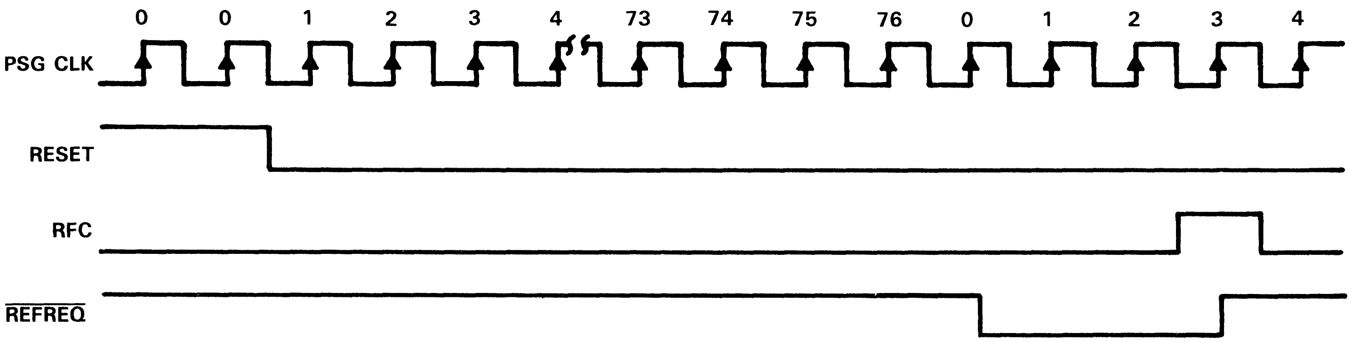


Figure 5. Refresh Timer Requirements (Example 2 — Refresh Timer)

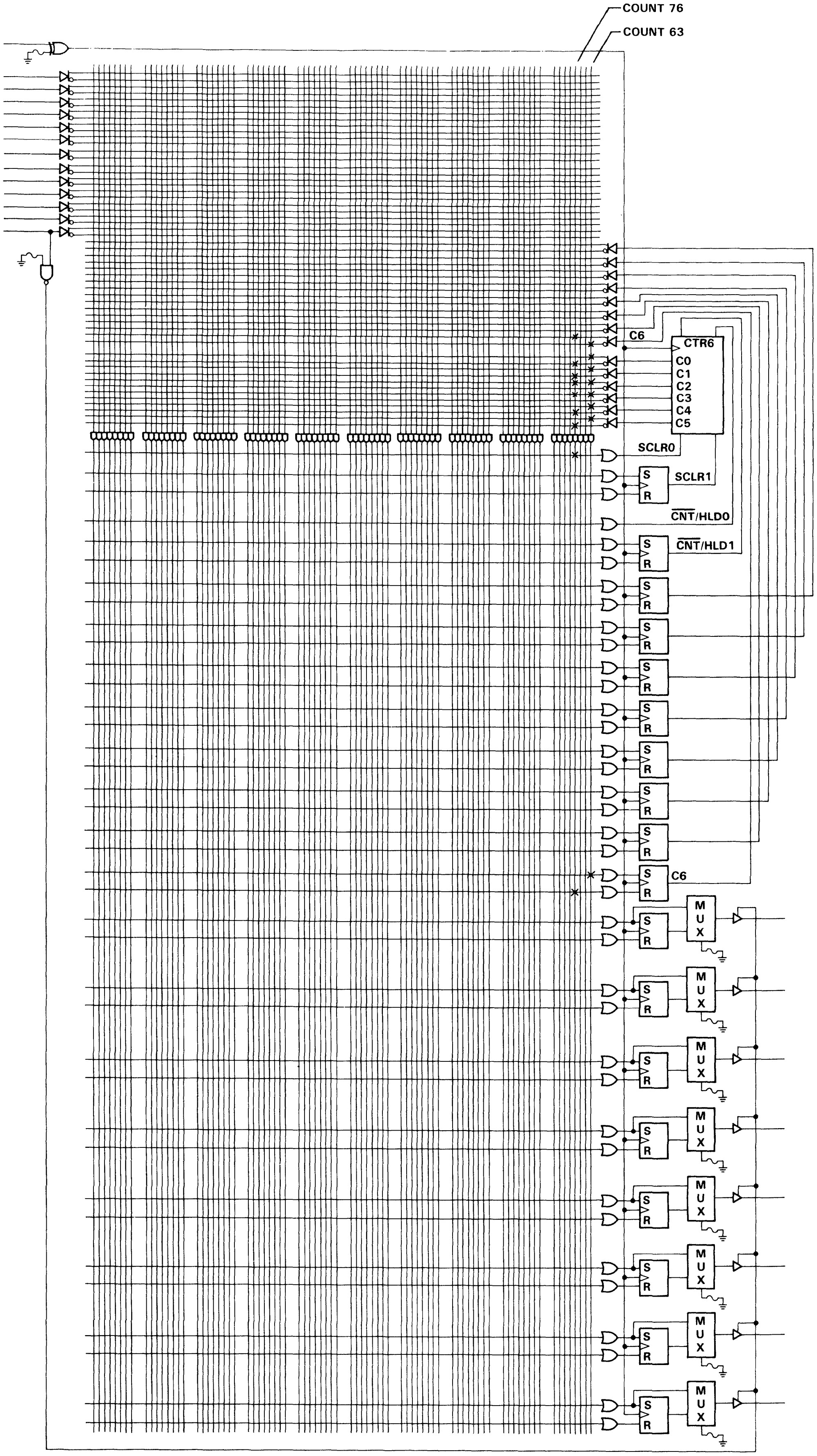


Figure 6. Expanding to 7-Bit Binary Counter (Example 2 — Refresh Timer)

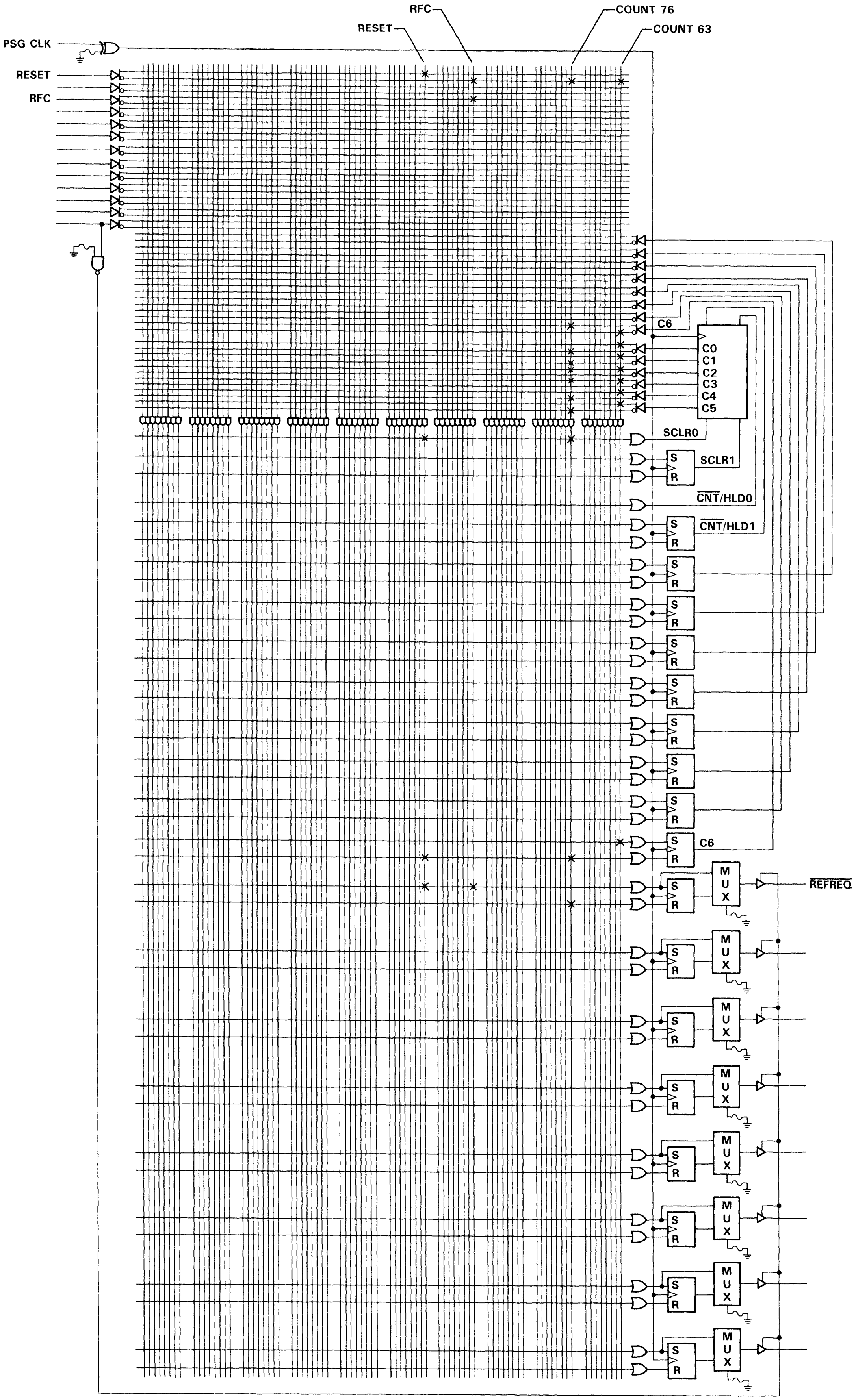


Figure 7. Refresh Timer (Example 2)

Example 3: Dynamic Memory Timing Controller

The third and last example will demonstrate a state machine design using the PSG507. Figure 8 shows the circuit requirement for a memory timing controller used for interfacing an Intel 8086 to an 'ALS2967 dynamic memory controller. Note that the clock generator and refresh timer, developed in Examples 1 and 2, can be used in this circuit.

The dynamic memory timing controller generates the control signals ($\overline{RAS}$, $\overline{CAS}$, MSEL, etc.) needed for accessing and refreshing the dynamic memory. The memory timing controller must also be capable of arbitrating between refresh and access cycles. In other words, if a refresh request ($\overline{REFREQ}$) occurs while the timing controller is performing an access cycle, the controller must finish the access cycle before granting the refresh request. Likewise, if an access cycle is requested during a refresh cycle, the controller must hold the processor while completing the refresh cycle. After the refresh cycle has been completed, the access cycle can be performed.

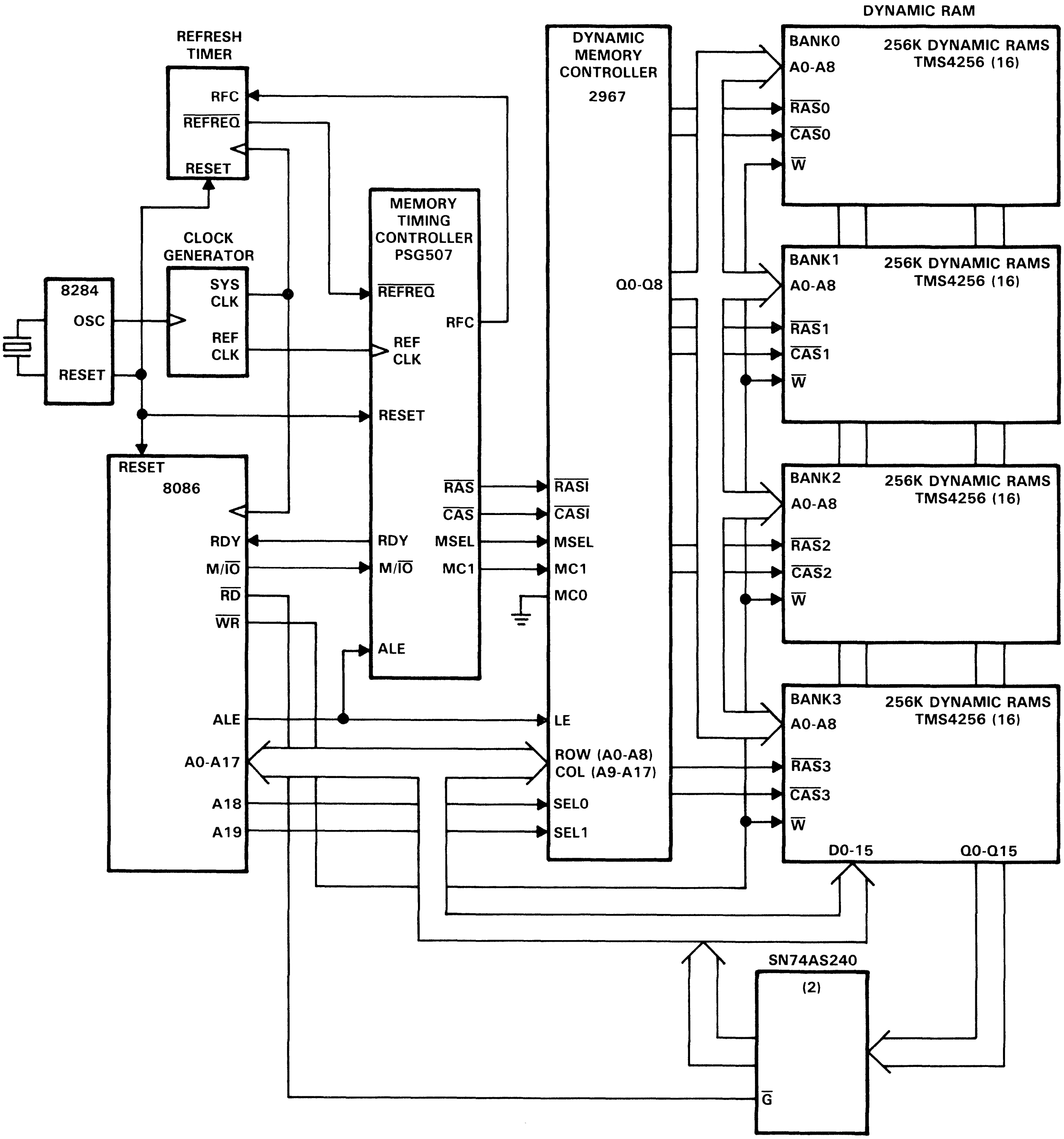


Figure 8. Memory Timing Controller (Example 3)

Figure 9 shows a detailed flow chart for the intended application. Note that two sequences are executed and three states are used. State 0 (ST0) provides an initalization and holding state, while state 1 (ST1) is assigned to the access sequence. The access sequence consists of 10 clock cycles as shown in Figure 10. State 2 (ST2) is assigned to the refresh/access grant sequence (Figure 11). This particular sequence takes 20 clock cycles, with a logical decision being made between count 9 and count 10. If at count 9 RDY is low, the counter will continue on and execute the access grant sequence. If RDY is high, the controller will clear the counter and return to state 0.

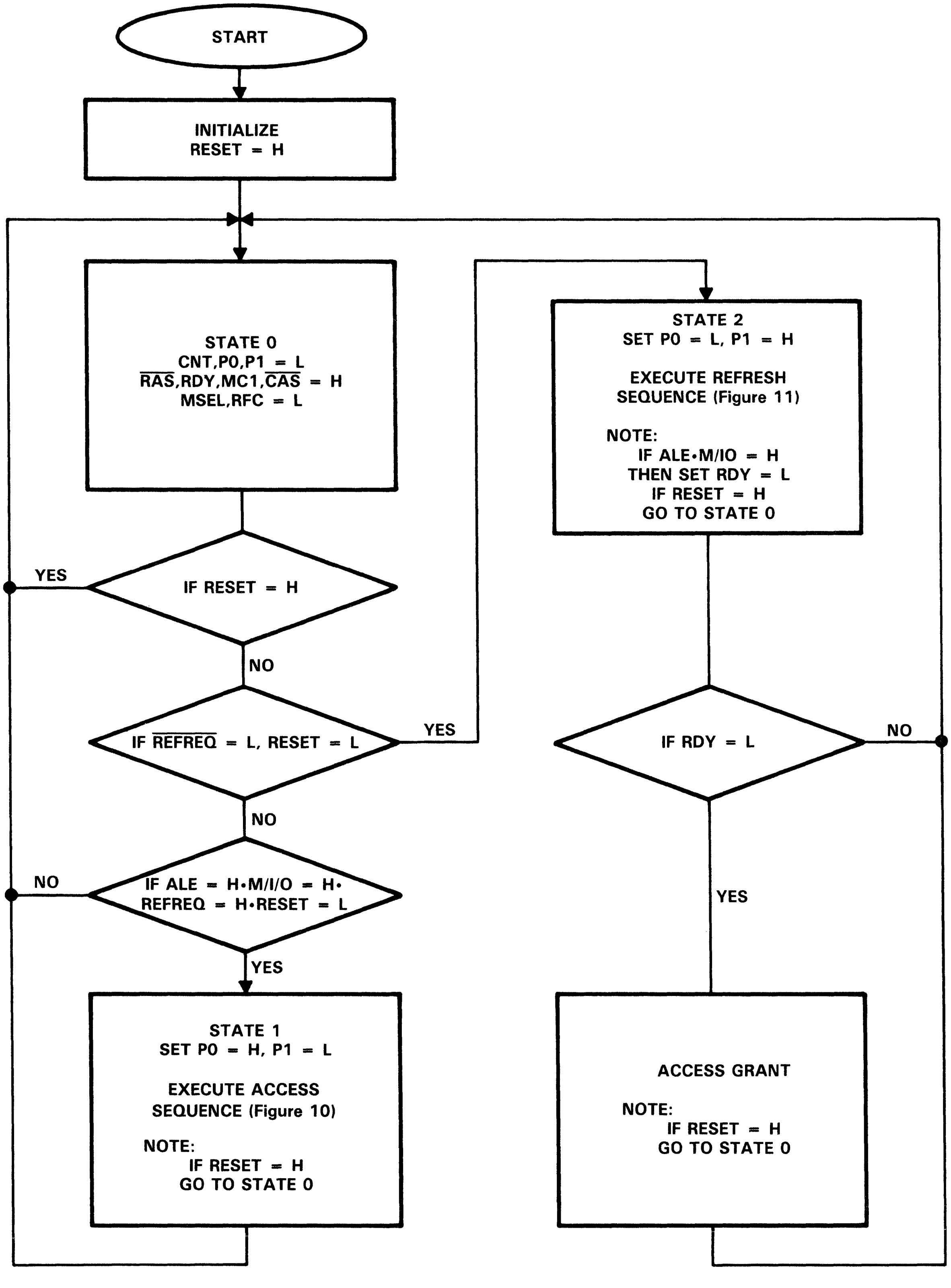
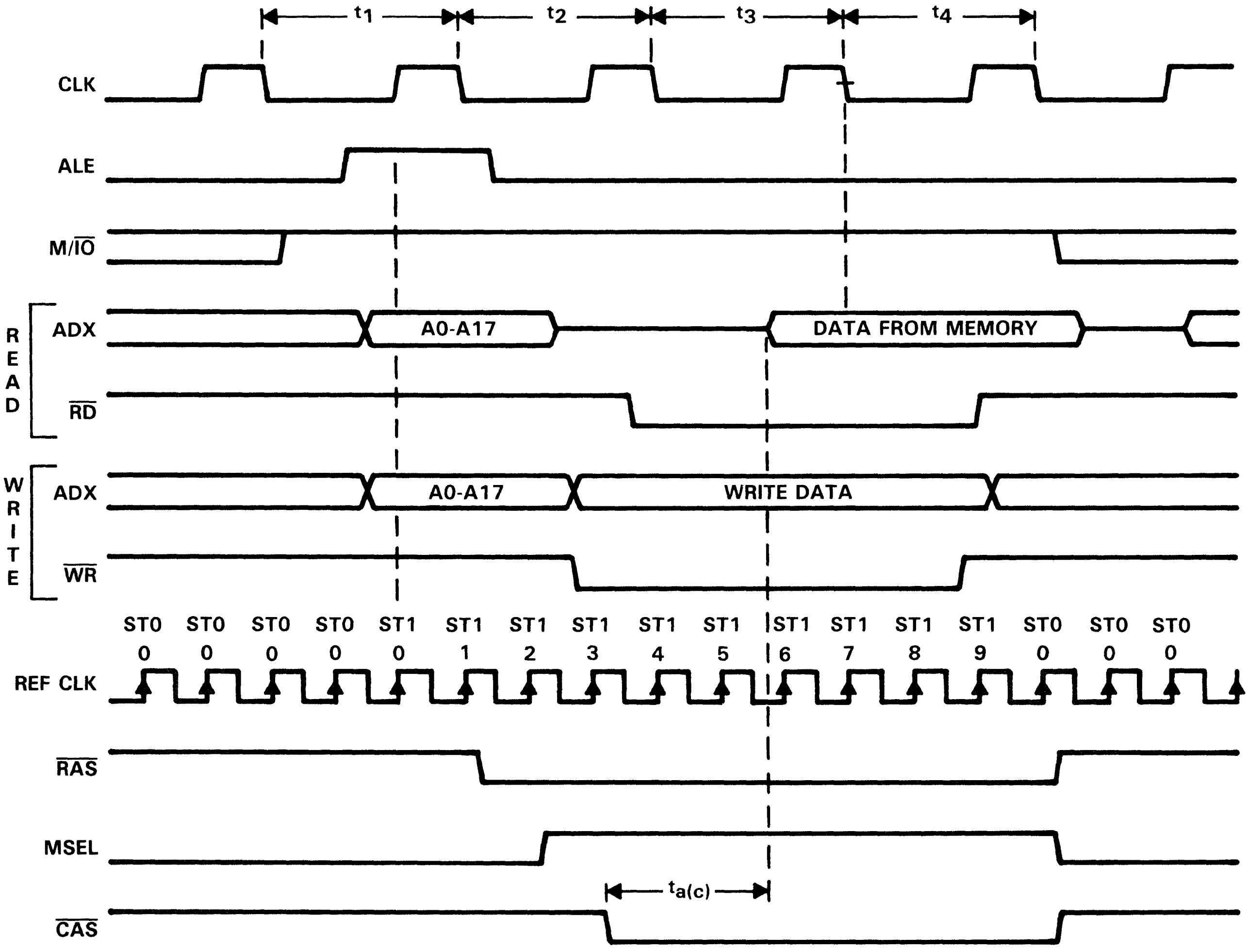


Figure 9. Flow Chart: Dynamic Memory Timing Controller



$\overline{\text{REFREQ}}$ = H, RESET = L
MC1 = H, RFC = L, RDY = H

Figure 10. Access Cycle

Developing the logic equations for this application becomes a simple matter when referencing the sequences to a decimal count (Figures 10 and 11). It is important to realize that each sequence has been referenced to a state. This allows the same binary counter to be used for each sequence, even though each sequence is of a different length.

The first step in implementing the above application is to define the logic equations which will make the binary counter perform as described in the flow chart of Figure 9. As will become evident, these equations fall directly from the flow chart. After the counter has been made to perform as described, the outputs can be easily decoded from the binary count and the present state of the state holding registers.

Figure 12 shows a fuse map for step 1 as described above. Initalization is performed by taking the reset input high. When this condition occurs, all product lines except the reset product line are forced inactive. When the reset product line is active, the counter and state holding registers (P0 and P1) are reset to zero on the first active clock edge. The $\overline{\text{CNT}}$/HLD1 register is set high, which places the counter in the hold mode. The RDY, MC1, $\overline{\text{RAS}}$, and $\overline{\text{CAS}}$ outputs are driven high on the same active clock edge. Since the RDY output does not feed back to the AND array, a buried state register, BRDY, is used to monitor the RDY output and is also set high. MSEL and RFC are driven low.

Controlling the binary counter is a simple matter and normally takes only a couple of logic equations. For each sequence, a start and stop condition must be defined. In the case of ST1, when the condition RESET = L, ALE = H, M/$\overline{\text{IO}}$ = H, $\overline{\text{REFREQ}}$ = H, P0 = L, and P1 = L occurs, ST0 (P1 = L, P0 = L) changes to ST1 (P1 = L, P0 = H), and the $\overline{\text{CNT}}$/HLD1 register is driven low to let the counter advance on the next active clock edge. When the counter reaches nine, ST1 returns to ST0 and the counter is cleared and put back into the hold condition.

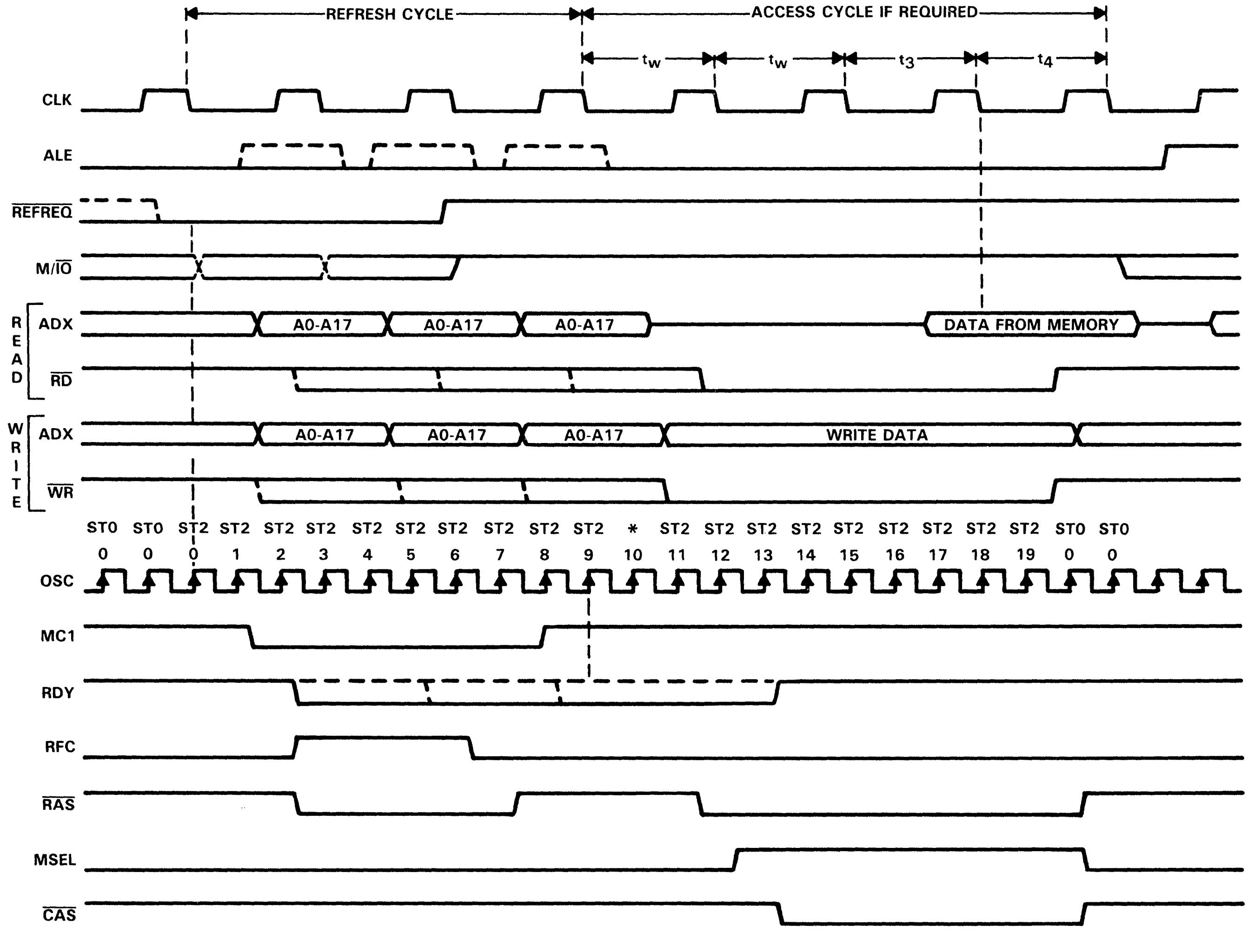
In the case of ST2, when the condition RESET = L, $\overline{\text{REFREQ}}$ = L, P0 = L, and P1 = L occurs, ST0 changes to ST2 (P1 = H, P0 = L) and the $\overline{\text{CNT}}$/HLD1 register is driven low to let the counter advance on the next active clock edge. As shown in the flow chart, if M/IO and ALE go high while in state 2, RDY and BRDY will be reset low on the next active clock edge. When the counter reaches nine, if RDY (BRDY) is high the state registers are returned to ST0 and the counter is cleared and placed back into the hold condition. If RDY (BRDY) is low, the counter advances on until it reaches 19. ST2 then returns to ST0 with the counter being cleared and placed back into the hold condition.

With the binary counter programmed to execute the flow chart in Figure 9, it is now a simple matter of decoding the outputs to perform as required in Figures 10 and 11. This is the same technique used in Example 1, except now a state has been assigned to each sequence. Below is a summary of the switching requirements for both the access (ST1) and the refresh sequence (ST2).

Access Sequence	
ST1 CNT 0:	Reset $\overline{RAS}$
ST1 CNT 1:	Set MSEL
ST1 CNT 2:	Reset $\overline{CAS}$
ST1 CNT 9:	Set $\overline{RAS}$, Reset MSEL, Set $\overline{CAS}$

Refresh Sequence	
ST2 CNT 0:	Reset MC1
ST2 CNT 1:	Set RFC, Reset $\overline{RAS}$
ST2 CNT 5:	Reset RFC
ST2 CNT 6:	Set $\overline{RAS}$
ST2 CNT 7:	Set MC1
ST2 CNT 10:	Reset $\overline{RAS}$
ST2 CNT 11:	Set MSEL
ST2 CNT 12:	Reset $\overline{CAS}$, Set RDY, Set BRDY
ST2 CNT 19:	Set $\overline{RAS}$, Reset MSEL, Set $\overline{CAS}$

Note that the transition changes are set up in the previous clock cycle, just as in Example 1. Figure 13 shows a complete fuse map for the memory controller.



*IF RDY = H, RETURN ST0

Figure 11. Refresh/Access Grant Cycle

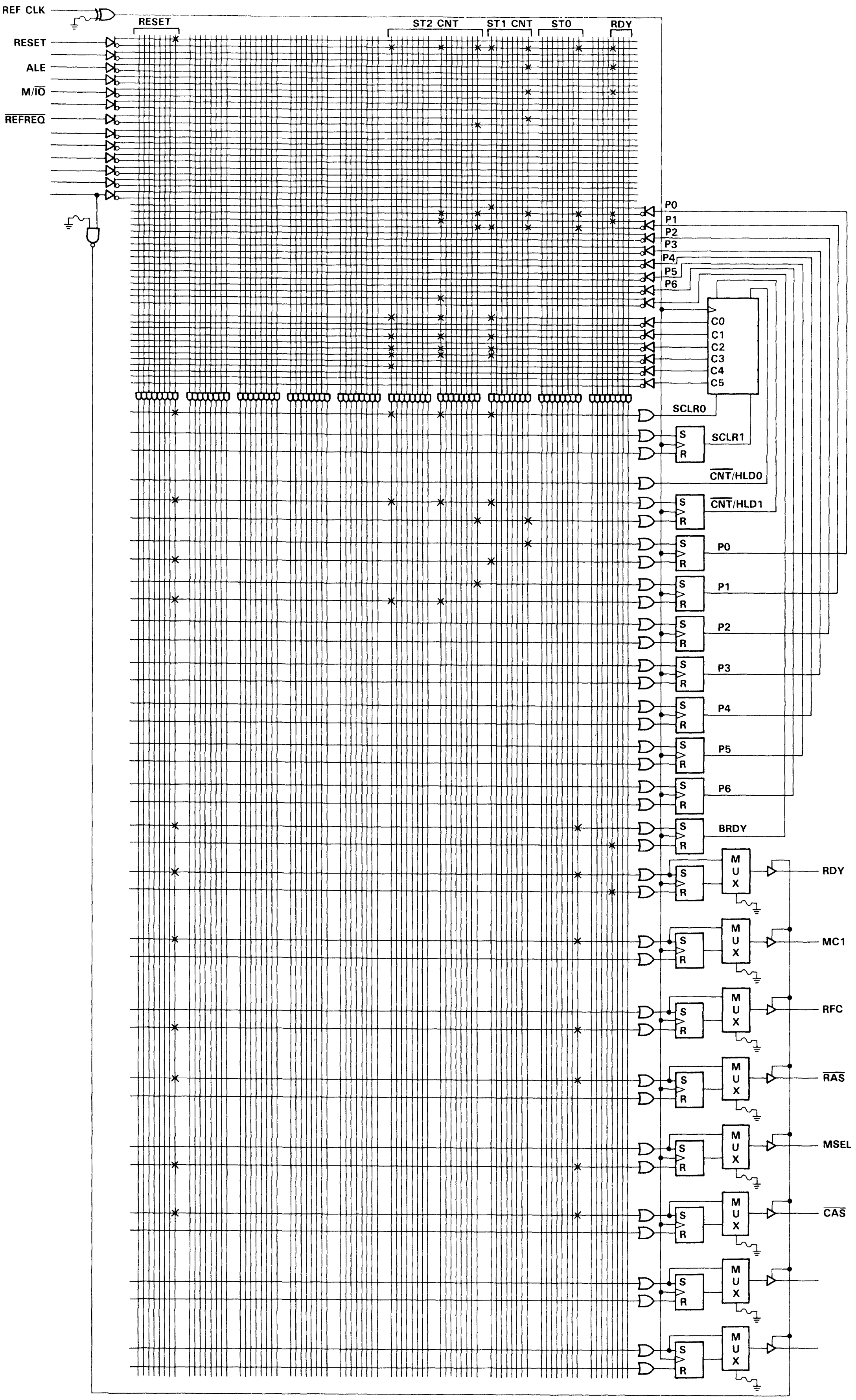


Figure 12. Counter Control Logic (Example 3 — Dynamic Memory Timing Controller)

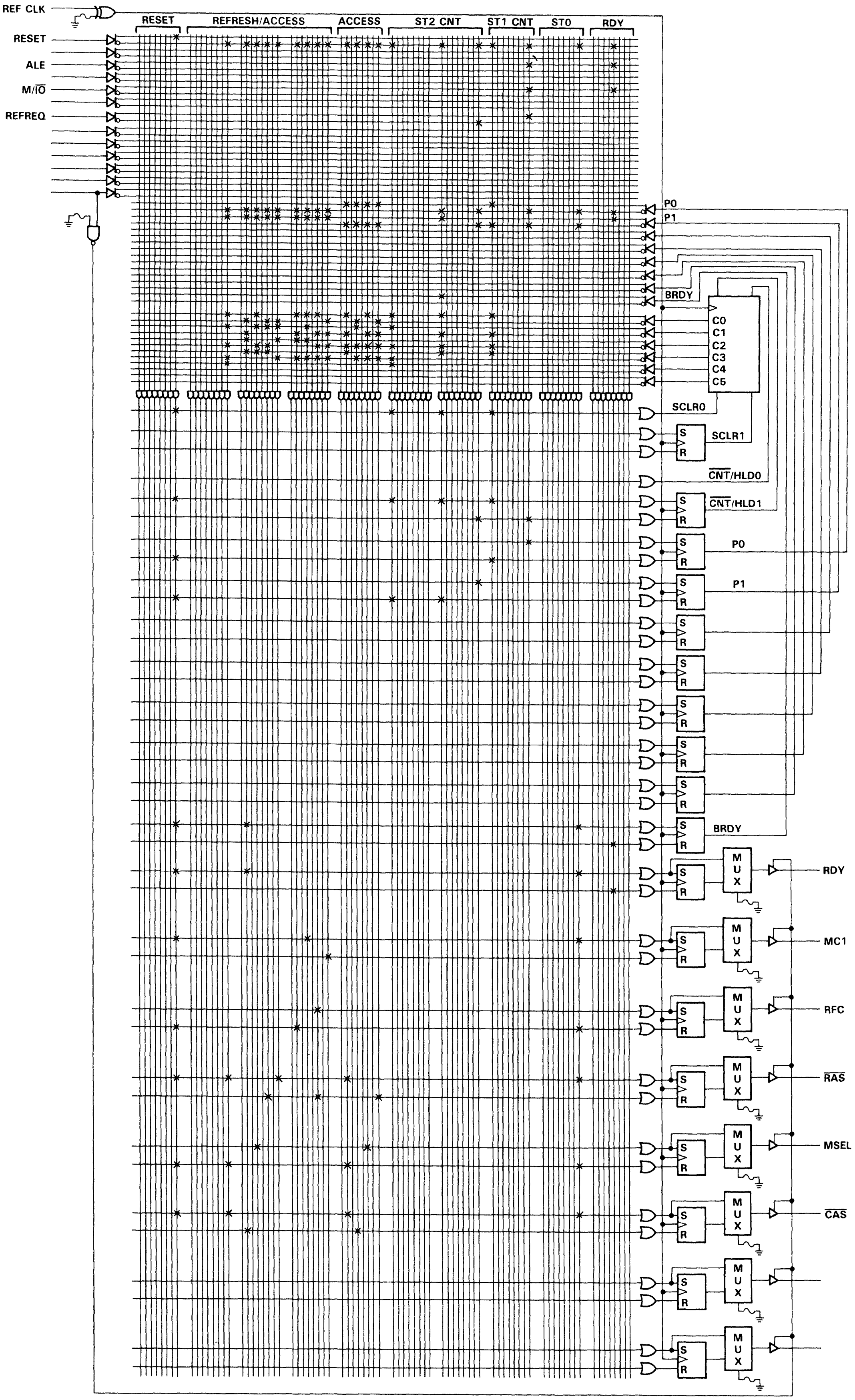


Figure 13. Memory Timing Controller (Example 3)

DESIGNER NOTES

Obtaining Maximum Counter Performance

As with any programmable logic device, there are usually several different methods for implementing any one application. In some cases, device performance is affected. On the PSG, maximum counter frequency is affected by how the designer controls the 6-bit counter.

For example, in the waveform generator example shown at the beginning of this application note, the counter was reset to zero after reaching count 11 by using the nonregistered SCLR0 function. By using the registered SCLR1 function, a higher operating frequency is obtainable.

This method requires an additional ''AND'' term as shown in Figure 14, but does provide maximum performance. Note that during the 10th clock cycle the set input on the SCLR1 register is high. On the next active clock edge, the counter advances to 11 and the SCLR1 register is set high. This causes the counter to be reset on the next active clock edge. At the same time, the SCLR1 register is reset low to allow the counter to advance past zero.

In effect, the setup time requirement for SCLR1 is performed in the previous clock cycle. When using the SCLR0 method, the setup time must be added to the f_{max} equation. This results in a lower f_{max}. The same tradeoffs apply with the $\overline{\text{CNT}}$/HLD function. The PSG507 data sheet specifies f_{max} for both methods.

Expanding the 6-Bit Counter

In Example 2, the six bit counter had to be expanded to 7 bits. This was accomplished by adding one of the state registers to the most significant bit of the counter. It should be noted that the synchronous clear and count hold functions must be controlled through the set and reset inputs of the added bits. The designer must be aware of certain limitations when trying to perform this function. Figure 15 shows three additional bits being added to the 6-bit counter. Note that every bit added requires two additional ''AND'' terms.

A problem can arise on certain counts when trying to generate a synchronous clear before reaching the full binary count (all outputs high). The designer must ensure that both S and R are not high simultaneously. For example, let's say we want the 9-bit counter to return to zero at count 383 (101111111_2). At count 383, the S/R register used for C7 is being told to set. Therefore, any reset command would result in both S and R being high simultaneously.

This problem, only seen on a few data words, can be solved by using another state register to control the counter reset. This method is similar to that used above to obtain maximum operating frequency. Figure 16 shows the 9-bit counter returning to zero after count 383. Notice that at count 382 the extra S/R register is being told to reset on the next active clock edge. At count 383 the six product lines controlling C6, C7, and C8 are disabled by the feedback from the extra register, in particular the S input on C7. At count 383, the 9-bit counter will return to zero and the extra register is set high.

An extra register may also be needed to achieve the count/hold function when using an expanded counter. During certain counts the added bits will change state, even though the 6-bit counter is programmed to hold. For example, let's say we want the 9-bit counter to hold at count 383. Even though the 6-bit counter can be held at 111111, C6 and C7 will advance on the next active clock edge. In order to hold C6 and C7 where they are, an extra register is used to disable the product lines responsible for the transition from count 383 to 384. Since the counter is on hold, the extra hold register can only be reset from an input pin or a state register(s) transition (not on the next count). In this example, an input pin is used to reset the extra register and the $\overline{\text{CNT}}$/HOLD register. When the CONTINUE input is taken low, the counter will continue to advance. The system must guarantee that the continue input will not be low during count 382 to avoid the indeterminant set = H, reset = H state. Figure 17 shows this 9-bit counter.

It is also important to note that when using extra registers a reset input may be necessary to set the extra registers high after powerup, since all S/R registers power-up clear. This requirement would not be necessary if the phase of the extra register was reversed. This is easily accomplished by using the inverted feedback from the extra register. However, it is good state machine design practice to include a reset input that forces all S/R registers to a known state.

Software Support

The PSG507 is supported by two software packages; CUPL, which was created by and is supported by Assisted Technologies, a division of Personal CAD Systems Incorporated, and ABEL, which was created by and is supported by FutureNet, a division of Data I/O Corporation. Each of these software packages can be used to reduce equations and to generate a fuse map necessary to program the PSG507. Appendices A and B show the ABEL and CUPL files for Examples 1, 2, and 3. In addition, a PSG507 template is shown for each software package. These templates provide software information that will make it easier for the designer to create the source files.

Test vectors are included with the ABEL and CUPL source files so software simulation can be performed on the computer. If the proper instruction is provided, the software will attach the test vectors to the end of the fuse map. This allows programming equipment to run a functional test on each device immediately after programming.

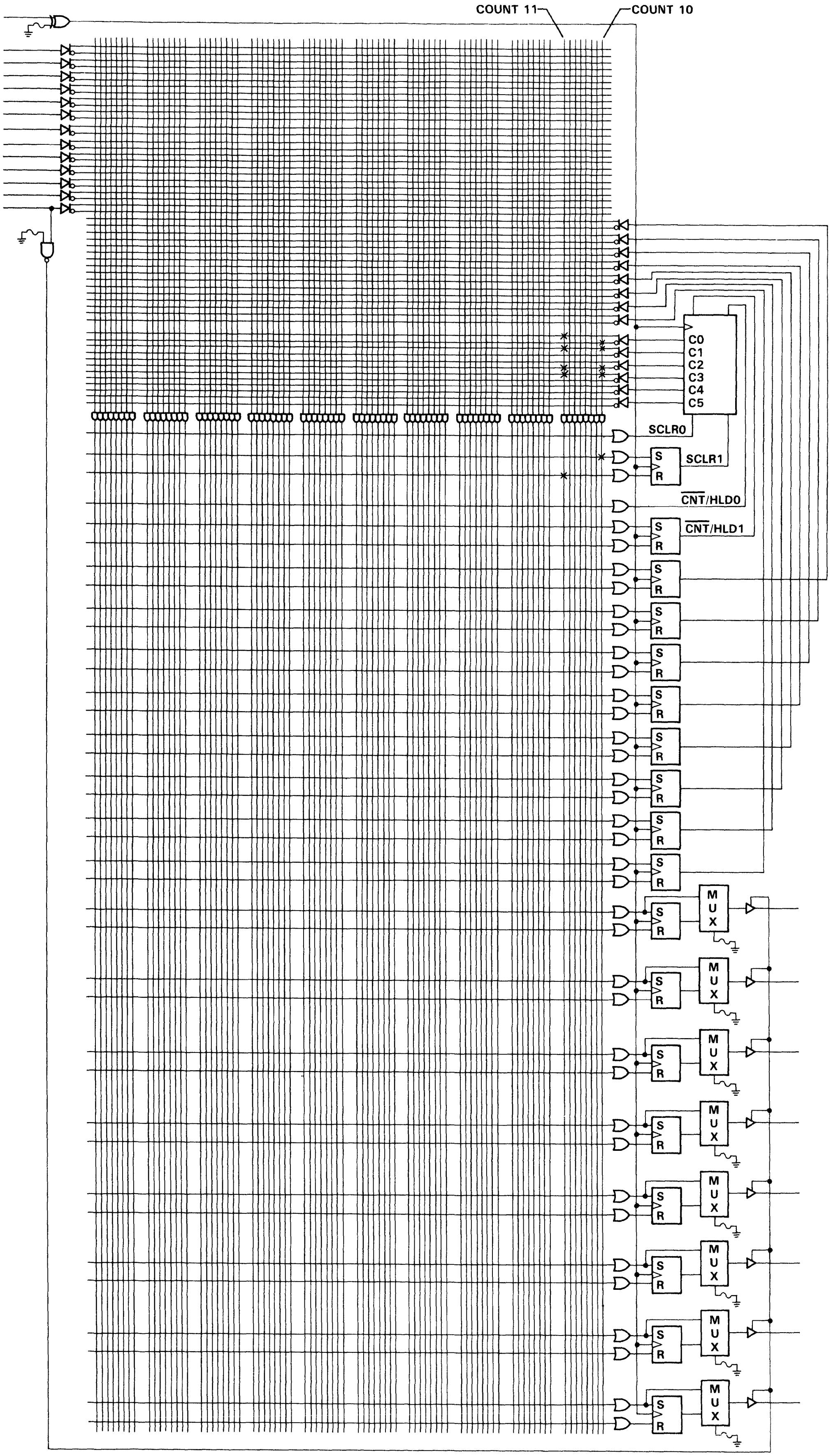


Figure 14. Registered $\overline{\text{SCLR}}$ Example
(Designer Notes)

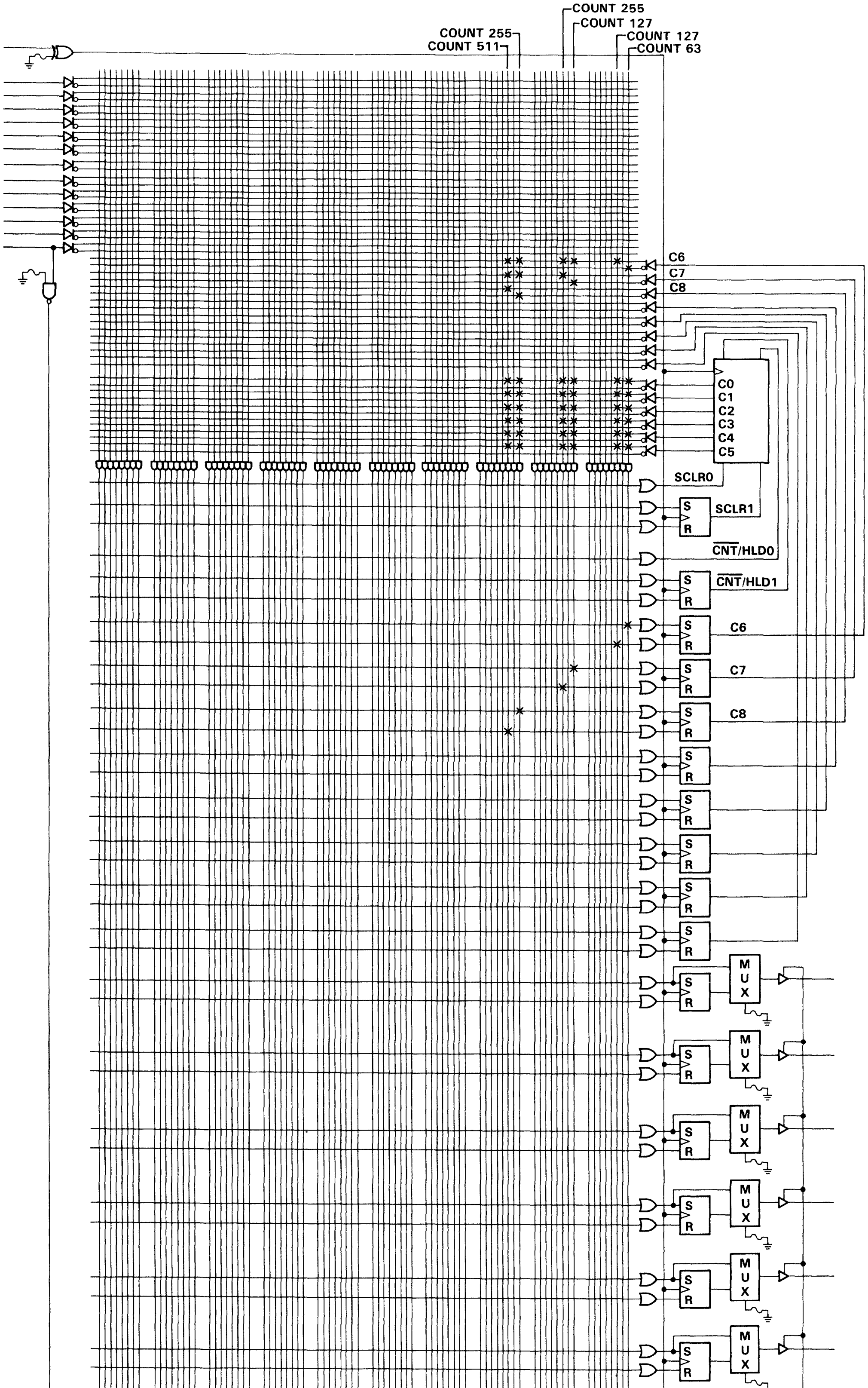


Figure 15. Expanding to 9-Bit Counter

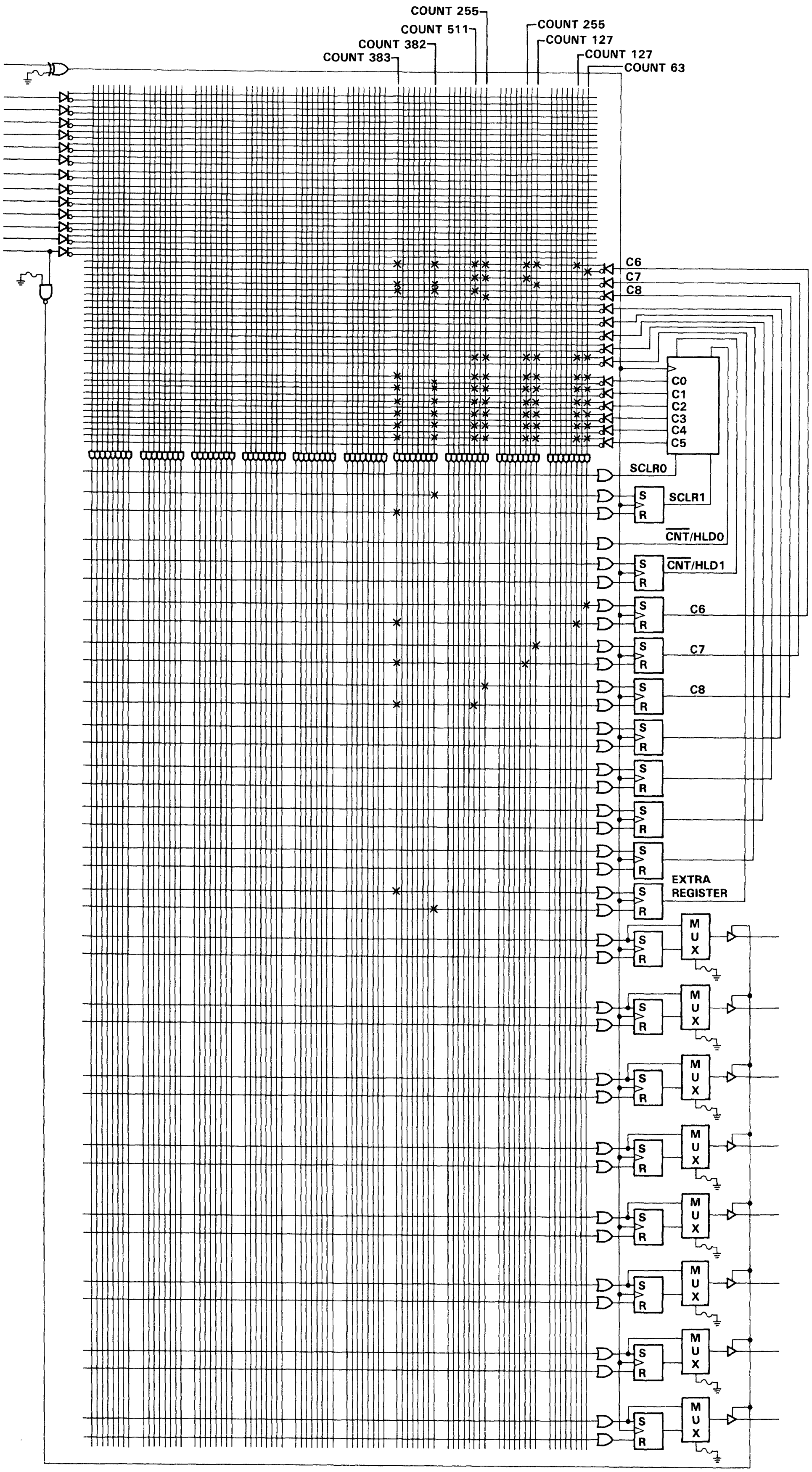


Figure 16. Resetting after Count 383 (Expanding the 6-Bit Counter)

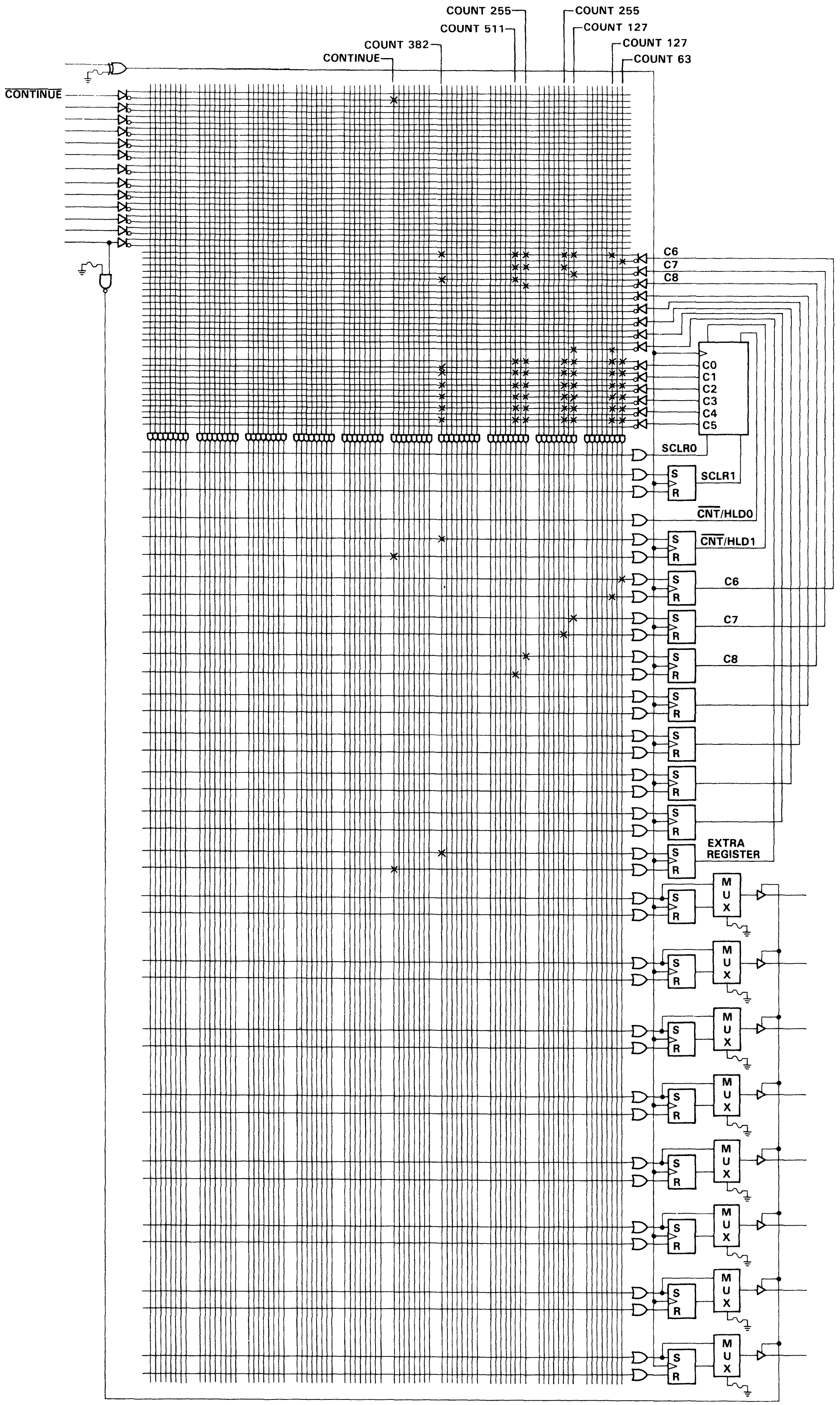


Figure 17. Holding the 9-Bit Counter at Count 383 (Expanding the 6-Bit Counter)

'PSG507 Example 1: Waveform Generator

```
title {

     Device:        TIBPSG507

     Application:   PSG507 Example 1:  Waveform Generator.

     Source:        Breuninger, R. K. "A Designer's Guide to the
                    TIBPSG507," Programmable Logic Data Book, 3-39,
                    Texas Instruments, 1988.

     Transcription: INLAB Inc.,
                    250-I West 6th Avenue
                    Broomfield, CO 80020
                    303-460-0103
}

include a507;

/* counter states of interest */
define COUNT1 =pt00;
               pt00 = !c3 & !c2 & !c1 &  c0;
define COUNT5 =pt01;
               pt01 = !c3 &  c2 & !c1 &  c0;
define COUNT7 =pt02;
               pt02 = !c3 &  c2 &  c1 &  c0;
define COUNT11=pt03;
               pt03 =  c3 & !c2 &  c1 &  c0;

/* ref_clk */  pin8   = pt04;
                        pt04 = !c0;

/* sys_clk */  pin9.s = COUNT5 | COUNT11;
               pin9.r = COUNT1 | COUNT7;

/* pclk */     pin10.s= COUNT11;
               pin10.r= COUNT5;

   sclr0              = COUNT11;
                              /* synchronous clear on count 11 */

test_vectors { /* psg_clk    count     ref_clk  sys_clk  pclk  */

                  pin1                 pin8      pin9   pin10;

                    0      /*  0 */     H         L       L  ;
                    C      /*  1 */     L         L       L  ;
                    C      /*  2 */     H         L       L  ;
                    C      /*  3 */     L         L       L  ;
                    C      /*  4 */     H         L       L  ;
                    C      /*  5 */     L         L       L  ;
                    C      /*  6 */     H         H       L  ;
                    C      /*  7 */     L          H        L  ;
                    C      /*  8 */     H          L        L  ;
                    C      /*  9 */     L          L        L  ;
                    C      /* 10 */     H          L        L  ;
                    C      /* 11 */     L          L        L  ;

                    C      /*  0 */     H          H        H  ;
                    C      /*  1 */     L          H        H  ;
                    C      /*  2 */     H          L        H  ;
                    C      /*  3 */     L          L        H  ;
                    C      /*  4 */     H          L        H  ;
                    C      /*  5 */     L          L        H  ;
                    C      /*  6 */     H          H        L  ;
                    C      /*  7 */     L          H        L  ;
                    C      /*  8 */     H          L        L  ;
                    C      /*  9 */     L          L        L  ;
                    C      /* 10 */     H          L        L  ;
                    C      /* 11 */     L          L        L  ;

                    C      /*  0 */     H          H        H  ;
}
```

’PSG507 Example 2: Refresh Timer

```
title {

     Device:         TIBPSG507

     Application:    PSG507 Example 2:  Refresh Timer.

     Source:         Breuninger, R. K. ″A Designer’s Guide to the
                     TIBPSG507,″ Programmable Logic Data Book, 3-42,
                     Texas Instruments, 1988.

     Transcription: INLAB Inc.,
                     250-I West 6th Avenue
                     Broomfield, CO 80020
                     303-460-0103
}

include a507;

/* input signals */
define  reset   = pin2;  /* inactive low */
define  RESET   =        pt00;
                         pt00 = reset;

define  rfc     = pin3;  /* active high refresh complete */
define  RFC     =        pt01;
                         pt01 = rfc;

/* the seventh counter bit */
define  c6      = p0;

/* counter states of interest */
define COUNT63  = pt02;
     pt02 = !c6.q &  c5 &  c4 &  c3 &  c2 &  c1 &  c0 & !reset;
define COUNT76  = pt03;
     pt03 =  c6.q & !c5 & !c4 &  c3 &  c2 & !c1 & !c0 & !reset;

/* counter control */
   c6.s         = COUNT63;
   c6.r         = COUNT76 | RESET;
   sclr0        = COUNT76 | RESET;

/* refreq */
define refreq = pin8;    refreq.s = RFC | RESET;
                         refreq.r = COUNT76;

test_vectors {

                 pin1  reset  rfc  refreq;

                  0      1     0      L;     /* power on */
  repeat  4 {     C      1     0      H;   } /* reset */
  repeat 76 {     C      0      0      H;    }
                  C      0      0      L;
                  C      0      0      L;
                  C      0      1      H;
  repeat 74 {     C      0      0      H;    }
                  C      0      0      L;
                  C      0      0      L;
                  C      0      1      H;
}
```

'PSG507 Example 3: Dynamic Memory Timing Controller

```
title {

     Device:         TIBPSG507

     Application:    PSG507 Example 3:  Dynamic Memory Timing
                     Controller.

     Source:         Breuninger, R. K. "A Designer's Guide to the
                     TIBPSG507," Programmable Logic Data Book, 3-45,
                     Texas Instruments, 1988.

     Transcription:  INLAB Inc.,
                     250-I West 6th Avenue
                     Broomfield, CO 80020
                     303-460-0103
}

include a507;

/* input signals */
define  reset   = pin2;  /* inactive low */
define  ale     = pin3;  /* address latch enable */
define  mio     = pin4;  /* memory I/O */
define  refreq  = pin5;  /* refresh request */

/* output signals */
define  rdy     = pin8;  /* ready */
define  mcl     = pin9;  /* mode control */
define  rfc     = pin10; /* refresh complete */
define  ras     = pin11; /* row address strobe */
define  msel    = pin13; /* multiplexer select */
define  cas     = pin14; /* column address strobe */

/* internal */
define  brdy    = p2;    /* buried ready - always identical to
                            output signal 'rdy'.  Permits testing
                            the output pin state. */

/* counter states of interest */
define COUNT0  = !c4 & !c3 & !c2 & !c1 & !c0;
define COUNT1  = !c4 & !c3 & !c2 & !c1 &  c0;
define COUNT2  = !c4 & !c3 & !c2 &  c1 & !c0;
define COUNT3  = !c4 & !c3 & !c2 &  c1 &  c0;
define COUNT5  = !c4 & !c3 &  c2 & !c1 &  c0;
define COUNT6  = !c4 & !c3 &  c2 &  c1 & !c0;
define COUNT9  = !c4 &  c3 & !c2 & !c1 &  c0;
define COUNT10 = !c4 &  c3 & !c2 &  c1 & !c0;
define COUNT11 = !c4 &  c3 & !c2 &  c1 &  c0;
define COUNT12 = !c4 &  c3 &  c2 & !c1 & !c0;
define COUNT19 =  c4 & !c3 & !c2 &  c1 &  c0;
```

```
/* LOW and HIGH operations for clarity */
define LOW  = .r=1; /* usage: (rs LOW) -> (rs.r=1) */
define HIGH = .s=1;

state_diagram (p1.q,p0.q) {

     if (reset) {
          /* These are the levels of all output and control
        signals in the idle state.  Other states return
             modified signals to these levels before resuming the
             idle state. */

          mc1 HIGH; rdy HIGH; rfc  LOW; ras HIGH; msel LOW;
                    cas HIGH; brdy HIGH;
          sclr0=1;  hld1 HIGH; /* counter cleared and holding */
          idleState;
     }

     state idleState=00 {
          /* Wait for Request */
          if (ale & mio & refreq) {
               /* Memory Access Request */
               hld1 LOW;
               accessCycle;
          }
          if (!refreq) {
               /* Memory Refresh Request */
               hld1 LOW;
               refreshCycle;
          }
     }

     state accessCycle=01 {
          /* Generate the Memory Access Sequence. */
          if (COUNT0)
               ras LOW;
          if (COUNT1)
               msel HIGH;
          if (COUNT2)
               cas LOW;
          if (COUNT9) {
               /* Return modified control and output
                  signals to their idle values. */
               ras HIGH; msel LOW; cas HIGH;
               sclr0=1;  hld1 HIGH;
               idleState;
          }
     }
     state refreshCycle=1x {
          /* Generate the Memory Refresh Sequence. */
          if (ale & mio) {
               /* An Access Request occurs during refresh.  Hold
                  off the processor until refresh is complete (at
                  COUNT9 below). */
               rdy  LOW;
               brdy LOW;
          }
          if (COUNT0)
               mc1 LOW;
          if (COUNT1) {
               rfc HIGH; ras LOW;
          }
          if (COUNT3)
               mc1 HIGH;
          if (COUNT5)
               rfc LOW;
          if (COUNT6)
               ras HIGH;
          if (COUNT9)
               /* The Refresh Sequence is complete. */
               if (brdy.q) {
                    /* The processor did NOT make a Memory Access
                       request during the Refresh Sequence.
                       Return to idle. */
                    sclr0=1; hld1 HIGH;
                    idleState;
               }
          /* A Memory Access Request was received during the
             Refresh Sequence.  Generate the Memory Access
             Sequence now. */
          if (COUNT10)
               ras LOW;
          if (COUNT11)
               msel HIGH;
          if (COUNT12) {
               rdy  HIGH; cas LOW;
               brdy HIGH;
          }
          if (COUNT19) {
               ras HIGH; msel LOW; cas HIGH;
               sclr0=1;  hld1 HIGH;
               idleState;
          }
     }
}

test_vectors {

               /* Access Cycle */

pin1 reset ale mio refreq rdy mc1 rfc ras msel cas;
```

```
 0    0     0   0    1     L   L   L   L   L    L ; /* power on*/

 C    1     0   0    1     H   H   L   H   L    H ; /* reset */
 C    0     0   0    1     H   H   L   H   L    H ;

 C    0     0   0    1     H   H   L   H   L    H ;
 C    0     0   1    1     H   H   L   H   L    H ;
 C    0     1   1    1     H   H   L   H   L    H ; /*  0 */
 C    0     0   1    1     H   H   L   L   L    H ; /*  1 */
 C    0     0   1    1     H   H   L   L   H    H ; /*  2 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  3 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  4 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  5 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  6 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  7 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  8 */
 C    0     0   1    1     H   H   L   L   H    L ; /*  9 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  0 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  0 */

          /* Refresh Cycle

pin1 reset ale mio refreq rdy mc1 rfc ras msel cas        */

 C    0     0   0    0     H   H   L   H   L    H ; /*  0 */
 C    0     0   1    0     H   L   L   H   L    H ; /*  1 */
 C    0     0   1    0     H   L   H   L   L    H ; /*  2 */
 C    0     0   1    0     H   L   H   L   L    H ; /*  3 */
 C    0     0   1    0     H   H   H   L   L    H ; /*  4 */
 C    0     0   1    0     H   H   H   L   L    H ; /*  5 */
 C    0     0   1    1     H   H   L   L   L    H ; /*  6 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  7 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  8 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  9 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  0 */
 C    0     0   1    1     H   H   L   H   L    H ; /*  0 */

          /* Refresh/Access Grant Cycle

pin1 reset ale mio refreq rdy mc1 rfc ras msel cas        */

 C    0     0   0    0     H   H   L   H   L    H ; /*  0 */
 C    0     0   1    0     H   L   L   H   L    H ; /*  1 */
 C    0     0   1    0     H   L   H   L   L    H ; /*  2 */
 C    0     1   1    0     L   L   H   L   L    H ; /*  3 */
 C    0     0   1    0     L   H   H   L   L    H ; /*  4 */
 C    0     0   1    0     L   H   H   L   L    H ; /*  5 */
 C    0     0   1    1     L   H   L   L   L    H ; /*  6 */
 C    0     0   1    1     L   H   L   H   L    H ; /*  7 */
 C    0     0   1    1     L   H   L   H   L    H ; /*  8 */
 C    0     0   1    1     L   H   L   H   L    H ; /*  9 */
 C    0     0   1    1     L   H   L   H   L    H ; /* 10 */
 C     0    0   1    1     L   H   L   L   L    H ; /* 11 */
 C     0    0   1    1     L   H   L   L   H    H ; /* 12 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 13 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 14 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 15 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 16 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 17 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 18 */
 C     0    0   1    1     H   H   L   L   H    L ; /* 19 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  0 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  0 */
}
```

Appendix B

TI Device Cross Reference

TI DEVICE CROSS REFERENCE

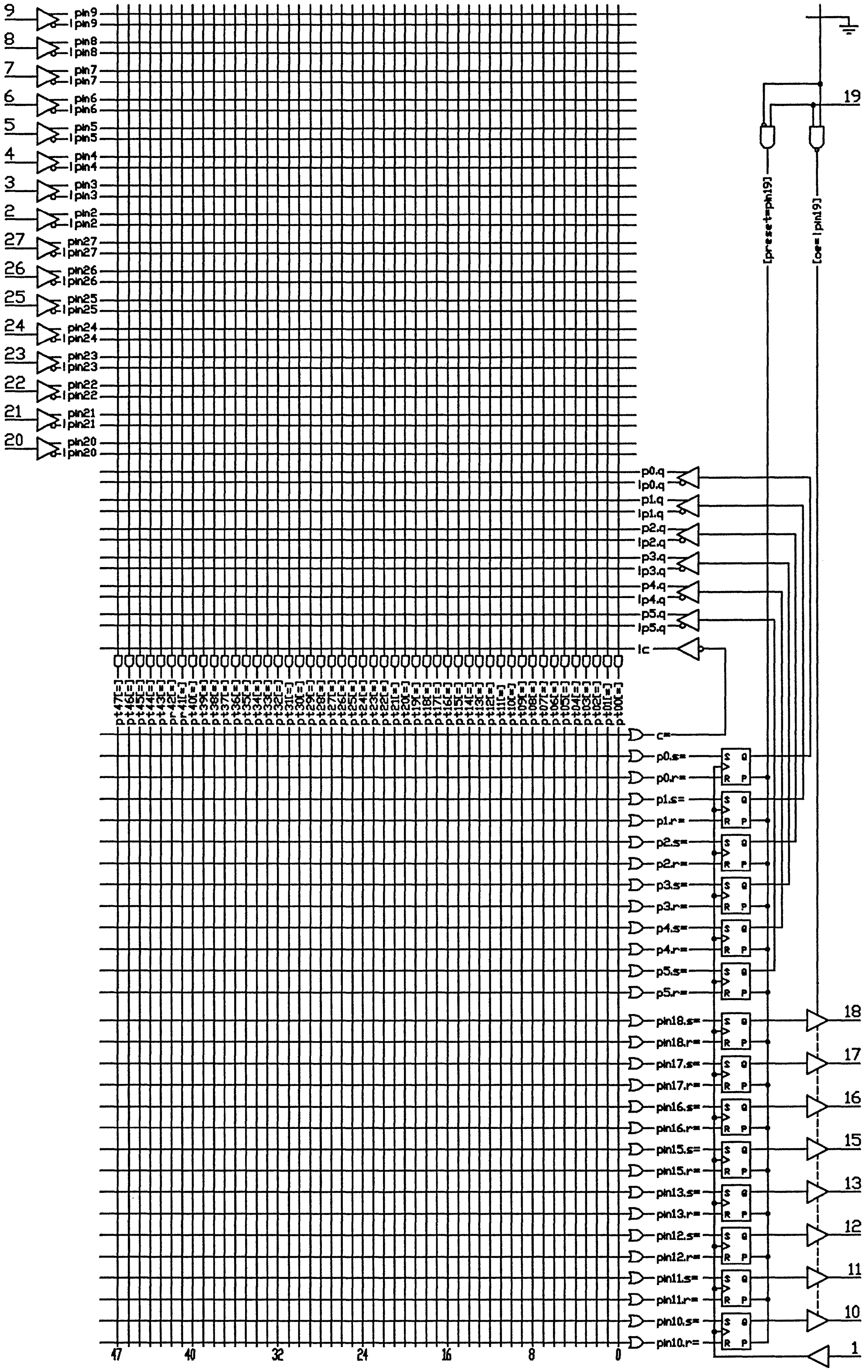
TI Device Name	proLogic Name	Page
PAL16L8A/A–2	P16L8	C–15
PAL16R4A/A–2	P16R4	C–19
PAL16R6A/A–2	P16R6	C–20
PAL16R8A/A–2	P16R8	C–21
PAL20L8A	P20L8	C–24
PAL20R4A	P20R4	C–26
PAL20R6A	P20R6	C–27
PAL20R8A	P20R8	C–28
TIBPAD16N8–7	P16N8	C–16
TIBPAD18N8–6	P18N8	C–22
TIBPAL16L8	P16L8	C–15
TIBPAL16R4	P16R4	C–19
TIBPAL16R6	P16R6	C–20
TIBPAL16R8	P16R8	C–21
TIBPAL20L8	P20L8	C–24
TIBPAL20R4	P20R4	C–26
TIBPAL20R6	P20R6	C–27
TIBPAL20R8	P20R8	C–28
TIBPAL20L10	P20L10	C–25
TIBPAL20X4	P20X4	C–29
TIBPAL20X8	P20X8	C–30
TIBPAL20X10	P20X10	C–31
TIBPAL22V10	P22V10	C–32
TIBPAL22VP10	P22VP10	C–34

TI DEVICE CROSS REFERENCE (continued)

TI Device Name	proLogic Name	Page
TIB82S105B	A105B	C–3
TIB82S167B	A167B	C–5
TIBPLS506	A506	C–7
TIBPSG507	A507	C–11
TICPAL16L8	P16L8	C–15
TICPAL16R4	P16R4	C–19
TICPAL16R6	P16R6	C–20
TICPAL16R8	P16R8	C–21
TICPAL22V10	P22V10	C–32
TIEPAL10H16P8	P16P8E	C–34

Appendix C

Logic Diagrams


9
8
7
6
5
4
3
2
27
26
25
24
23
22
21
20
pin9
!pin9
pin8
!pin8
pin7
!pin7
pin6
!pin6
pin5
!pin5
pin4
!pin4
pin3
!pin3
pin2
!pin2
pin27
!pin27
pin26
!pin26
pin25
!pin25
pin24
!pin24
pin23
!pin23
pin22
!pin22
pin21
!pin21
pin20
!pin20
19
[preset=pin19]
[oe=!pin19]
p0.q
!p0.q
p1.q
!p1.q
p2.q
!p2.q
p3.q
!p3.q
p4.q
!p4.q
p5.q
!p5.q
!c
pt47[=]
pt46[=]
pt45[=]
pt44[=]
pt43[=]
pr42[=]
pr41[=]
pt40[=]
pt39[=]
pt38[=]
pt37[=]
pt36[=]
pt35[=]
pt34[=]
pt33[=]
pt32[=]
pt31[=]
pt30[=]
pt29[=]
pt28[=]
pt27[=]
pt26[=]
pt25[=]
pt24[=]
pt23[=]
pt22[=]
pt21[=]
pt20[=]
pt19[=]
pt18[=]
pt17[=]
pt16[=]
pt15[=]
pt14[=]
pt13[=]
pt12[=]
pt11[=]
pt10[=]
pt09[=]
pt08[=]
pt07[=]
pt06[=]
pt05[=]
pt04[=]
pt03[=]
pt02[=]
pt01[=]
pt00[=]
c=
p0.s=
p0.r=
p1.s=
p1.r=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
pin18.s=
pin18.r=
pin17.s=
pin17.r=
pin16.s=
pin16.r=
pin15.s=
pin15.r=
pin13.s=
pin13.r=
pin12.s=
pin12.r=
pin11.s=
pin11.r=
pin10.s=
pin10.r=
S
Q
R
P
18
17
16
15
13
12
11
10
1
47
40
32
24
16
8
0

a105b-1

DEFAULT STATES

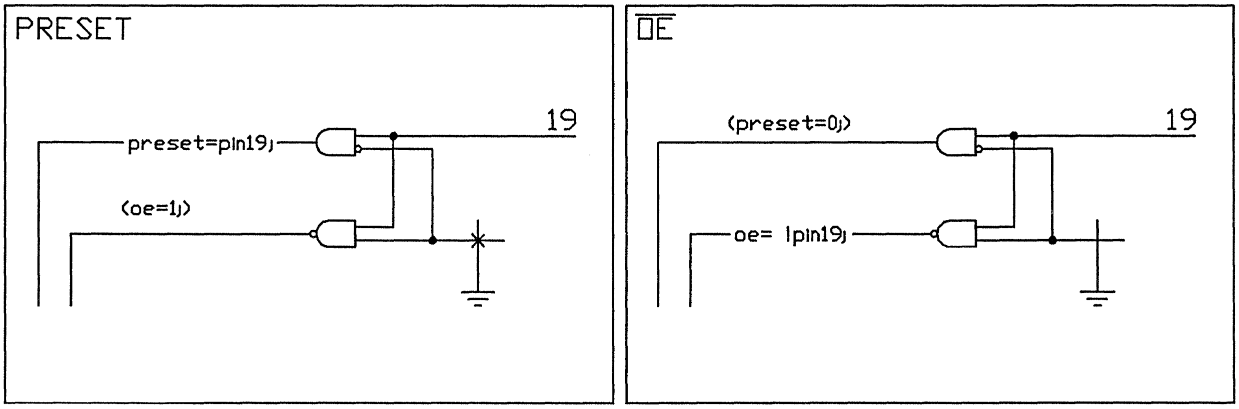
All unreferenced gates remain unprogrammed.

Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined.

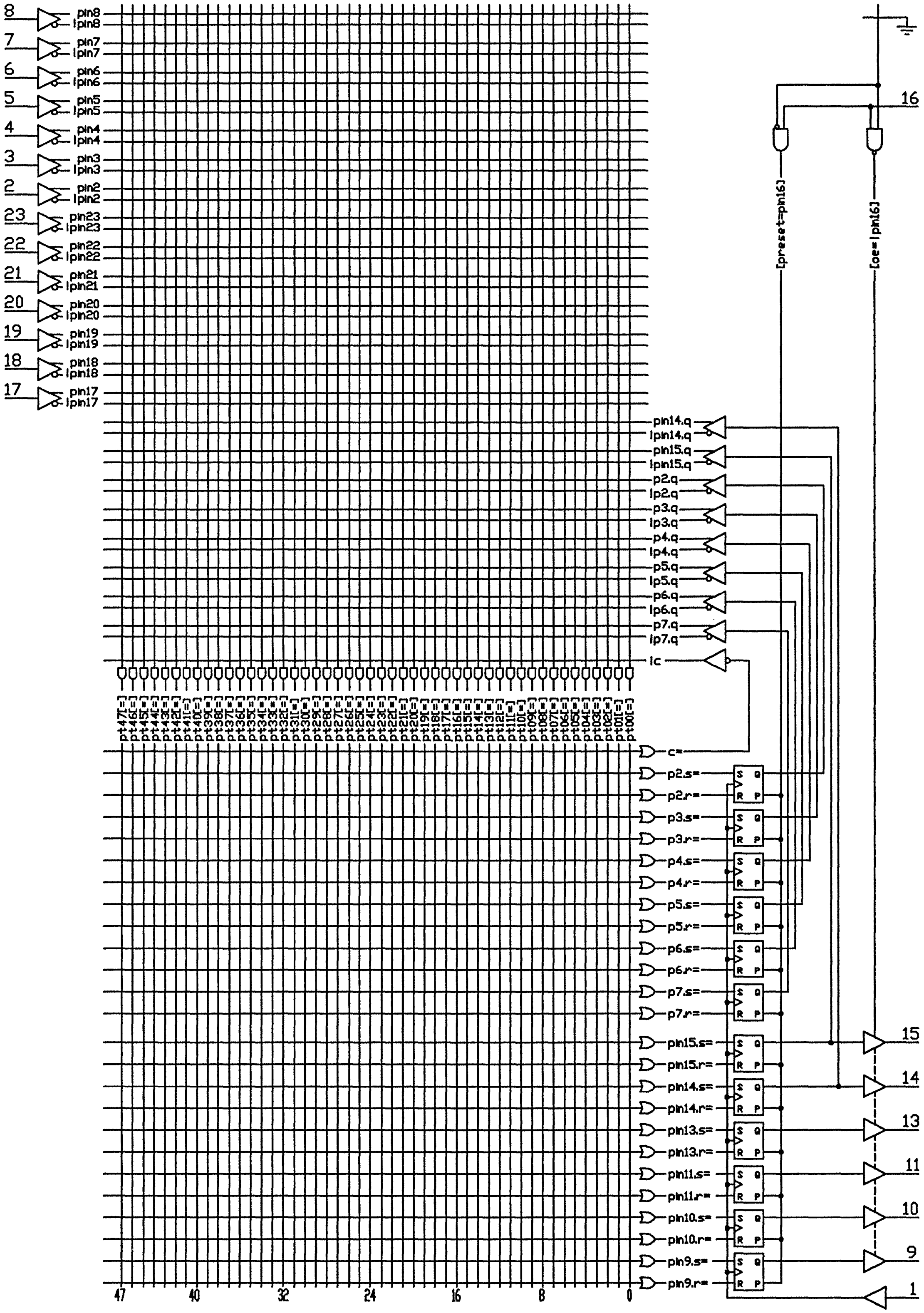
Example:

```
        pt00 = 0;              /* default AND gate */
        preset = pin19;        /* default PRESET/OE OPTION */
```

PRESET/$\overline{\text{OE}}$ OPTION



a105b-2


8
pin8
!pin8
7
pin7
!pin7
6
pin6
!pin6
5
pin5
!pin5
4
pin4
!pin4
3
pin3
!pin3
2
pin2
!pin2
23
pin23
!pin23
22
pin22
!pin22
21
pin21
!pin21
20
pin20
!pin20
19
pin19
!pin19
18
pin18
!pin18
17
pin17
!pin17
16
[preset=pin16]
[oe=!pin16]
pin14.q
!pin14.q
pin15.q
!pin15.q
p2.q
!p2.q
p3.q
!p3.q
p4.q
!p4.q
p5.q
!p5.q
p6.q
!p6.q
p7.q
!p7.q
!c
pt47[=] pt46[=] pt45[=] pt44[=] pt43[=] pt42[=] pt41[=] pt40[=] pt39[=] pt38[=] pt37[=] pt36[=] pt35[=] pt34[=] pt33[=] pt32[=] pt31[=] pt30[=] pt29[=] pt28[=] pt27[=] pt26[=] pt25[=] pt24[=] pt23[=] pt22[=] pt21[=] pt20[=] pt19[=] pt18[=] pt17[=] pt16[=] pt15[=] pt14[=] pt13[=] pt12[=] pt11[=] pt10[=] pt09[=] pt08[=] pt07[=] pt06[=] pt05[=] pt04[=] pt03[=] pt02[=] pt01[=] pt00[=]
c=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
p6.s=
p6.r=
p7.s=
p7.r=
pin15.s=
pin15.r=
pin14.s=
pin14.r=
pin13.s=
pin13.r=
pin11.s=
pin11.r=
pin10.s=
pin10.r=
pin9.s=
pin9.r=
S Q R P
15
14
13
11
10
9
1
47
40
32
24
16
8
0

a167b-1

DEFAULT STATES

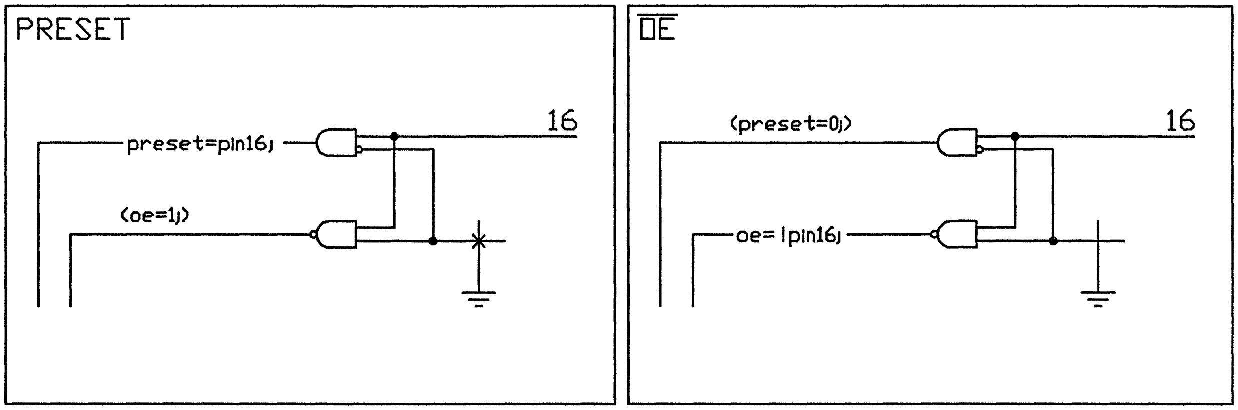
All unreferenced gates remain unprogrammed.

Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined.

Example:

```
pt00 = 0;              /* default AND gate */
preset = pin16;        /* default PRESET/OE OPTION */
```

PRESET/$\overline{OE}$ OPTION



a167b-2

DEFAULT STATES

All unreferenced gates remain unprogrammed.

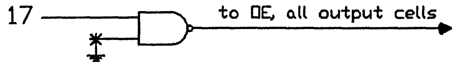
Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined. The Output Cells default to registered operation. Registers are positive-edge triggered. Outputs are permanently enabled.

Example:

```
pt96 = 0;          /* default AND gate */
oe = 1;            /* output buffers always enabled */
clk = pin1;        /* registers positive-edge triggered */
```

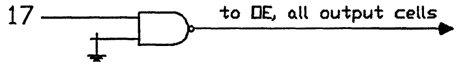
OUTPUT ENABLE

OUTPUTS PERMANENTLY ENABLED



```
oe = 1;          /* default */
```

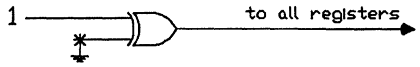
PIN 17 OUTPUT ENABLE



```
oe = !pin17;
```

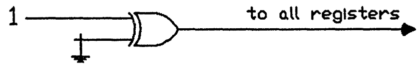
CLOCK POLARITY

POSITIVE-EDGE TRIGGERED



```
clk = pin1;    /* default */
```

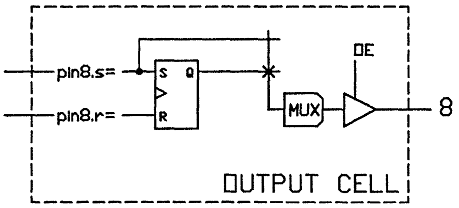
NEGATIVE-EDGE TRIGGERED



```
clk = !pin1;
```

TYPICAL OUTPUT CELL OPERATION

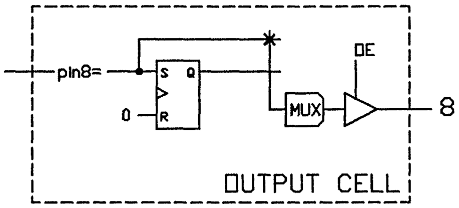
REGISTERED



No signal specification for pin8= is permitted.

Unprogrammed State

COMBINATORIAL

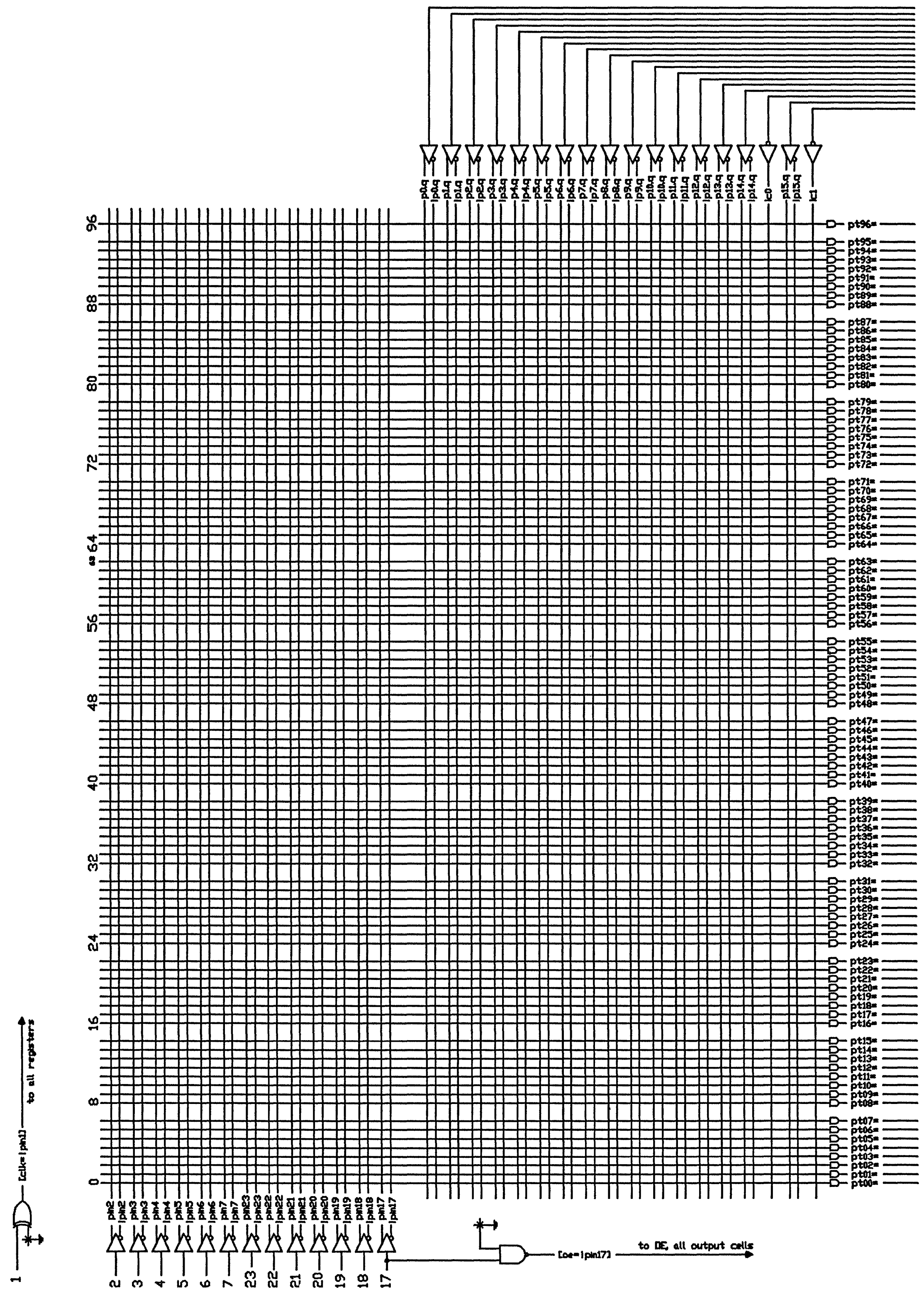


Signal specifications for pin8.s= and pin8.r= are not permitted. All cells of the reset term are programmed as if

pin8.r = 0;

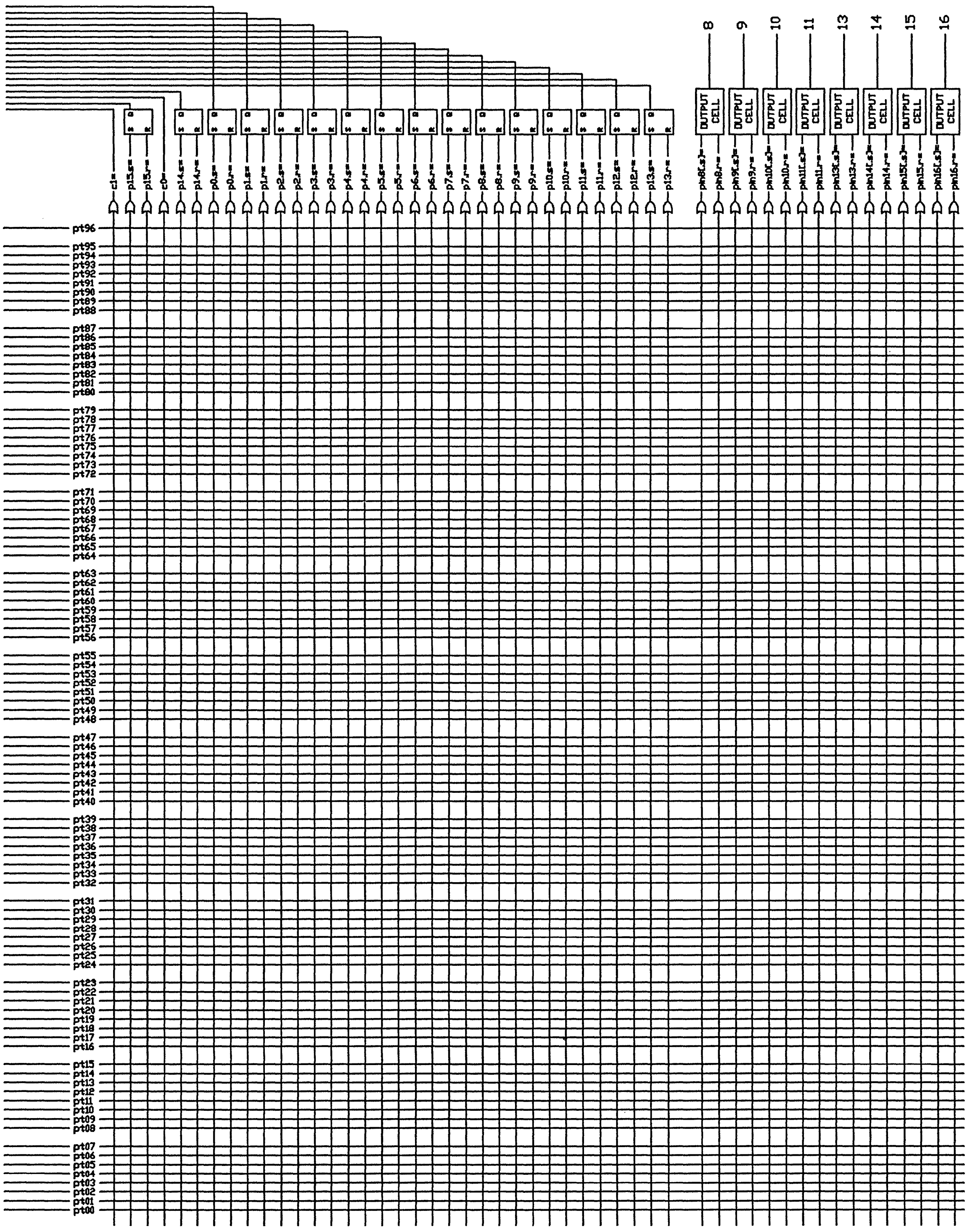
had been specified.

a506-1



p0.q
!p0.q
p1.q
!p1.q
p2.q
!p2.q
p3.q
!p3.q
p4.q
!p4.q
p5.q
!p5.q
p6.q
!p6.q
p7.q
!p7.q
p8.q
!p8.q
p9.q
!p9.q
p10.q
!p10.q
p11.q
!p11.q
p12.q
!p12.q
p13.q
!p13.q
p14.q
!p14.q
!c0
p15.q
!p15.q
!c1
96
88
80
72
64
56
48
40
32
24
16
8
0
pt96=
pt95=
pt94=
pt93=
pt92=
pt91=
pt90=
pt89=
pt88=
pt87=
pt86=
pt85=
pt84=
pt83=
pt82=
pt81=
pt80=
pt79=
pt78=
pt77=
pt76=
pt75=
pt74=
pt73=
pt72=
pt71=
pt70=
pt69=
pt68=
pt67=
pt66=
pt65=
pt64=
pt63=
pt62=
pt61=
pt60=
pt59=
pt58=
pt57=
pt56=
pt55=
pt54=
pt53=
pt52=
pt51=
pt50=
pt49=
pt48=
pt47=
pt46=
pt45=
pt44=
pt43=
pt42=
pt41=
pt40=
pt39=
pt38=
pt37=
pt36=
pt35=
pt34=
pt33=
pt32=
pt31=
pt30=
pt29=
pt28=
pt27=
pt26=
pt25=
pt24=
pt23=
pt22=
pt21=
pt20=
pt19=
pt18=
pt17=
pt16=
pt15=
pt14=
pt13=
pt12=
pt11=
pt10=
pt09=
pt08=
pt07=
pt06=
pt05=
pt04=
pt03=
pt02=
pt01=
pt00=
pin2
!pin2
pin3
!pin3
pin4
!pin4
pin5
!pin5
pin6
!pin6
pin7
!pin7
pin23
!pin23
pin22
!pin22
pin21
!pin21
pin20
!pin20
pin19
!pin19
pin18
!pin18
pin17
!pin17
1
2
3
4
5
6
7
23
22
21
20
19
18
17
[clk=!pin1]
to all registers
[oe=!pin17]
to OE, all output cells

a506-2


8
9
10
11
13
14
15
16
OUTPUT CELL
Q
S
R
c1=
p15.s=
p15.r=
c0=
p14.s=
p14.r=
p0.s=
p0.r=
p1.s=
p1.r=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
p6.s=
p6.r=
p7.s=
p7.r=
p8.s=
p8.r=
p9.s=
p9.r=
p10.s=
p10.r=
p11.s=
p11.r=
p12.s=
p12.r=
p13.s=
p13.r=
pin8[.s]=
pin8.r=
pin9[.s]=
pin9.r=
pin10[.s]=
pin10.r=
pin11[.s]=
pin11.r=
pin13[.s]=
pin13.r=
pin14[.s]=
pin14.r=
pin15[.s]=
pin15.r=
pin16[.s]=
pin16.r=
pt96
pt95
pt94
pt93
pt92
pt91
pt90
pt89
pt88
pt87
pt86
pt85
pt84
pt83
pt82
pt81
pt80
pt79
pt78
pt77
pt76
pt75
pt74
pt73
pt72
pt71
pt70
pt69
pt68
pt67
pt66
pt65
pt64
pt63
pt62
pt61
pt60
pt59
pt58
pt57
pt56
pt55
pt54
pt53
pt52
pt51
pt50
pt49
pt48
pt47
pt46
pt45
pt44
pt43
pt42
pt41
pt40
pt39
pt38
pt37
pt36
pt35
pt34
pt33
pt32
pt31
pt30
pt29
pt28
pt27
pt26
pt25
pt24
pt23
pt22
pt21
pt20
pt19
pt18
pt17
pt16
pt15
pt14
pt13
pt12
pt11
pt10
pt09
pt08
pt07
pt06
pt05
pt04
pt03
pt02
pt01
pt00

a506-3

DEFAULT STATES

All unreferenced gates remain unprogrammed.

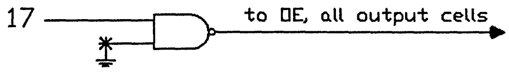
Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined. The Output Cells default to registered operation. Registers are positive-edge triggered. Outputs are premanently enabled.

Example:

```
    pt79 = 0;               /* default AND gate */
    oe = 1;                 /* output buffers always enabled */
    clk = pin1;             /* registers positive-edge triggered */
```

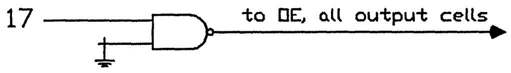
OUTPUT ENABLE

OUTPUTS PERMANENTLY ENABLED



```
oe = 1;          /* default */
```

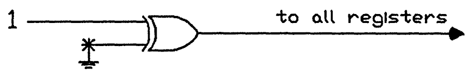
PIN 17 OUTPUT ENABLE



```
oe = !pin17;
```

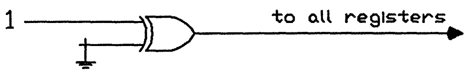
CLOCK POLARITY

POSITIVE-EDGE TRIGGERED



```
clk = pin1;    /* default */
```

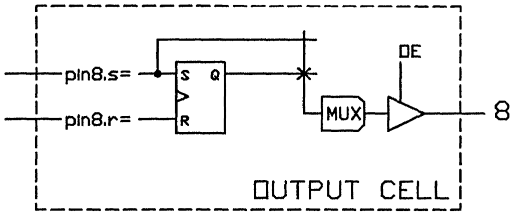
NEGATIVE-EDGE TRIGGERED



```
clk = !pin1;
```

TYPICAL OUTPUT CELL OPERATION

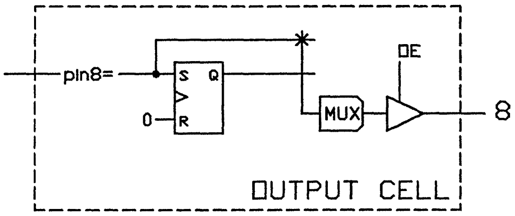
REGISTERED



No signal specification for pin8= is permitted.

Unprogrammed State

COMBINATORIAL



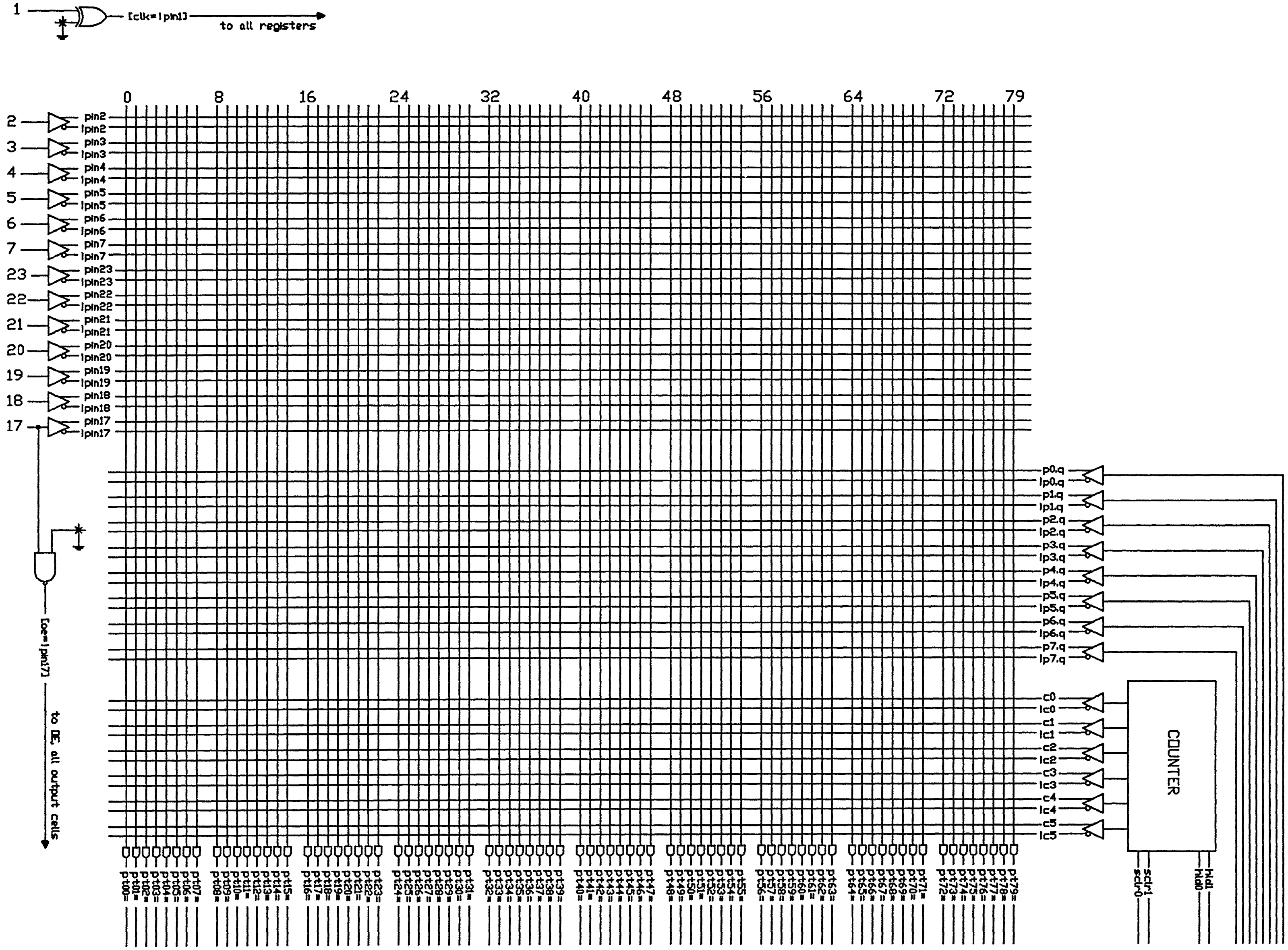
Signal specifications for pin8.s= and pin8.r= are not permitted. All cells of the reset term are programmed as if

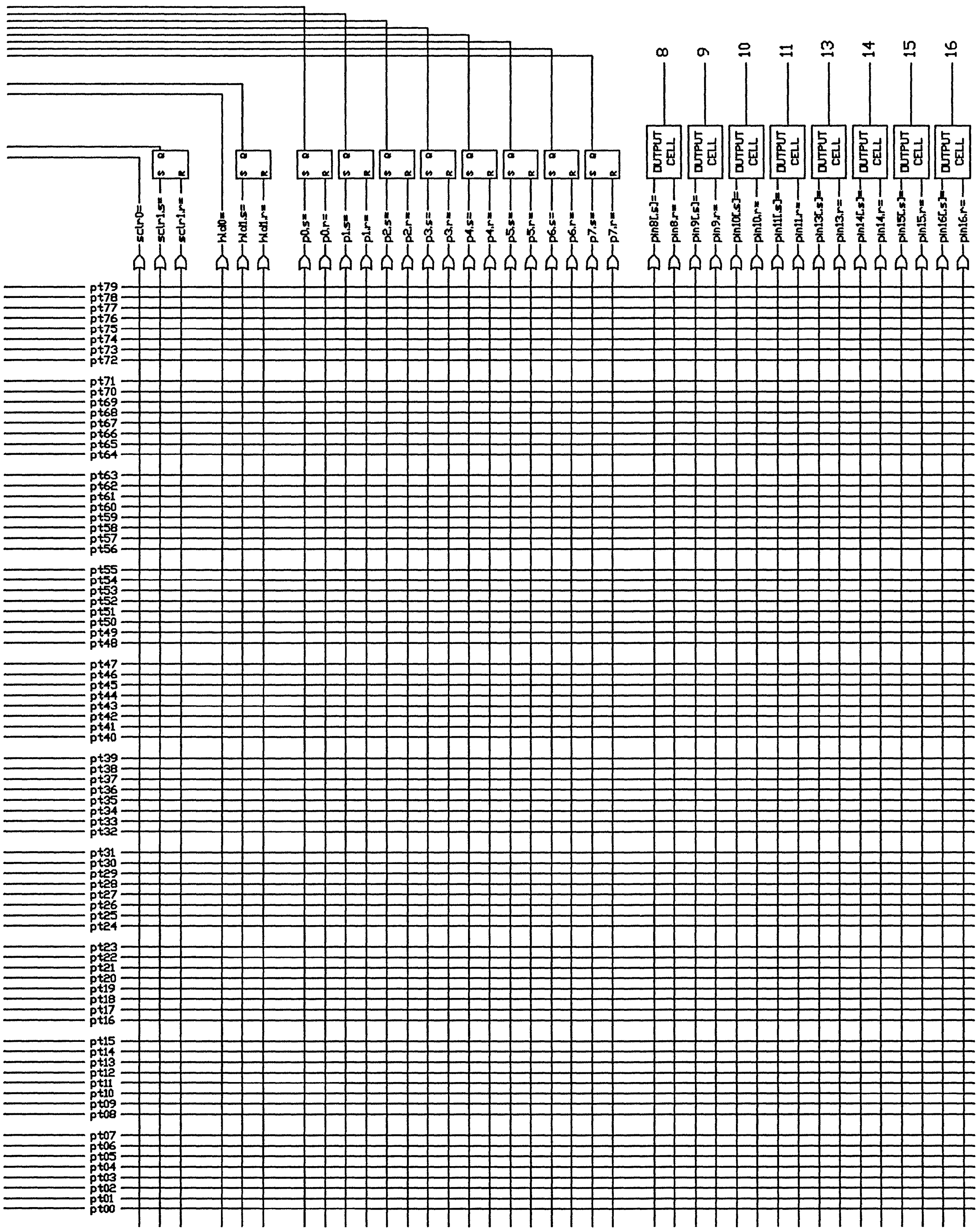
```
pin8.r = 0;
```

had been specified.

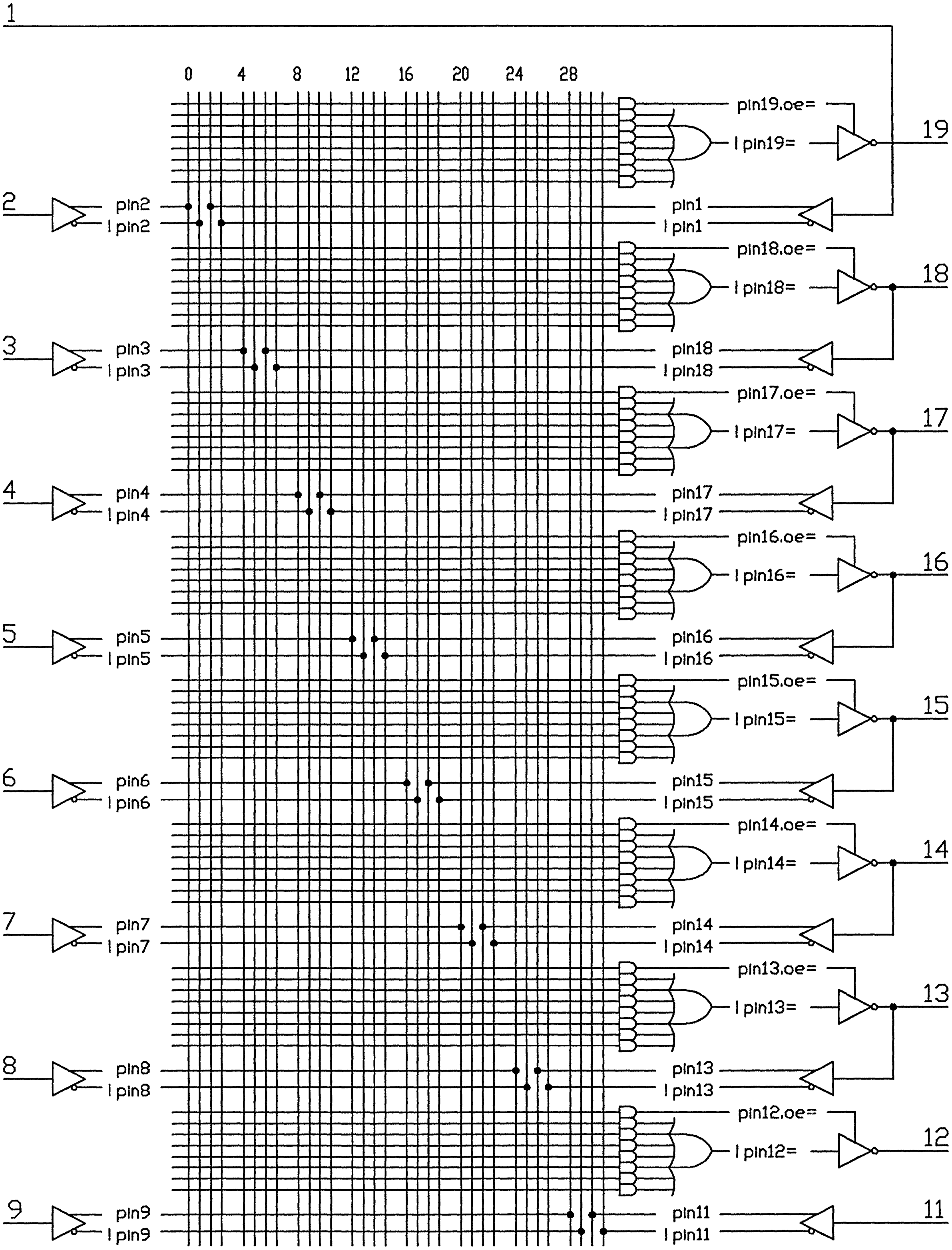
a507-1

a507-2

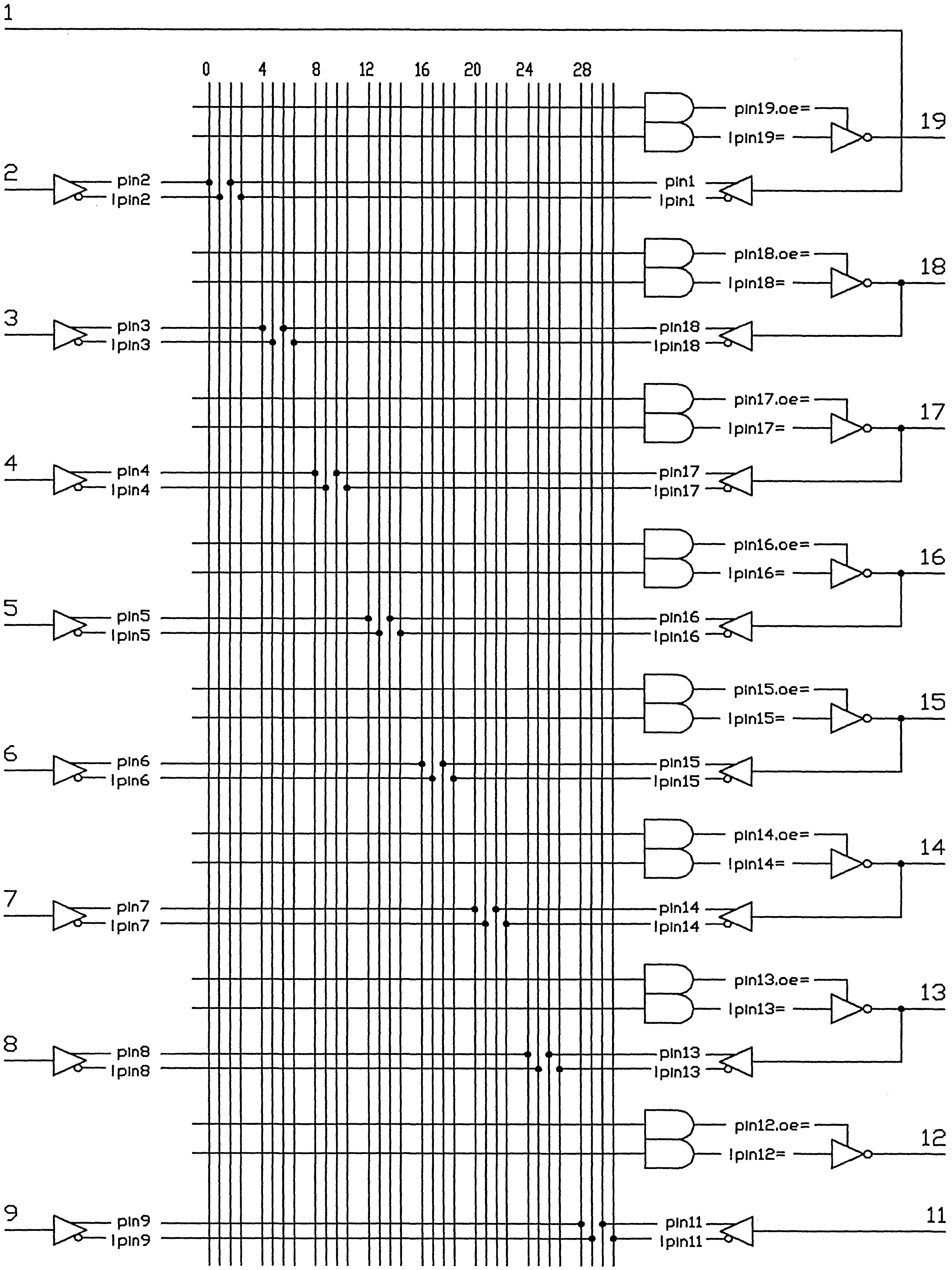

1
[clk=!pin1]
to all registers
0
8
16
24
32
40
48
56
64
72
79
2
3
4
5
6
7
23
22
21
20
19
18
17
pin2
!pin2
pin3
!pin3
pin4
!pin4
pin5
!pin5
pin6
!pin6
pin7
!pin7
pin23
!pin23
pin22
!pin22
pin21
!pin21
pin20
!pin20
pin19
!pin19
pin18
!pin18
pin17
!pin17
[oe=!pin17]
to OE, all output cells
p0.q
!p0.q
p1.q
!p1.q
p2.q
!p2.q
p3.q
!p3.q
p4.q
!p4.q
p5.q
!p5.q
p6.q
!p6.q
p7.q
!p7.q
c0
!c0
c1
!c1
c2
!c2
c3
!c3
c4
!c4
c5
!c5
COUNTER
sclr1
sclr0
hld1
hld0
pt00=
pt01=
pt02=
pt03=
pt04=
pt05=
pt06=
pt07=
pt08=
pt09=
pt10=
pt11=
pt12=
pt13=
pt14=
pt15=
pt16=
pt17=
pt18=
pt19=
pt20=
pt21=
pt22=
pt23=
pt24=
pt25=
pt26=
pt27=
pt28=
pt29=
pt30=
pt31=
pt32=
pt33=
pt34=
pt35=
pt36=
pt37=
pt38=
pt39=
pt40=
pt41=
pt42=
pt43=
pt44=
pt45=
pt46=
pt47=
pt48=
pt49=
pt50=
pt51=
pt52=
pt53=
pt54=
pt55=
pt56=
pt57=
pt58=
pt59=
pt60=
pt61=
pt62=
pt63=
pt64=
pt65=
pt66=
pt67=
pt68=
pt69=
pt70=
pt71=
pt72=
pt73=
pt74=
pt75=
pt76=
pt77=
pt78=
pt79=


8
9
10
11
13
14
15
16
OUTPUT CELL
OUTPUT CELL
OUTPUT CELL
OUTPUT CELL
OUTPUT CELL
OUTPUT CELL
OUTPUT CELL
OUTPUT CELL
Q S R
sctr0=
sctr1.s=
sctr1.r=
hld0=
hld1.s=
hld1.r=
p0.s=
p0.r=
p1.s=
p1.r=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
p6.s=
p6.r=
p7.s=
p7.r=
pin8[.s]=
pin8.r=
pin9[.s]=
pin9.r=
pin10[.s]=
pin10.r=
pin11[.s]=
pin11.r=
pin13[.s]=
pin13.r=
pin14[.s]=
pin14.r=
pin15[.s]=
pin15.r=
pin16[.s]=
pin16.r=
pt79
pt78
pt77
pt76
pt75
pt74
pt73
pt72
pt71
pt70
pt69
pt68
pt67
pt66
pt65
pt64
pt63
pt62
pt61
pt60
pt59
pt58
pt57
pt56
pt55
pt54
pt53
pt52
pt51
pt50
pt49
pt48
pt47
pt46
pt45
pt44
pt43
pt42
pt41
pt40
pt39
pt38
pt37
pt36
pt35
pt34
pt33
pt32
pt31
pt30
pt29
pt28
pt27
pt26
pt25
pt24
pt23
pt22
pt21
pt20
pt19
pt18
pt17
pt16
pt15
pt14
pt13
pt12
pt11
pt10
pt09
pt08
pt07
pt06
pt05
pt04
pt03
pt02
pt01
pt00

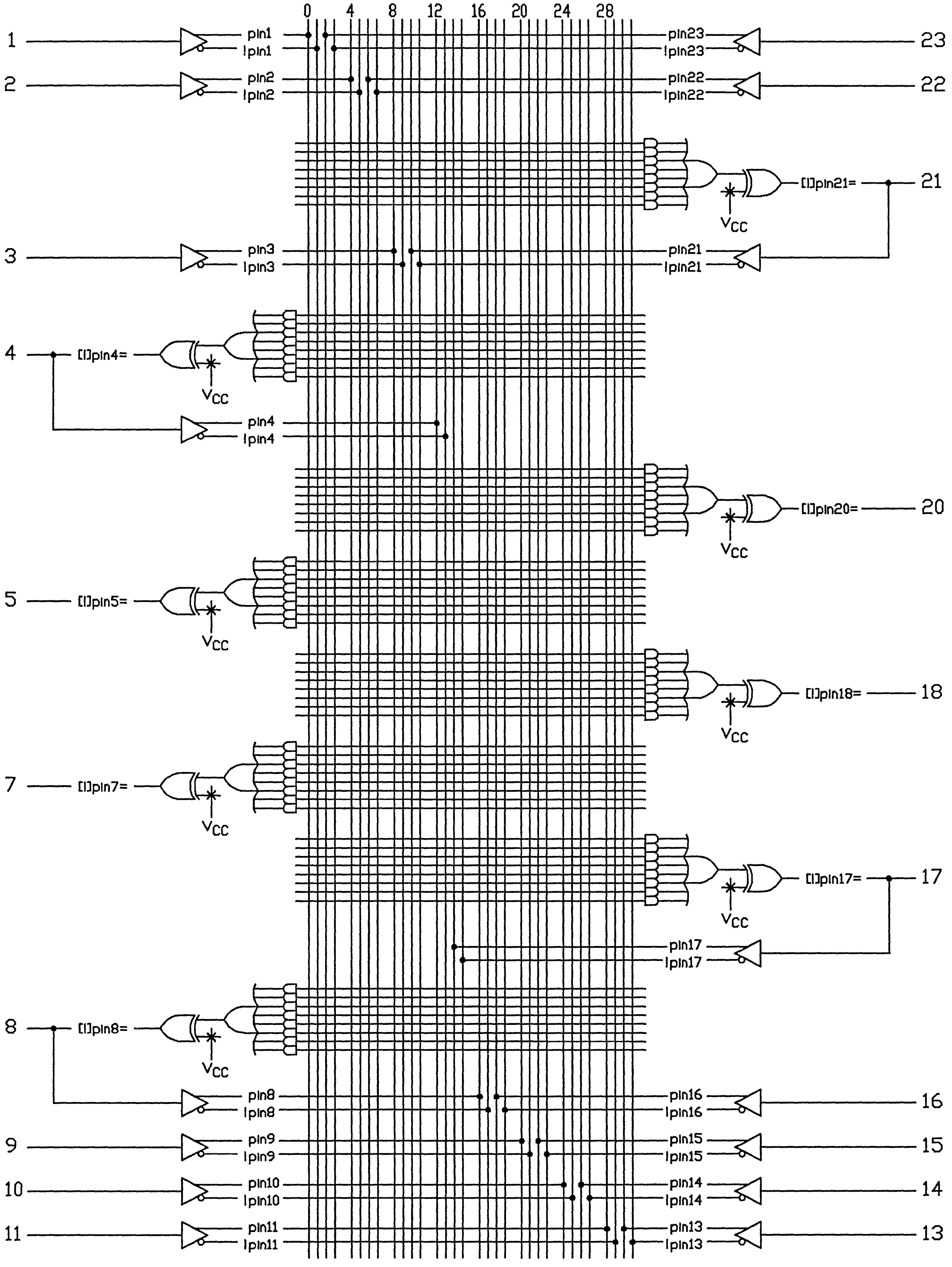
a507-3


1
0
4
8
12
16
20
24
28
pin19.oe=
I pin19=
19
2
pin2
I pin2
pin1
I pin1
pin18.oe=
I pin18=
18
3
pin3
I pin3
pin18
I pin18
pin17.oe=
I pin17=
17
4
pin4
I pin4
pin17
I pin17
pin16.oe=
I pin16=
16
5
pin5
I pin5
pin16
I pin16
pin15.oe=
I pin15=
15
6
pin6
I pin6
pin15
I pin15
pin14.oe=
I pin14=
14
7
pin7
I pin7
pin14
I pin14
pin13.oe=
I pin13=
13
8
pin8
I pin8
pin13
I pin13
pin12.oe=
I pin12=
12
9
pin9
I pin9
pin11
I pin11
11

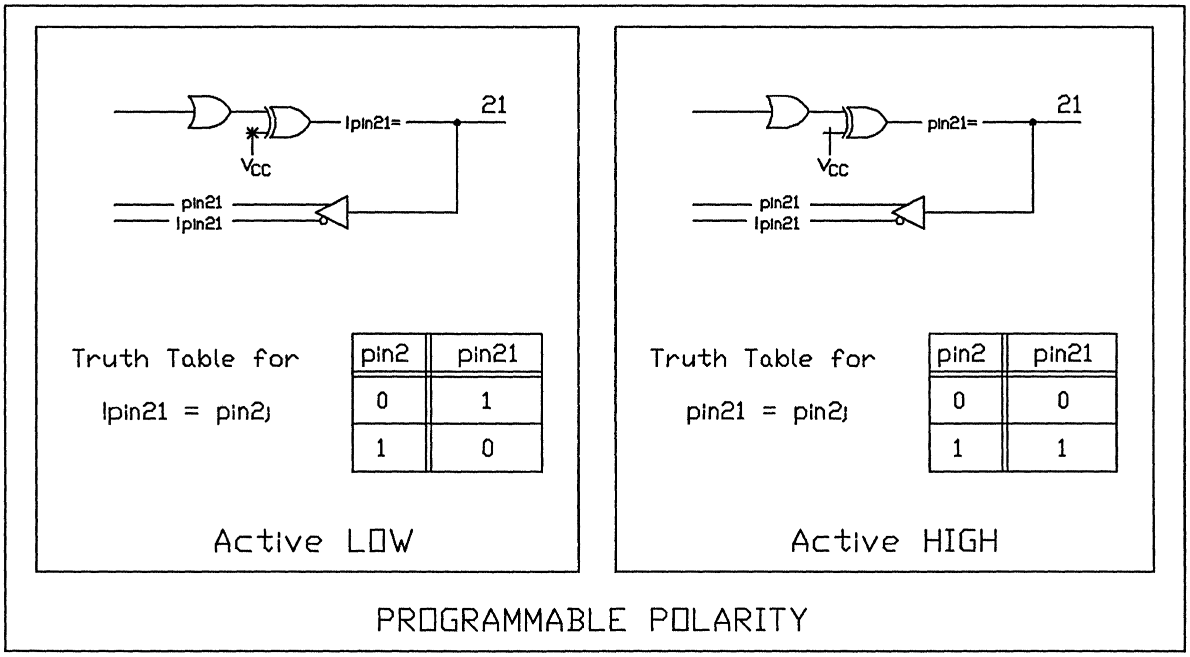
p16l8


1
0
4
8
12
16
20
24
28
pin19.oe=
Ipin19=
19
2
pin2
Ipin2
pin1
Ipin1
pin18.oe=
Ipin18=
18
3
pin3
Ipin3
pin18
Ipin18
pin17.oe=
Ipin17=
17
4
pin4
Ipin4
pin17
Ipin17
pin16.oe=
Ipin16=
16
5
pin5
Ipin5
pin16
Ipin16
pin15.oe=
Ipin15=
15
6
pin6
Ipin6
pin15
Ipin15
pin14.oe=
Ipin14=
14
7
pin7
Ipin7
pin14
Ipin14
pin13.oe=
Ipin13=
13
8
pin8
Ipin8
pin13
Ipin13
pin12.oe=
Ipin12=
12
9
pin9
Ipin9
pin11
Ipin11
11

p16n8


0
4
8
12
16
20
24
28
1
pin1
!pin1
pin23
!pin23
23
2
pin2
!pin2
pin22
!pin22
22
[1]pin21=
21
VCC
3
pin3
!pin3
pin21
!pin21
4
[1]pin4=
VCC
pin4
!pin4
[1]pin20=
20
VCC
5
[1]pin5=
VCC
[1]pin18=
18
VCC
7
[1]pin7=
VCC
[1]pin17=
17
VCC
pin17
!pin17
8
[1]pin8=
VCC
pin8
!pin8
pin16
!pin16
16
9
pin9
!pin9
pin15
!pin15
15
10
pin10
!pin10
pin14
!pin14
14
11
pin11
!pin11
pin13
!pin13
13

p16p8e-1



Truth Table for

!pin21 = pin2;

pin2	pin21
0	1
1	0

Active LOW

Truth Table for

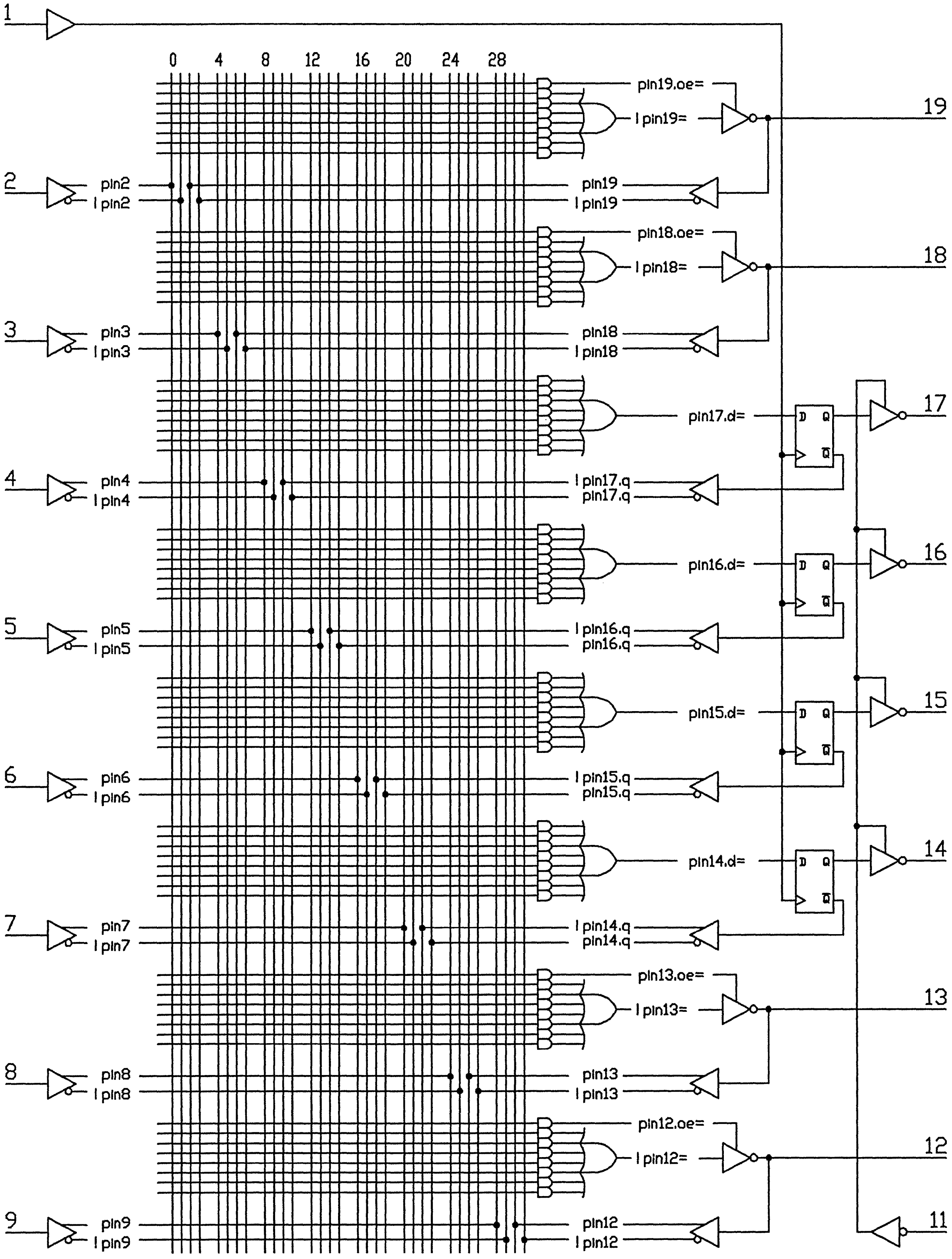
pin21 = pin2;

pin2	pin21
0	0
1	1

Active HIGH

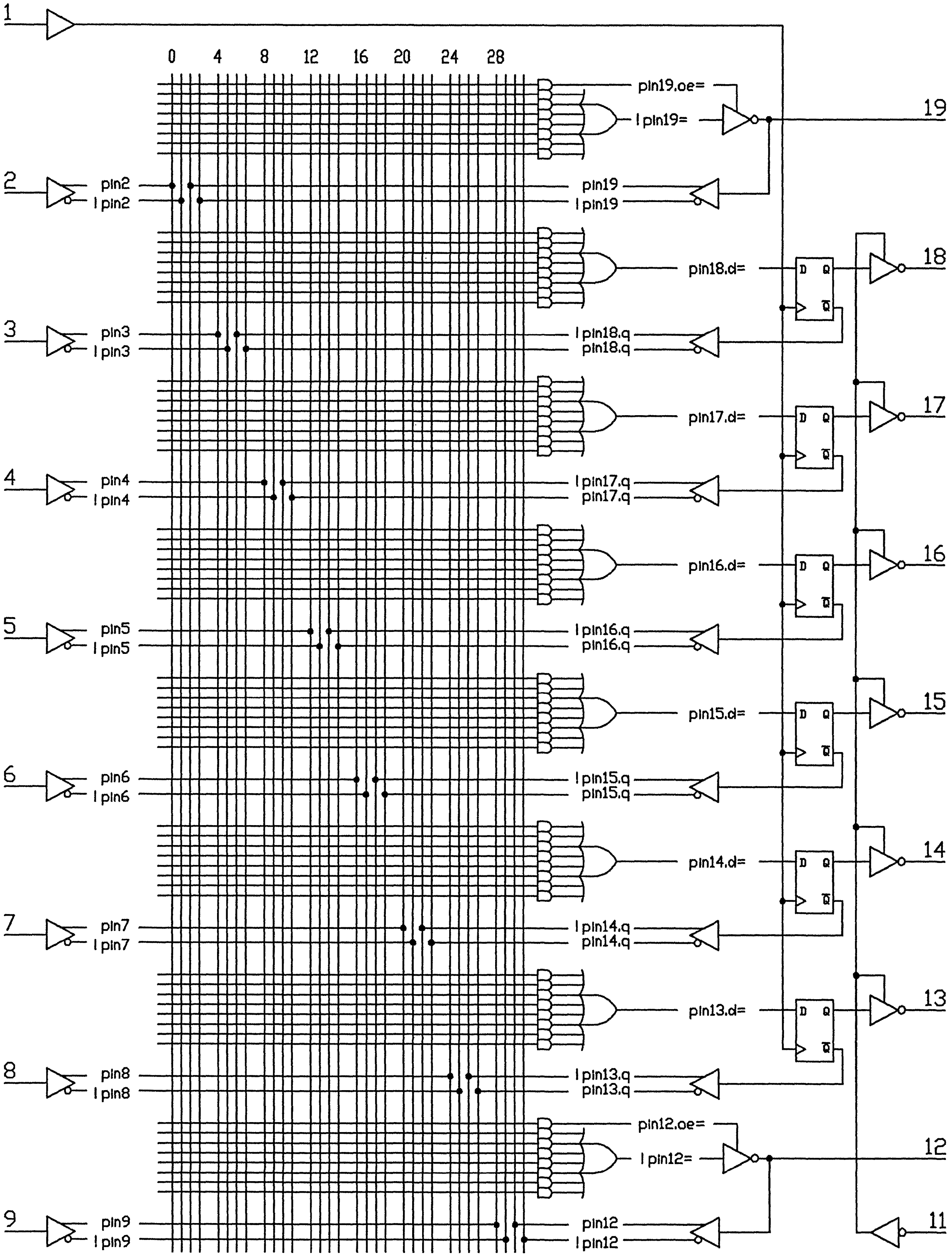
PROGRAMMABLE POLARITY

p16p8e-2

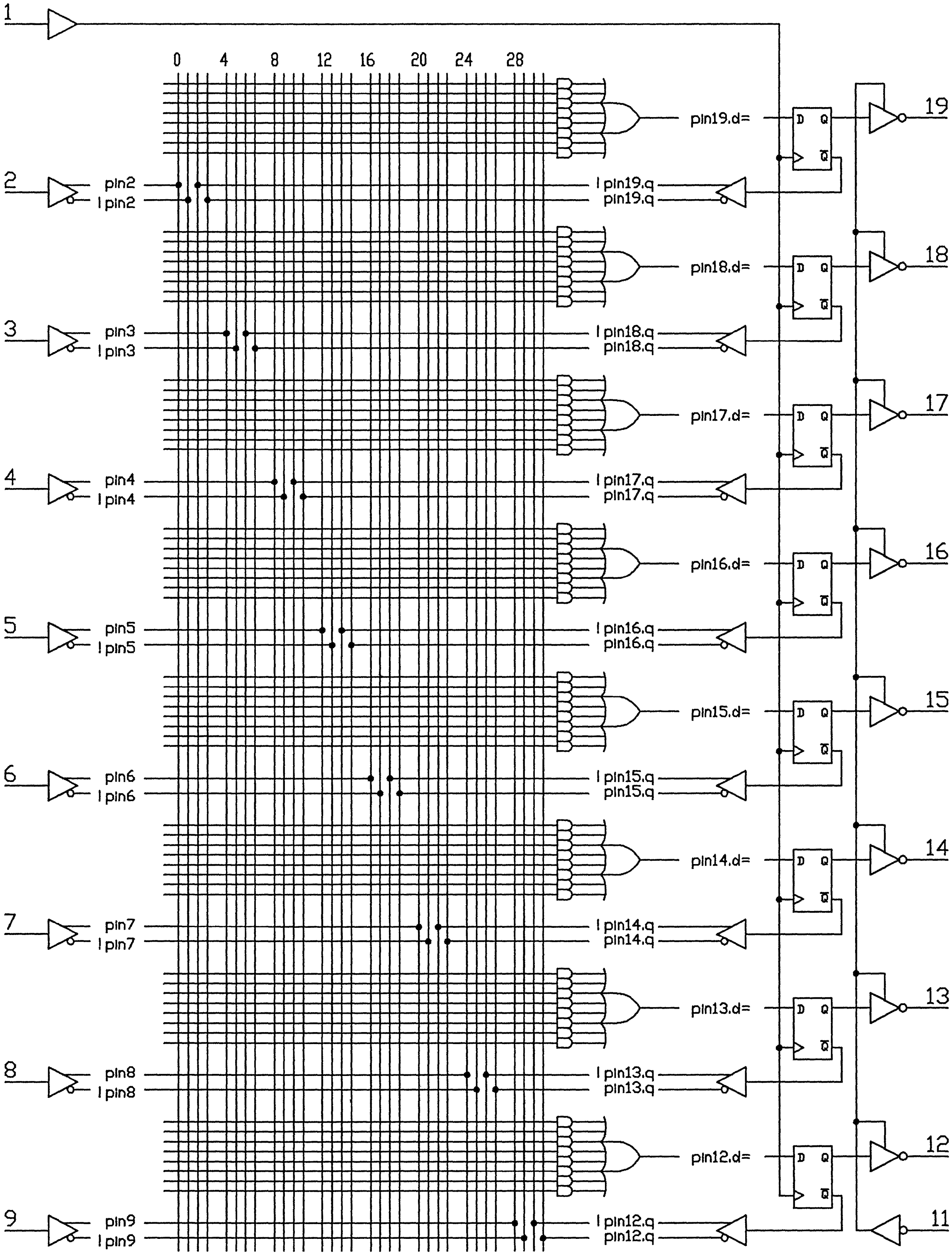


1
0
4
8
12
16
20
24
28
pin19.oe=
! pin19=
19
2
pin2
! pin2
pin19
! pin19
pin18.oe=
! pin18=
18
3
pin3
! pin3
pin18
! pin18
pin17.d=
D
Q
Q̄
17
4
pin4
! pin4
! pin17.q
pin17.q
pin16.d=
16
5
pin5
! pin5
! pin16.q
pin16.q
pin15.d=
15
6
pin6
! pin6
! pin15.q
pin15.q
pin14.d=
14
7
pin7
! pin7
! pin14.q
pin14.q
pin13.oe=
! pin13=
13
8
pin8
! pin8
pin13
! pin13
pin12.oe=
! pin12=
12
9
pin9
! pin9
pin12
! pin12
11

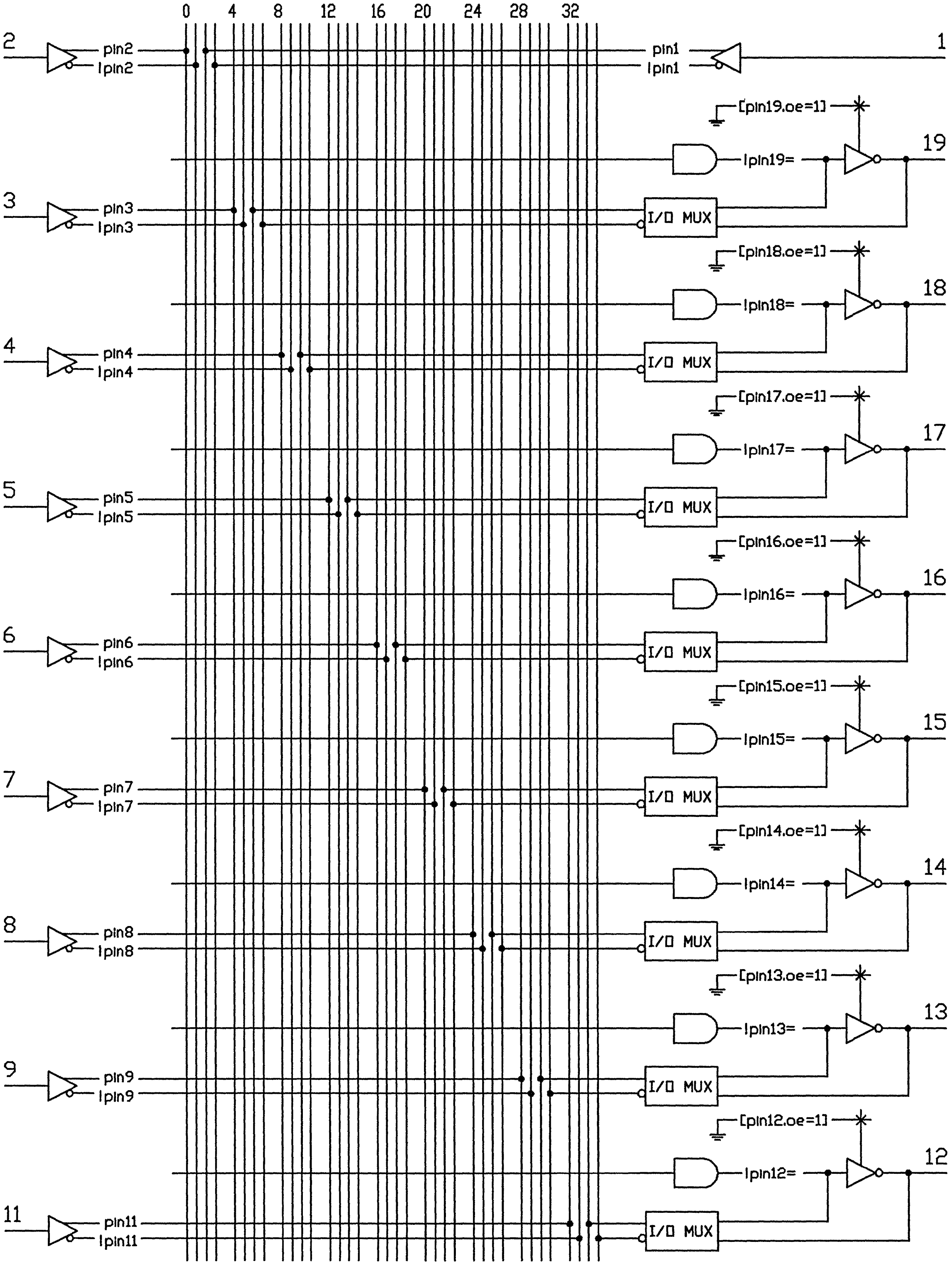
p16r4


1
0
4
8
12
16
20
24
28
pin19.oe=
! pin19=
19
2
pin2
! pin2
pin19
! pin19
pin18.d=
18
3
pin3
! pin3
! pin18.q
pin18.q
pin17.d=
17
4
pin4
! pin4
! pin17.q
pin17.q
pin16.d=
16
5
pin5
! pin5
! pin16.q
pin16.q
pin15.d=
15
6
pin6
! pin6
! pin15.q
pin15.q
pin14.d=
14
7
pin7
! pin7
! pin14.q
pin14.q
pin13.d=
13
8
pin8
! pin8
! pin13.q
pin13.q
pin12.oe=
! pin12=
12
9
pin9
! pin9
pin12
! pin12
11
D
Q
Q̄

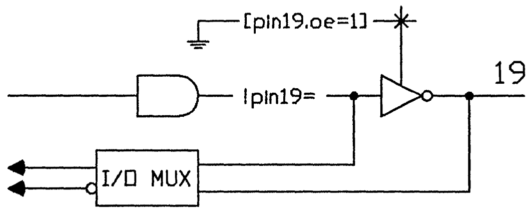
p16r6


1
0
4
8
12
16
20
24
28
pin19.d=
19
2
pin2
! pin2
! pin19.q
pin19.q
pin18.d=
18
3
pin3
! pin3
! pin18.q
pin18.q
pin17.d=
17
4
pin4
! pin4
! pin17.q
pin17.q
pin16.d=
16
5
pin5
! pin5
! pin16.q
pin16.q
pin15.d=
15
6
pin6
! pin6
! pin15.q
pin15.q
pin14.d=
14
7
pin7
! pin7
! pin14.q
pin14.q
pin13.d=
13
8
pin8
! pin8
! pin13.q
pin13.q
pin12.d=
12
9
pin9
! pin9
! pin12.q
pin12.q
11
D
Q
Q̄

p16r8


0
4
8
12
16
20
24
28
32
2
pin2
!pin2
pin1
!pin1
1
[pin19.oe=1]
!pin19=
19
3
pin3
!pin3
I/O MUX
[pin18.oe=1]
!pin18=
18
4
pin4
!pin4
I/O MUX
[pin17.oe=1]
!pin17=
17
5
pin5
!pin5
I/O MUX
[pin16.oe=1]
!pin16=
16
6
pin6
!pin6
I/O MUX
[pin15.oe=1]
!pin15=
15
7
pin7
!pin7
I/O MUX
[pin14.oe=1]
!pin14=
14
8
pin8
!pin8
I/O MUX
[pin13.oe=1]
!pin13=
13
9
pin9
!pin9
I/O MUX
[pin12.oe=1]
!pin12=
12
11
pin11
!pin11
I/O MUX

p18n8-1



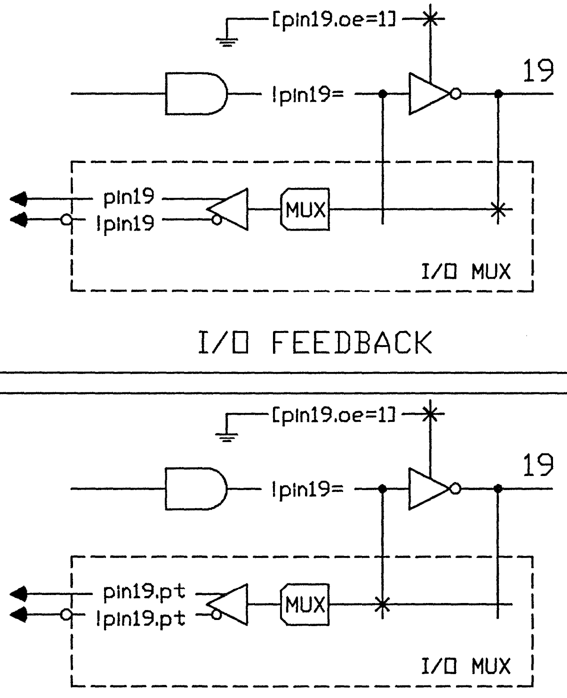
If there is no signal specification for pin19.oe the Architectural Fuse remains intact and the output buffer is in high-impedence state.

If there is a signal specification of

pin19.oe = 1;

the fuse is programmed and the output buffer is in output state.

TYPICAL OUTPUT BUFFER PROGRAMMING

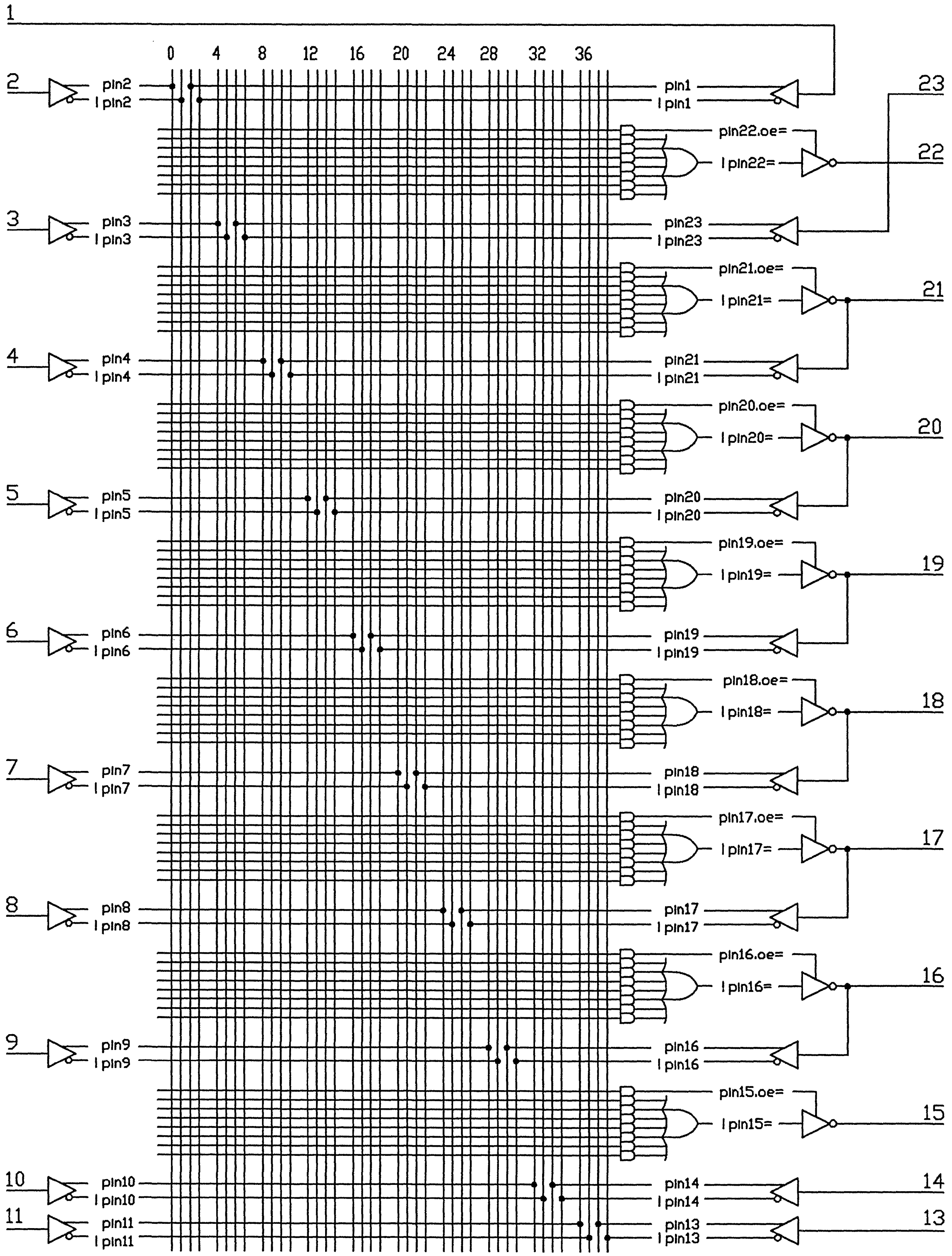


I/O FEEDBACK

HIGH-SPEED FEEDBACK

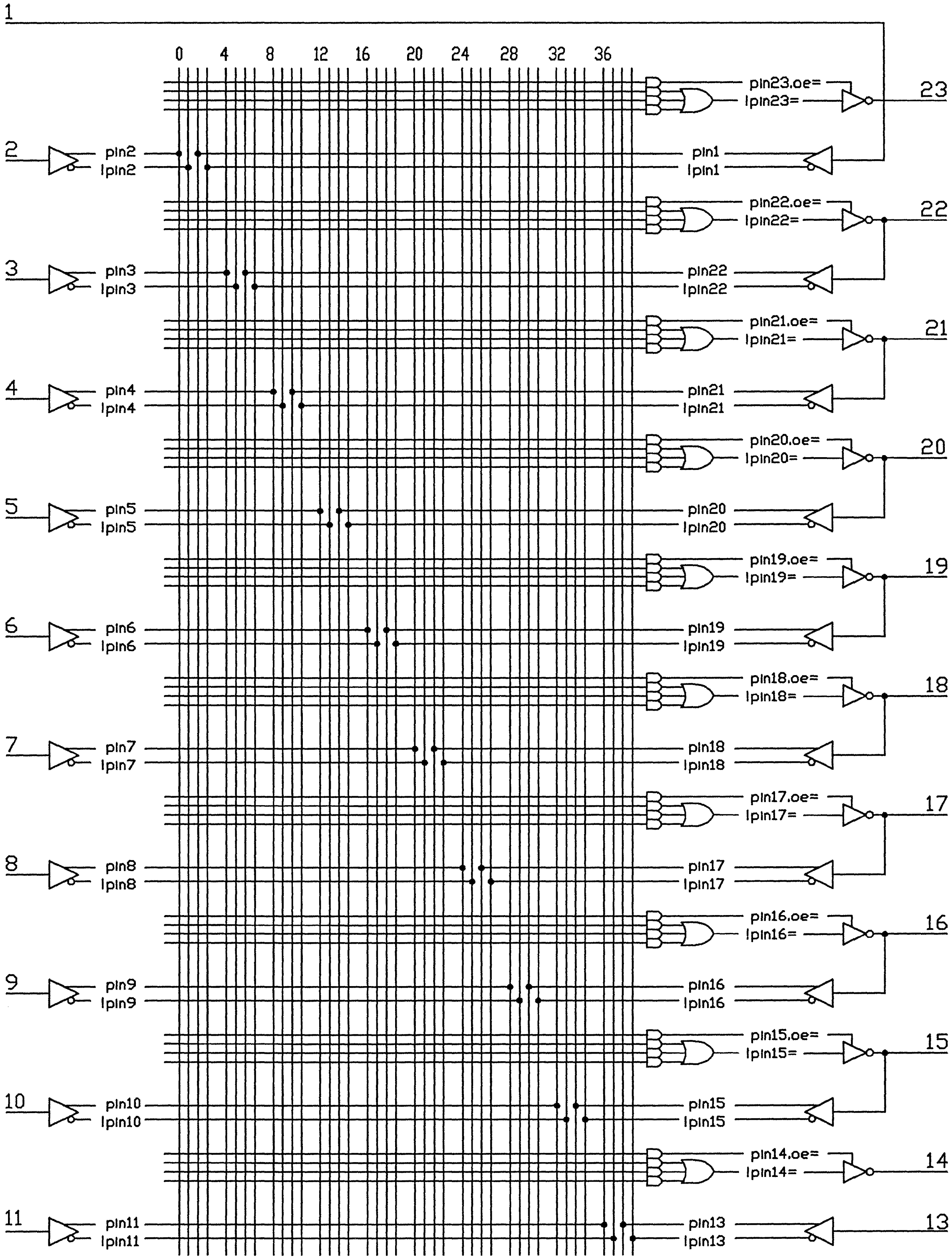
TYPICAL I/O MULTIPLEXER PROGRAMMING

p18n8-2

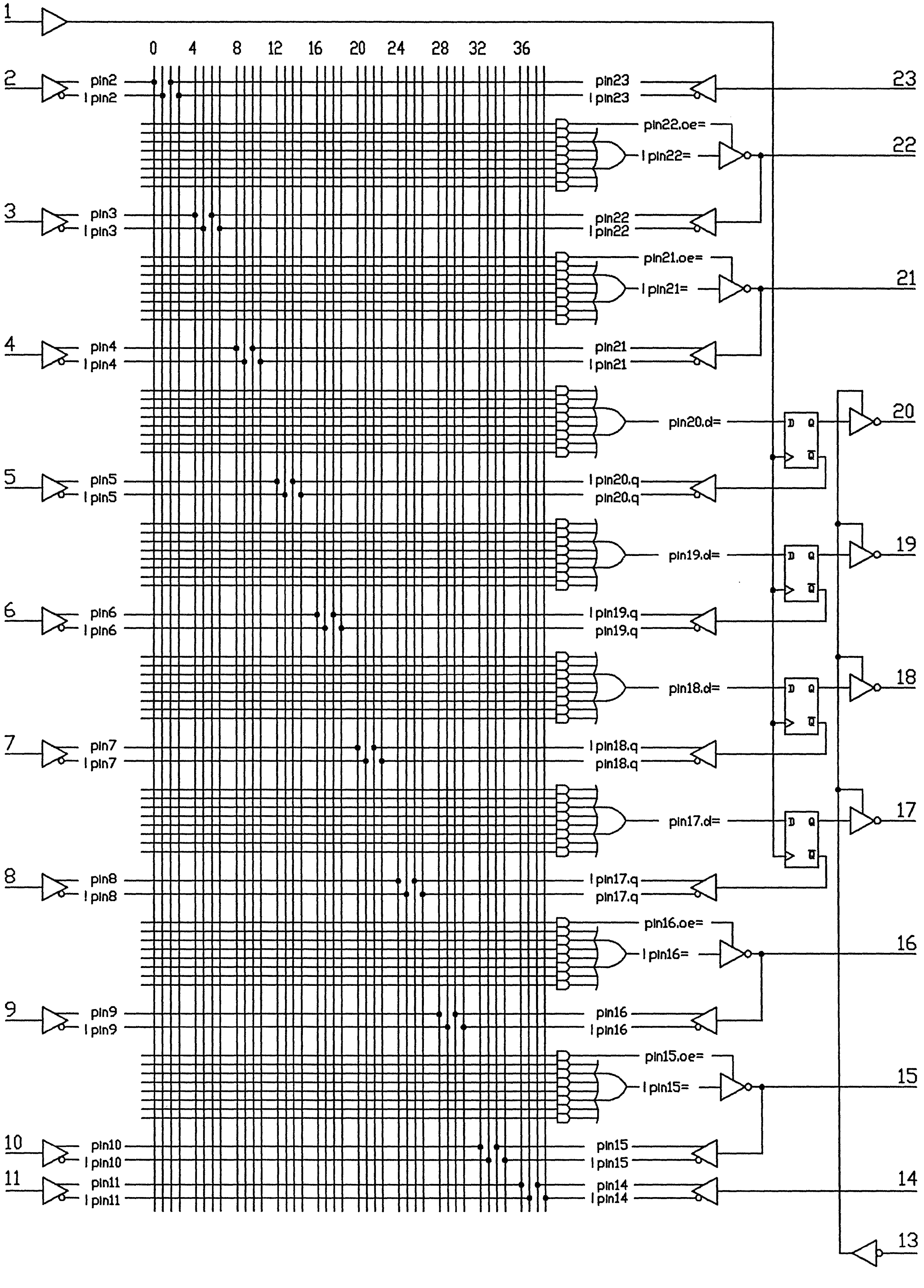


1
0
4
8
12
16
20
24
28
32
36
2
pin2
I pin2
pin1
I pin1
23
pin22.oe=
I pin22=
22
3
pin3
I pin3
pin23
I pin23
pin21.oe=
I pin21=
21
4
pin4
I pin4
pin21
I pin21
pin20.oe=
I pin20=
20
5
pin5
I pin5
pin20
I pin20
pin19.oe=
I pin19=
19
6
pin6
I pin6
pin19
I pin19
pin18.oe=
I pin18=
18
7
pin7
I pin7
pin18
I pin18
pin17.oe=
I pin17=
17
8
pin8
I pin8
pin17
I pin17
pin16.oe=
I pin16=
16
9
pin9
I pin9
pin16
I pin16
pin15.oe=
I pin15=
15
10
pin10
I pin10
pin14
I pin14
14
11
pin11
I pin11
pin13
I pin13
13

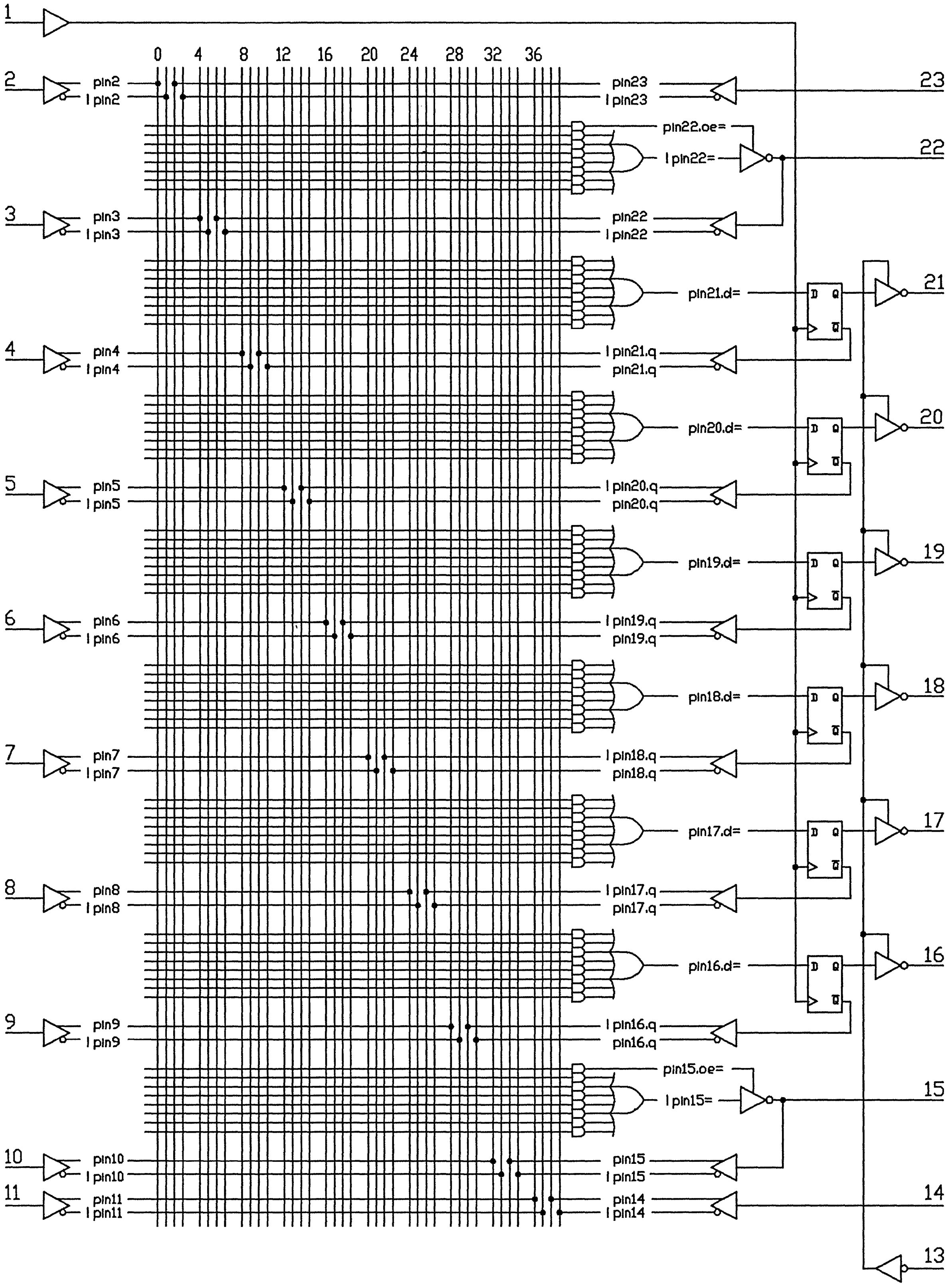
p20l8


1
0
4
8
12
16
20
24
28
32
36
pin23.oe=
!pin23=
23
2
pin2
!pin2
pin1
!pin1
pin22.oe=
!pin22=
22
3
pin3
!pin3
pin22
!pin22
pin21.oe=
!pin21=
21
4
pin4
!pin4
pin21
!pin21
pin20.oe=
!pin20=
20
5
pin5
!pin5
pin20
!pin20
pin19.oe=
!pin19=
19
6
pin6
!pin6
pin19
!pin19
pin18.oe=
!pin18=
18
7
pin7
!pin7
pin18
!pin18
pin17.oe=
!pin17=
17
8
pin8
!pin8
pin17
!pin17
pin16.oe=
!pin16=
16
9
pin9
!pin9
pin16
!pin16
pin15.oe=
!pin15=
15
10
pin10
!pin10
pin15
!pin15
pin14.oe=
!pin14=
14
11
pin11
!pin11
pin13
!pin13
13

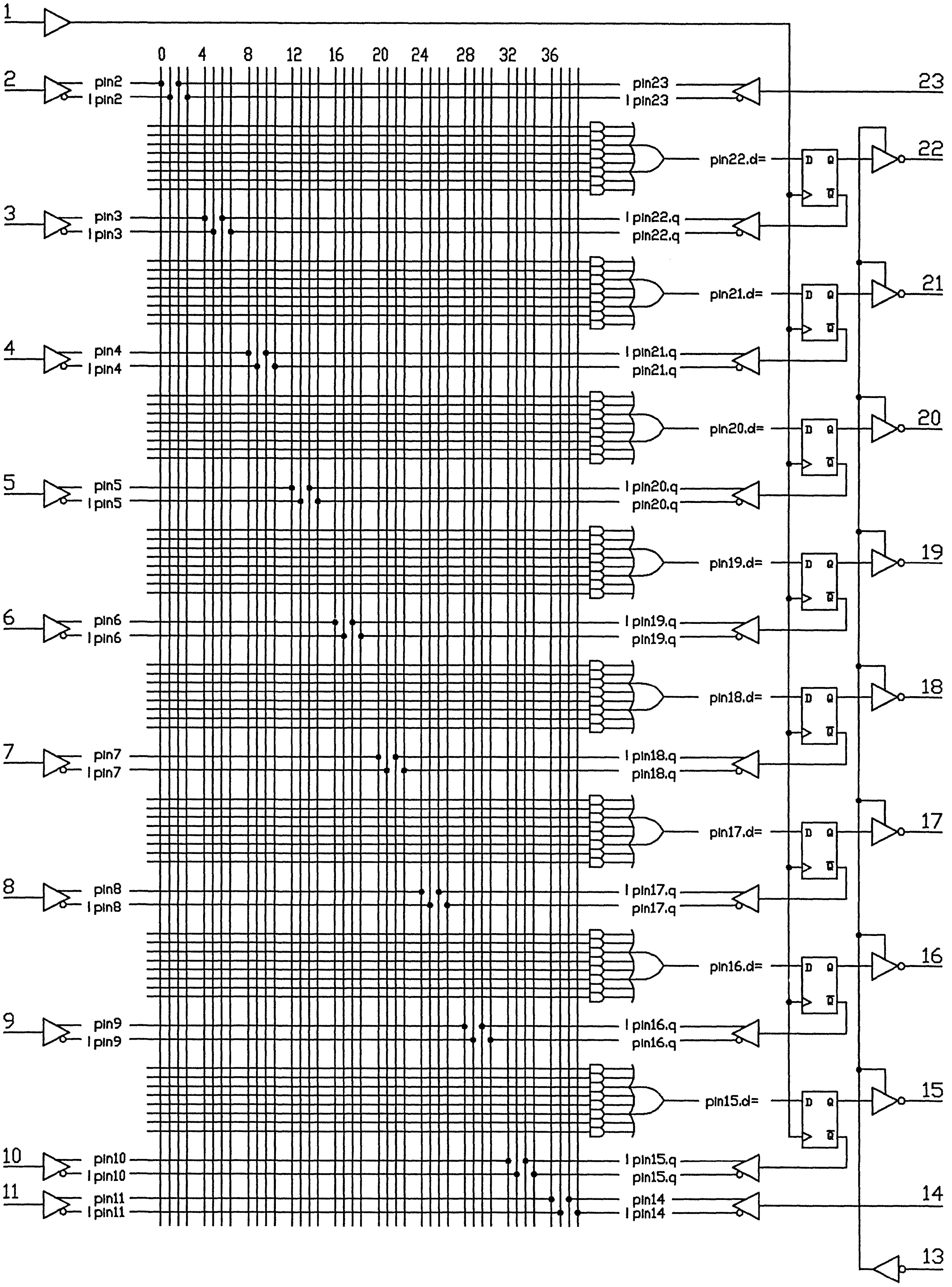
p20l10


1
0
4
8
12
16
20
24
28
32
36
2
pin2
! pin2
pin23
! pin23
23
pin22.oe=
! pin22=
22
3
pin3
! pin3
pin22
! pin22
pin21.oe=
! pin21=
21
4
pin4
! pin4
pin21
! pin21
pin20.d=
D
Q
Q̄
20
5
pin5
! pin5
! pin20.q
pin20.q
pin19.d=
19
6
pin6
! pin6
! pin19.q
pin19.q
pin18.d=
18
7
pin7
! pin7
! pin18.q
pin18.q
pin17.d=
17
8
pin8
! pin8
! pin17.q
pin17.q
pin16.oe=
! pin16=
16
9
pin9
! pin9
pin16
! pin16
pin15.oe=
! pin15=
15
10
pin10
! pin10
pin15
! pin15
11
pin11
! pin11
pin14
! pin14
14
13

p20r4

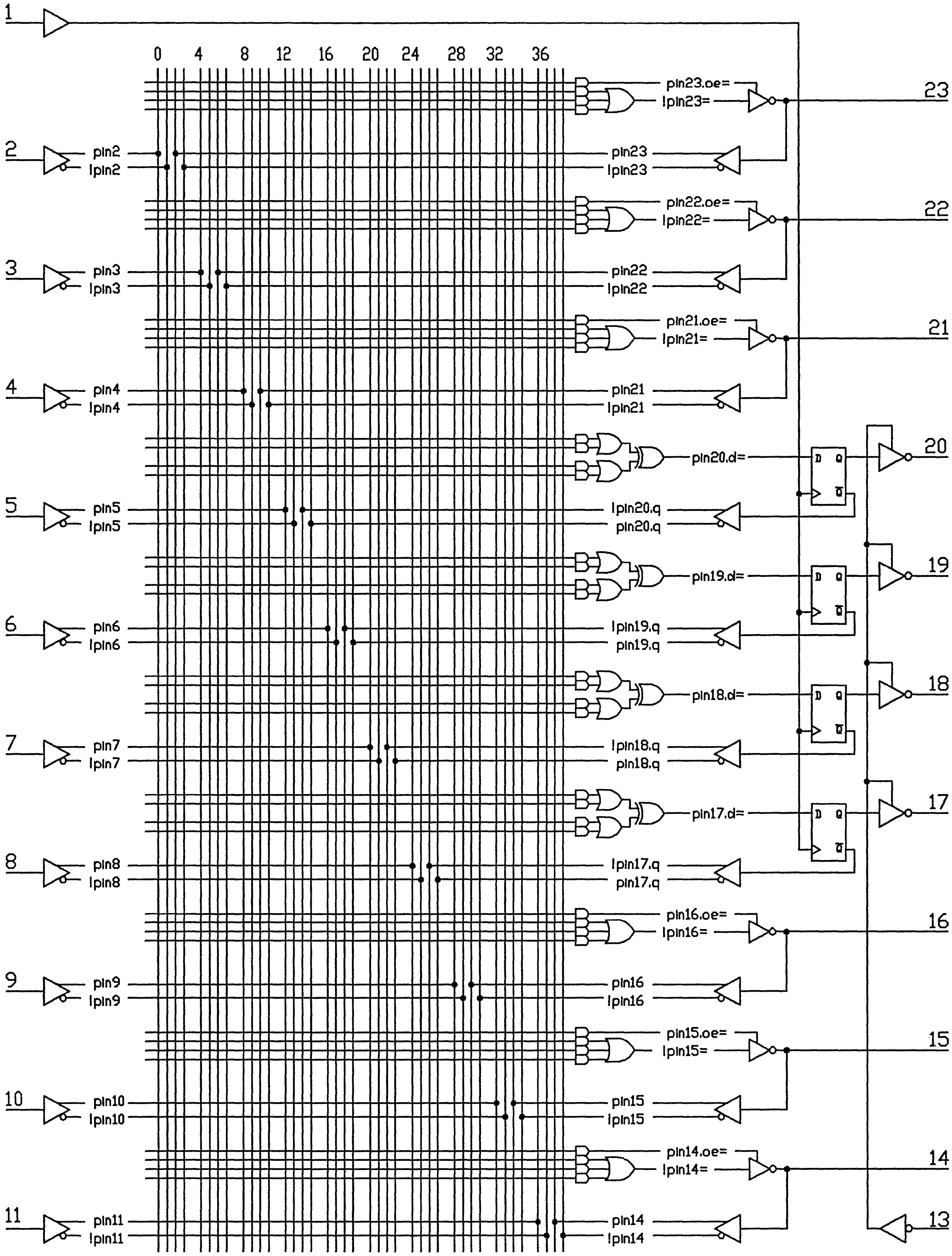

1
2
3
4
5
6
7
8
9
10
11
0
4
8
12
16
20
24
28
32
36
pin2
! pin2
pin3
! pin3
pin4
! pin4
pin5
! pin5
pin6
! pin6
pin7
! pin7
pin8
! pin8
pin9
! pin9
pin10
! pin10
pin11
! pin11
pin23
! pin23
pin22.oe=
! pin22=
pin22
! pin22
pin21.d=
! pin21.q
pin21.q
pin20.d=
! pin20.q
pin20.q
pin19.d=
! pin19.q
pin19.q
pin18.d=
! pin18.q
pin18.q
pin17.d=
! pin17.q
pin17.q
pin16.d=
! pin16.q
pin16.q
pin15.oe=
! pin15=
pin15
! pin15
pin14
! pin14
D
Q
Q̄
23
22
21
20
19
18
17
16
15
14
13

p20r6

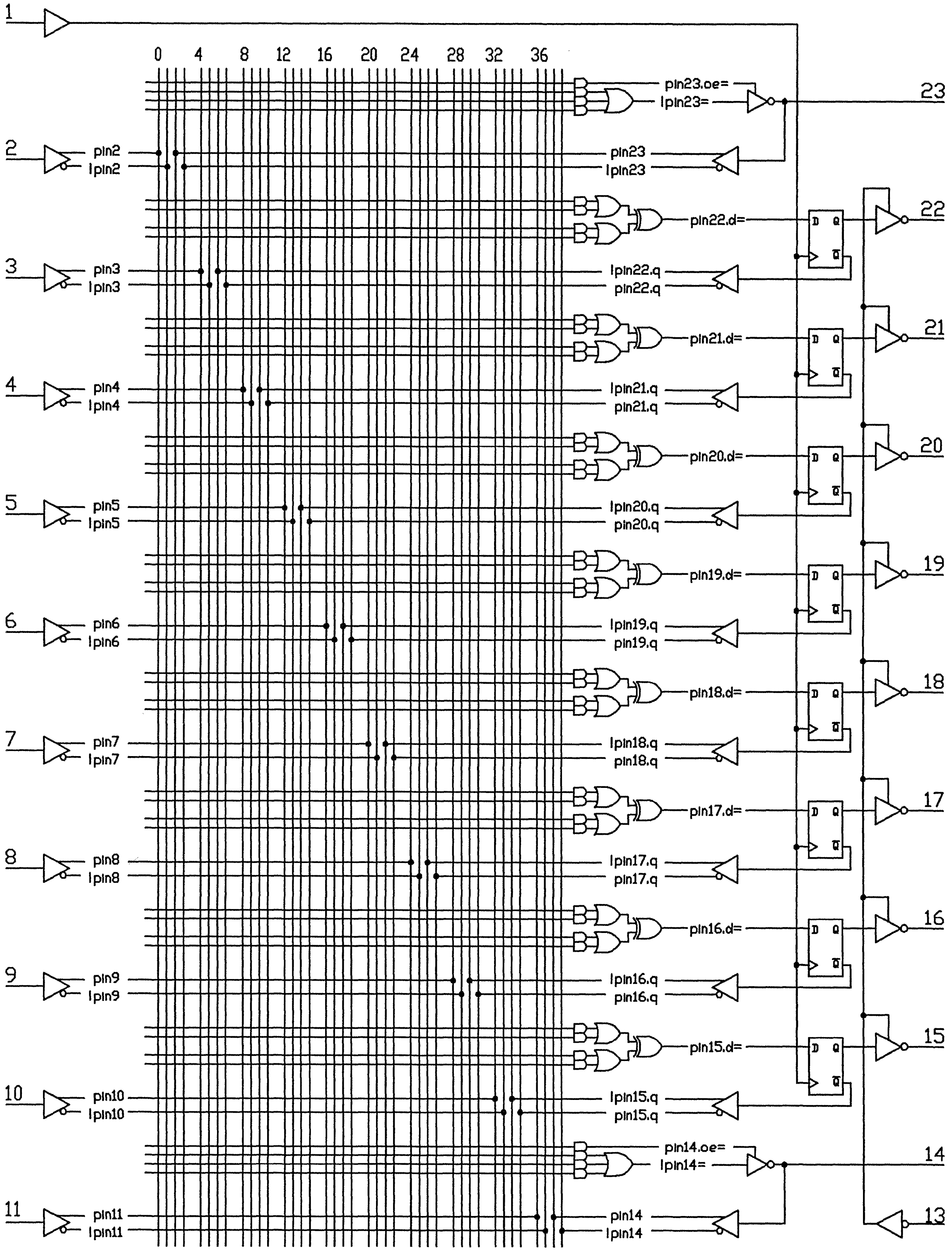


1
2
3
4
5
6
7
8
9
10
11
0
4
8
12
16
20
24
28
32
36
pin2
! pin2
pin3
! pin3
pin4
! pin4
pin5
! pin5
pin6
! pin6
pin7
! pin7
pin8
! pin8
pin9
! pin9
pin10
! pin10
pin11
! pin11
pin23
! pin23
pin22.d=
! pin22.q
pin22.q
pin21.d=
! pin21.q
pin21.q
pin20.d=
! pin20.q
pin20.q
pin19.d=
! pin19.q
pin19.q
pin18.d=
! pin18.q
pin18.q
pin17.d=
! pin17.q
pin17.q
pin16.d=
! pin16.q
pin16.q
pin15.d=
! pin15.q
pin15.q
pin14
! pin14
D
Q
$\overline{Q}$
23
22
21
20
19
18
17
16
15
14
13

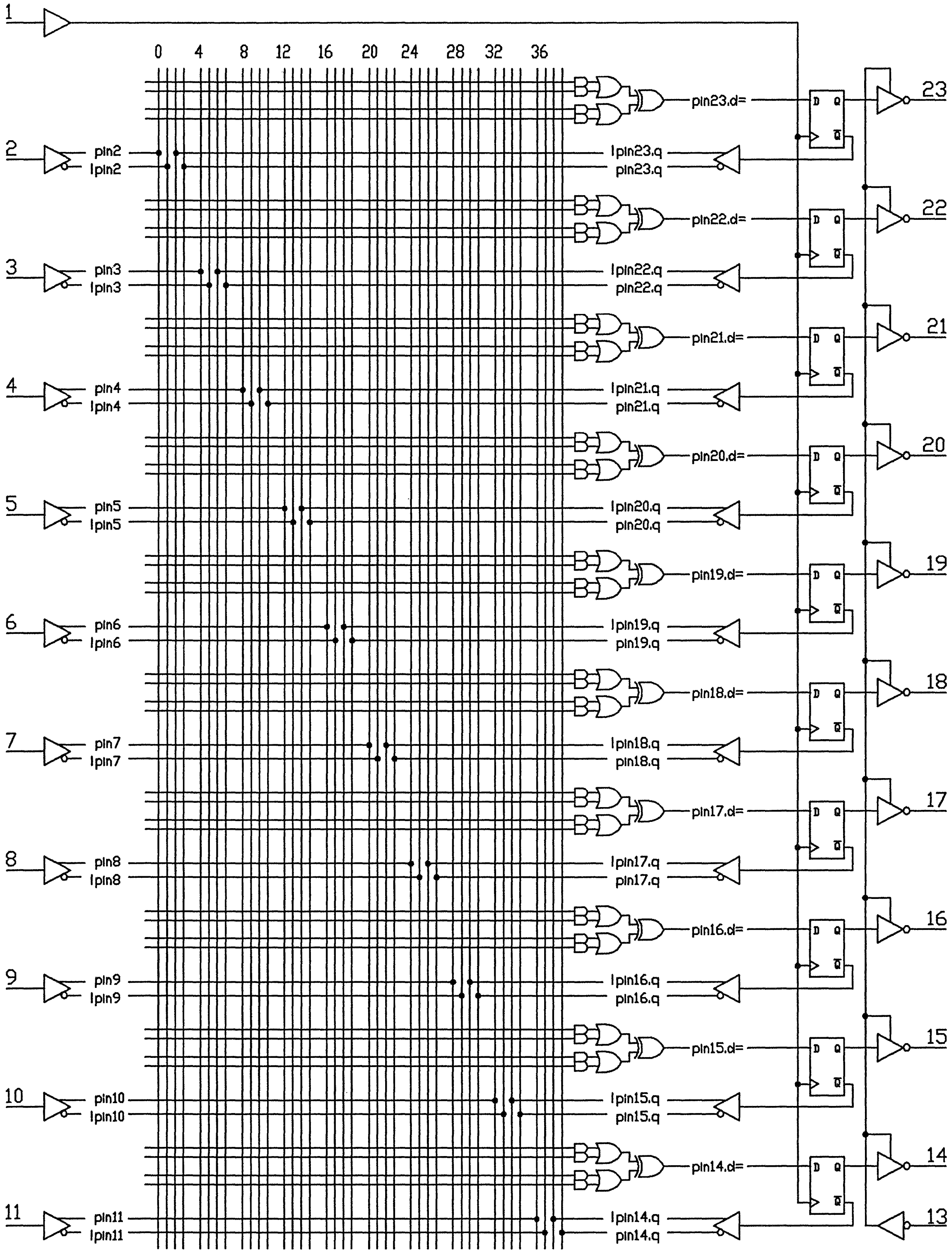
p20r8


1
0
4
8
12
16
20
24
28
32
36
pin23.oe=
!pin23=
23
2
pin2
!pin2
pin23
!pin23
pin22.oe=
!pin22=
22
3
pin3
!pin3
pin22
!pin22
pin21.oe=
!pin21=
21
4
pin4
!pin4
pin21
!pin21
pin20.d=
D Q
20
5
pin5
!pin5
!pin20.q
pin20.q
pin19.d=
19
6
pin6
!pin6
!pin19.q
pin19.q
pin18.d=
18
7
pin7
!pin7
!pin18.q
pin18.q
pin17.d=
17
8
pin8
!pin8
!pin17.q
pin17.q
pin16.oe=
!pin16=
16
9
pin9
!pin9
pin16
!pin16
pin15.oe=
!pin15=
15
10
pin10
!pin10
pin15
!pin15
pin14.oe=
!pin14=
14
11
pin11
!pin11
pin14
!pin14
13

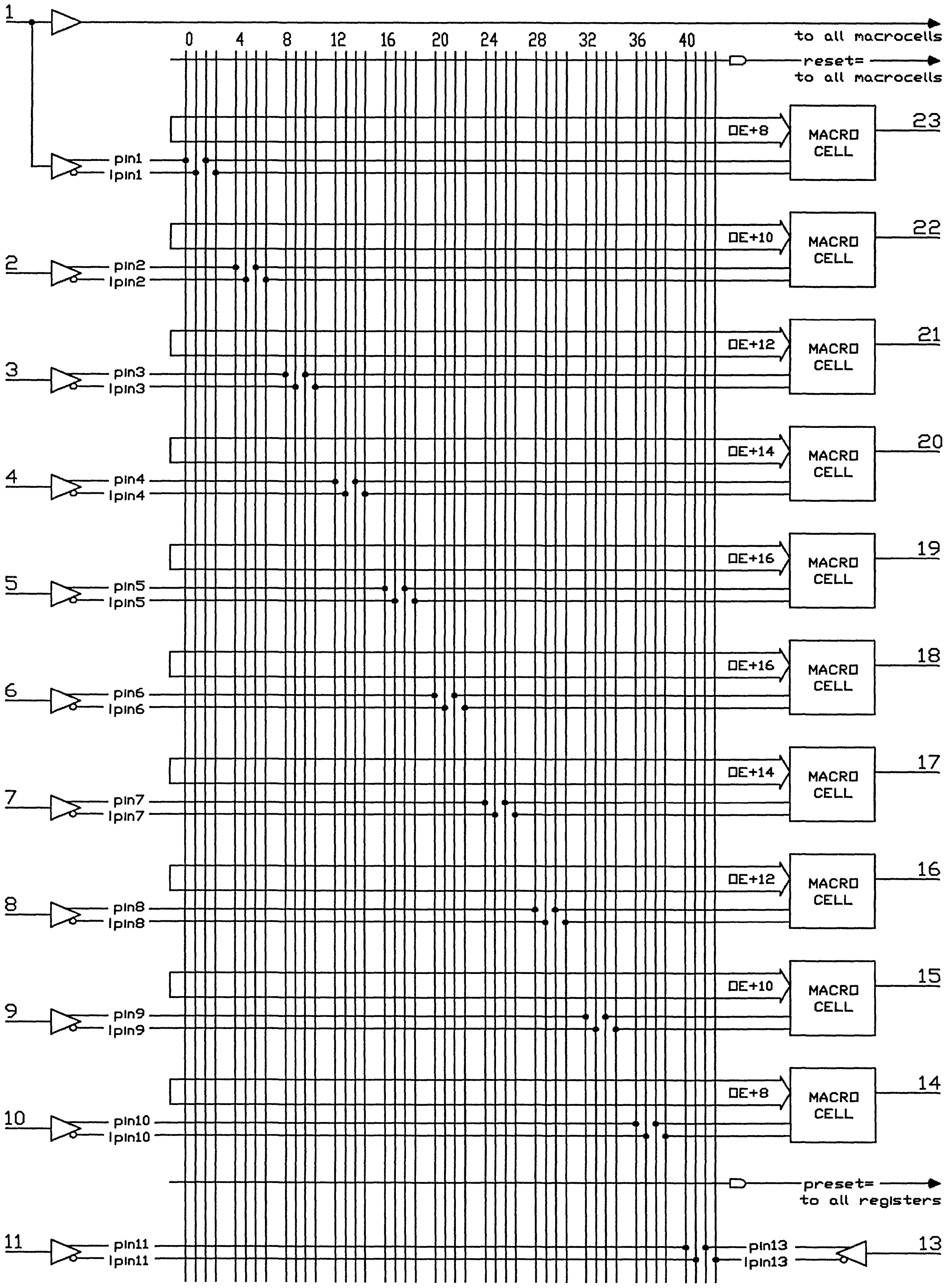
p20x4


1
0 4 8 12 16 20 24 28 32 36
pin23.oe=
!pin23=
23
2
pin2
!pin2
pin23
!pin23
pin22.d=
22
3
pin3
!pin3
!pin22.q
pin22.q
pin21.d=
21
4
pin4
!pin4
!pin21.q
pin21.q
pin20.d=
20
5
pin5
!pin5
!pin20.q
pin20.q
pin19.d=
19
6
pin6
!pin6
!pin19.q
pin19.q
pin18.d=
18
7
pin7
!pin7
!pin18.q
pin18.q
pin17.d=
17
8
pin8
!pin8
!pin17.q
pin17.q
pin16.d=
16
9
pin9
!pin9
!pin16.q
pin16.q
pin15.d=
15
10
pin10
!pin10
!pin15.q
pin15.q
pin14.oe=
!pin14=
14
11
pin11
!pin11
pin14
!pin14
13
D Q
Q

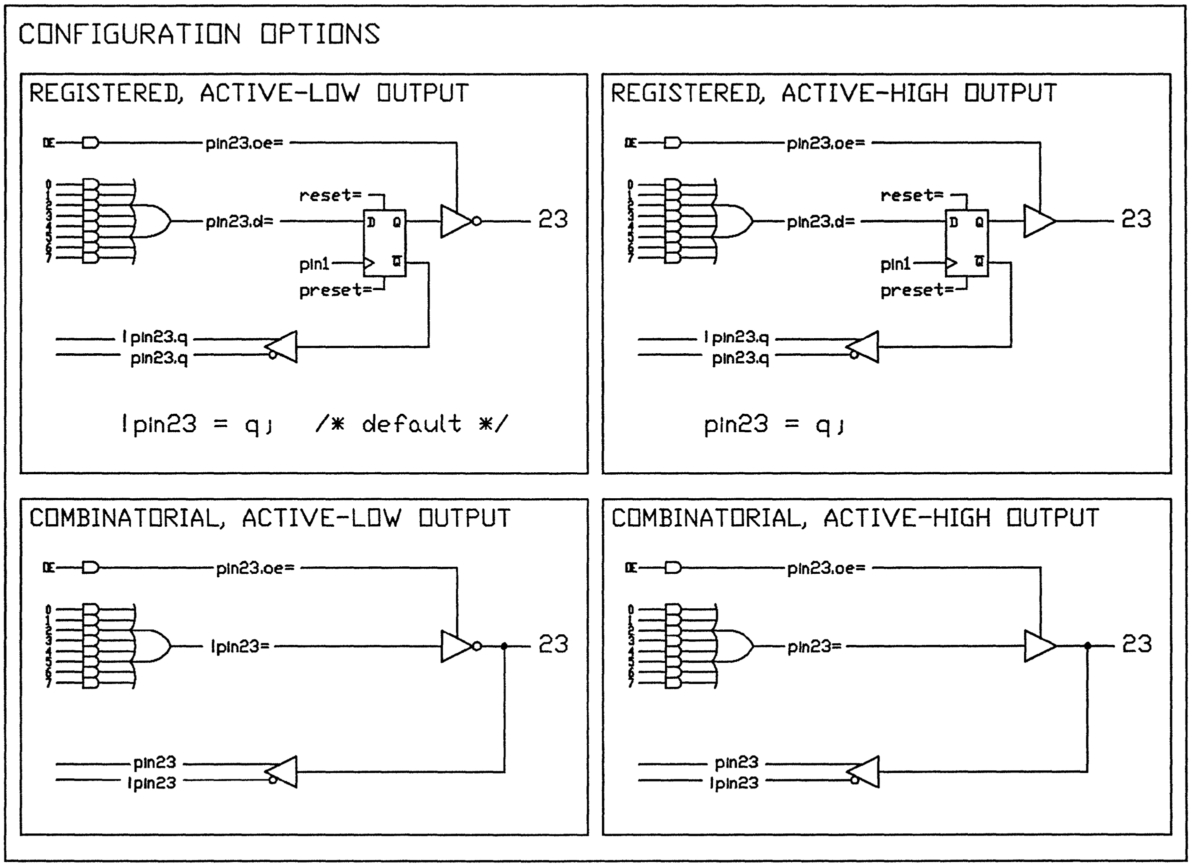
p20x8


1
0
4
8
12
16
20
24
28
32
36
pin23.d=
D Q
23
2
pin2
Ipin2
Ipin23.q
pin23.q
pin22.d=
22
3
pin3
Ipin3
Ipin22.q
pin22.q
pin21.d=
21
4
pin4
Ipin4
Ipin21.q
pin21.q
pin20.d=
20
5
pin5
Ipin5
Ipin20.q
pin20.q
pin19.d=
19
6
pin6
Ipin6
Ipin19.q
pin19.q
pin18.d=
18
7
pin7
Ipin7
Ipin18.q
pin18.q
pin17.d=
17
8
pin8
Ipin8
Ipin17.q
pin17.q
pin16.d=
16
9
pin9
Ipin9
Ipin16.q
pin16.q
pin15.d=
15
10
pin10
Ipin10
Ipin15.q
pin15.q
pin14.d=
14
11
pin11
Ipin11
Ipin14.q
pin14.q
13

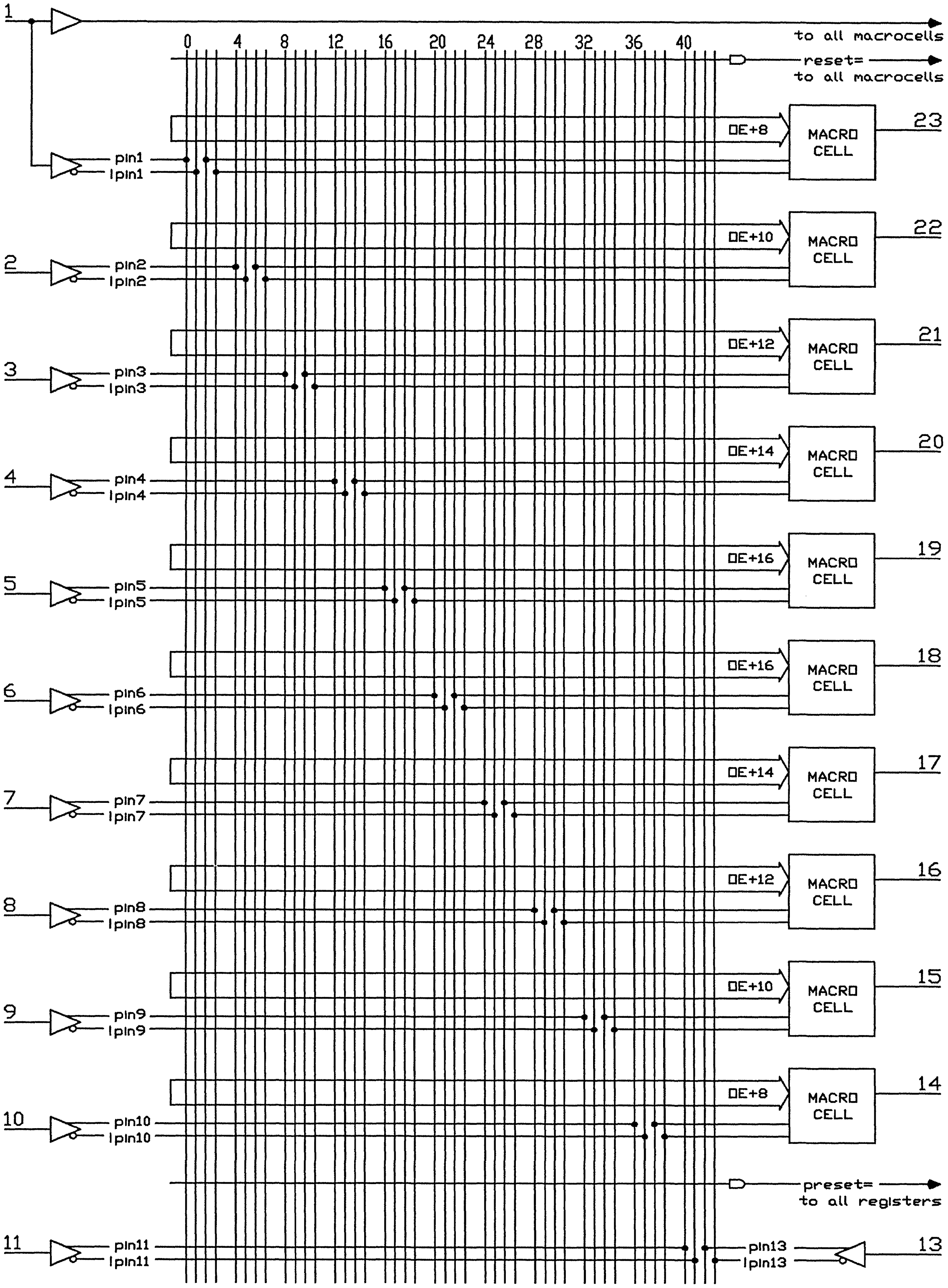
p20x10


1
to all macrocells
0
4
8
12
16
20
24
28
32
36
40
reset=
to all macrocells
OE+8
MACRO CELL
23
pin1
!pin1
OE+10
MACRO CELL
22
2
pin2
!pin2
OE+12
MACRO CELL
21
3
pin3
!pin3
OE+14
MACRO CELL
20
4
pin4
!pin4
OE+16
MACRO CELL
19
5
pin5
!pin5
OE+16
MACRO CELL
18
6
pin6
!pin6
OE+14
MACRO CELL
17
7
pin7
!pin7
OE+12
MACRO CELL
16
8
pin8
!pin8
OE+10
MACRO CELL
15
9
pin9
!pin9
OE+8
MACRO CELL
14
10
pin10
!pin10
preset=
to all registers
11
pin11
!pin11
pin13
!pin13
13

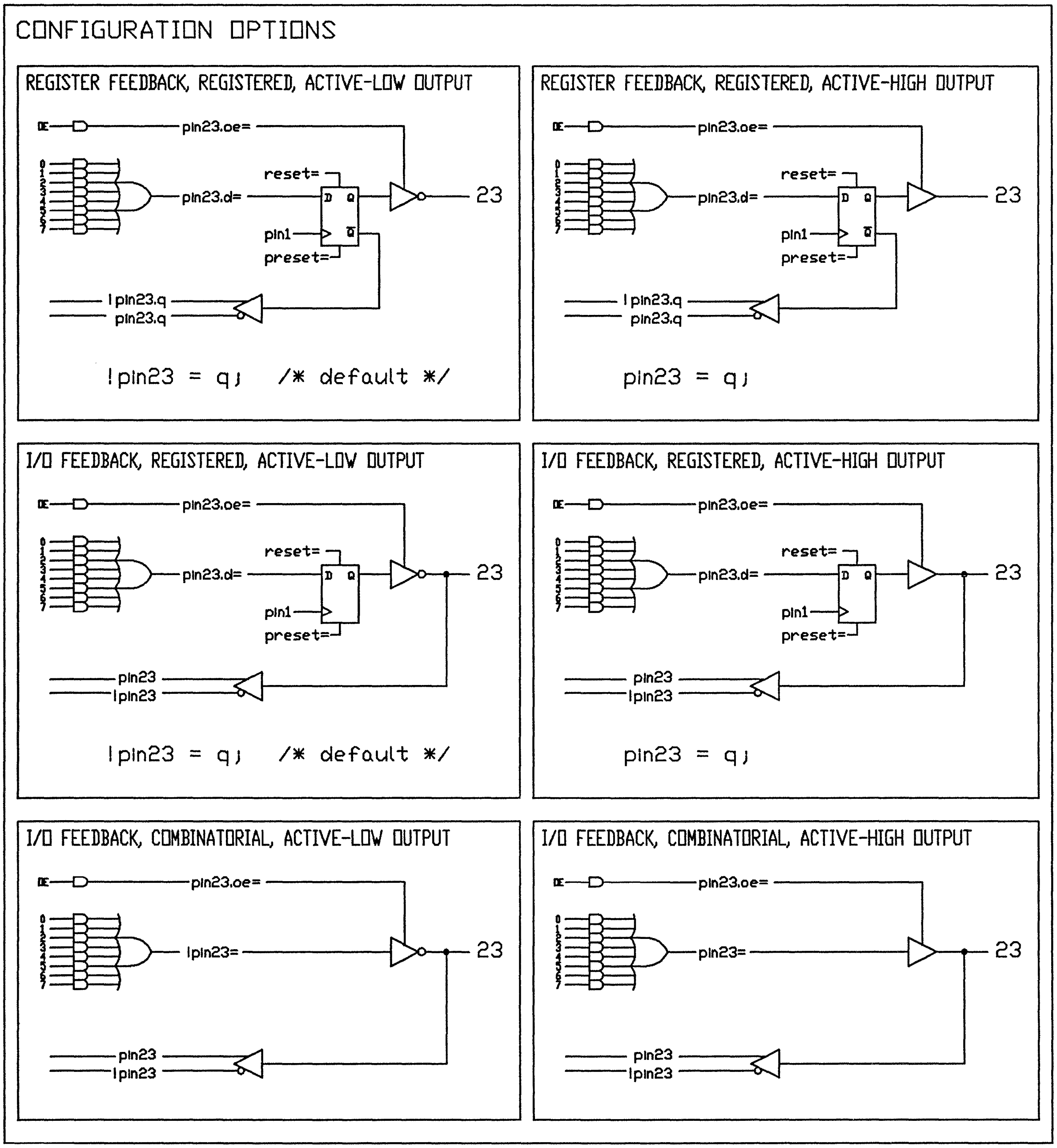
p22v10-1


CONFIGURATION OPTIONS
REGISTERED, ACTIVE-LOW OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
reset=
pin23.d=
D
Q
23
pin1
Q̄
preset=
!pin23.q
pin23.q
!pin23 = q;   /* default */
REGISTERED, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
reset=
pin23.d=
D
Q
23
pin1
Q̄
preset=
!pin23.q
pin23.q
pin23 = q;
COMBINATORIAL, ACTIVE-LOW OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
!pin23=
23
pin23
!pin23
COMBINATORIAL, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
pin23=
23
pin23
!pin23

p22v10-2


1
0
4
8
12
16
20
24
28
32
36
40
to all macrocells
reset=
to all macrocells
OE+8
MACRO CELL
23
pin1
!pin1
OE+10
MACRO CELL
22
2
pin2
!pin2
OE+12
MACRO CELL
21
3
pin3
!pin3
OE+14
MACRO CELL
20
4
pin4
!pin4
OE+16
MACRO CELL
19
5
pin5
!pin5
OE+16
MACRO CELL
18
6
pin6
!pin6
OE+14
MACRO CELL
17
7
pin7
!pin7
OE+12
MACRO CELL
16
8
pin8
!pin8
OE+10
MACRO CELL
15
9
pin9
!pin9
OE+8
MACRO CELL
14
10
pin10
!pin10
preset=
to all registers
11
pin11
!pin11
pin13
!pin13
13

p22vp10-1


CONFIGURATION OPTIONS
REGISTER FEEDBACK, REGISTERED, ACTIVE-LOW OUTPUT
OE
pin23.oe=
0 1 2 3 4 5 6 7
reset=
pin23.d=
D Q
pin1
Q̄
preset=
23
!pin23.q
pin23.q
!pin23 = q; /* default */
REGISTER FEEDBACK, REGISTERED, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
reset=
pin23.d=
D Q
pin1
Q̄
preset=
23
!pin23.q
pin23.q
pin23 = q;
I/O FEEDBACK, REGISTERED, ACTIVE-LOW OUTPUT
OE
pin23.oe=
reset=
pin23.d=
D Q
pin1
preset=
23
pin23
!pin23
!pin23 = q; /* default */
I/O FEEDBACK, REGISTERED, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
reset=
pin23.d=
D Q
pin1
preset=
23
pin23
!pin23
pin23 = q;
I/O FEEDBACK, COMBINATORIAL, ACTIVE-LOW OUTPUT
OE
pin23.oe=
!pin23=
23
pin23
!pin23
I/O FEEDBACK, COMBINATORIAL, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
pin23=
23
pin23
!pin23

p22vp10-2

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 292-2640.

CALIFORNIA: Irvine (714) 660-1200;
Roseville (916) 786-9208;
San Diego (619) 278-9601;
Santa Clara (408) 980-9000;
Torrance (213) 217-7010;
Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Altamonte Springs (305) 260-2116;
Ft. Lauderdale (305) 973-8502;
Tampa (813) 885-7411.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-2925.

INDIANA: Carmel (317) 573-6400;
Ft. Wayne (219) 424-5174.

IOWA: Cedar Rapids (319) 395-9550.

KANSAS: Overland Park (913) 451-4511.

MARYLAND: Columbia (301) 964-2003.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1569;
Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291;
Melville (516) 454-6600;
Pittsford (716) 385-6770;
Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0933;
Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100;
Beaver Creek (513) 427-6200.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Blue Bell (215) 825-9500.

PUERTO RICO: Hato Rey (809) 753-8700.

TENNESSEE: Johnson City (615) 461-2192.

TEXAS: Austin (512) 250-7655;
Houston (713) 778-6592;
Richardson (214) 680-5082;
San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 782-2899.

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 336-1860.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8105;
Santa Clara (408) 748-2220;

GEORGIA: Norcross (404) 662-7945.

ILLINOIS Arlington Heights (312) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9196.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970.

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow/Kieruiff Electronics Group
Arrow (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

–OBSOLETE PRODUCT ONLY–
Rochester Electronics, Inc.
Newburyport, Massachusetts
(508) 462-9332

ALABAMA: Arrow/Kierulff (205) 837-6955;
Hall-Mark (205) 837-8700; Marshall (205) 881-9235;
Schweber (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750;
Hall-Mark (602) 437-1200; Marshall (602) 496-0290;
Schweber (602) 431-0030; Wyle (602) 866-2888.

CALIFORNIA: Los Angeles/Orange County:
Arrow/Kierulff (818) 701-7500, (714) 838-5422;
Hall-Mark (818) 773-4500, (714) 669-4100;
Marshall (818) 407-0101, (818) 459-5500,
(714) 458-5395; Schweber (818) 880-9686;
(714) 863-0200, (213) 320-8090; Wyle (818) 880-9000,
(714) 863-9953; Zeus (714) 921-9000; (818) 889-3838;
Sacramento: Hall-Mark (916) 624-9781;
Marshall (916) 635-9700; Schweber (916) 364-0222;
Wyle (916) 638-5282;
San Diego: Arrow/Kierulff (619) 565-4800;
Hall-Mark (619) 268-1201; Marshall (619) 578-9600;
Schweber (619) 450-0454; Wyle (619) 565-9171;
San Francisco Bay Area: Arrow/Kierulff (408) 745-6600,
Hall-Mark (408) 432-0900; Marshall (408) 942-4600;
Schweber (408) 432-7171; Wyle (408) 727-2500;
Zeus (408) 998-5121.

COLORADO: Arrow/Kierulff (303) 790-4444;
Hall-Mark (303) 790-1662; Marshall (303) 451-8383;
Schweber (303) 799-0258; Wyle (303) 457-9953.

CONNETICUT: Arrow/Kierulff (203) 265-7741;
Hall-Mark (203) 271-2844; Marshall (203) 265-3822;
Schweber (203) 264-4700.

FLORIDA: Ft. Lauderdale:
Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280;
Marshall (305) 977-4880; Schweber (305) 977-7511;
Orlando: Arrow/Kierulff (407) 323-0252;
Hall-Mark (407) 830-5855; Marshall (407) 767-8585;
Schweber (407) 331-7555; Zeus (407) 365-3000;
Tampa: Hall-Mark (813) 530-4543;
Marshall (813) 576-1399; Schweber (813) 541-5100.

GEORGIA: Arrow/Kierulff (404) 449-8252;
Hall-Mark (404) 447-8000; Marshall (404) 923-5750;
Schweber (404) 449-9170.

ILLINOIS: Arrow/Kierulff (312) 250-0500;
Hall-Mark (312) 860-3800; Marshall (312) 490-0155;
Newark (312) 784-5100; Schweber (312) 364-3750.

INDIANA: Indianapolis: Arrow/Kierulff (317) 243-9353;
Hall-Mark (317) 872-8875; Marshall (317) 297-0483;
Schweber (317) 843-1050.

IOWA: Arrow/Kierulff (319) 395-7230;
Schweber (319) 373-1417.

KANSAS: Kansas City: Arrow/Kierulff (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121;
Schweber (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002;
Hall-Mark (301) 988-9800; Marshall (301) 235-9464;
Schweber (301) 840-5900; Zeus (301) 997-1118.

MASSACHUSETTS Arrow/Kierulff (508) 658-0900;
Hall-Mark (508) 667-0902; Marshall (508) 658-0810;
Schweber (617) 275-5100; Time (617) 532-6200;
Wyle (617) 273-7300; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow/Kierulff (313) 462-2290;
Hall-Mark (313) 462-1205; Marshall (313) 525-5850;
Newark (313) 967-0600; Schweber (313) 525-8100;
Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800;
Hall-Mark (612) 941-2600; Marshall (612) 559-2211;
Schweber (612) 941-5280.

MISSOURI: St. Louis: Arrow/Kierulff (314) 567-6888;
Hall-Mark (314) 291-5350; Marshall (314) 291-4650;
Schweber (314) 739-0526.

NEW HAMPSHIRE: Arrow/Kierulff (603) 668-6968;
Schweber (603) 625-2250.

NEW JERSEY: Arrow/Kierulff (201) 538-0900,
(609) 596-8000; GRS Electronics (609) 964-8560;
Hall-Mark (201) 575-4415, (201) 882-9773,
(609) 235-1900; Marshall (201) 882-0320,
(609) 234-9100; Schweber (201) 227-7880.

NEW MEXICO: Arrow/Kierulff (505) 243-4566.

NEW YORK: Long Island:
Arrow/Kierulff (516) 231-1009; Hall-Mark (516) 737-0600;
Marshall (516) 273-2424; Schweber (516) 334-7474;
Zeus (914) 937-7400;
Rochester: Arrow/Kierulff (716) 427-0300;
Hall-Mark (716) 425-3300; Marshall (716) 235-7620;
Schweber (716) 424-2222;
Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Marshall (919) 878-9882; Schweber (919) 876-0000.

OHIO: Cleveland: Arrow/Kierulff (216) 248-3990;
Hall-Mark (216) 349-4632; Marshall (216) 248-1788;
Schweber (216) 464-2970;
Columbus: Hall-Mark (614) 888-3313;
Dayton: Arrow/Kierulff (513) 435-5563;
Marshall (513) 898-4480; Schweber (513) 439-1800.

OKLAHOMA: Arrow/Kierulff (918) 252-7537;
Schweber (918) 622-8003.

OREGON: Arrow/Kierulff (503) 645-6456;
Marshall (503) 644-5050; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow/Kierulff (412) 856-7000,
(215) 928-1800; GRS Electronics (215) 922-7037;
Marshall (412) 963-0441; Schweber (215) 441-0600,
(412) 963-6804.

TEXAS: Austin: Arrow/Kierulff (512) 835-4180;
Hall-Mark (512) 258-8848; Marshall (512) 837-1991;
Schweber (512) 339-0088; Wyle (512) 834-9957;
Dallas: Arrow/Kierulff (214) 380-6464;
Hall-Mark (214) 553-4300; Marshall (214) 233-5200;
Schweber (214) 661-5010; Wyle (214) 235-9953;
Zeus (214) 783-7010;
El Paso: Marshall (915) 593-0706;
Houston: Arrow/Kierulff (713) 530-4700;
Hall-Mark (713) 781-6100; Marshall (713) 895-9200;
Schweber (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913;
Hall-Mark (801) 972-1008; Marshall (801) 485-1551;
Wyle (801) 974-9953.

WASHINGTON: Arrow/Kierulff (206) 575-4420;
Marshall (206) 486-5747; Wyle (206) 881-1150.

WISCONSIN: Arrow/Kierulff (414) 792-0150;
Hall-Mark (414) 797-7844; Marshall (414) 797-8400;
Schweber (414) 784-9020.

CANADA: Calgary: Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 871-7500;
Toronto: Arrow Canada (416) 672-7769;
Future (416) 638-4771; Marshall (416) 674-2161;
Vancouver: Arrow Canada (604) 291-2986;
Future (604) 294-1166.

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. – 5:00 p.m. CST)

TEXAS INSTRUMENTS

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514,
Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix,
AZ 85021, (602) 995-1007;**TUCSON:** 818 W. Miracle
Mile, Suite 43, Tucson, AZ 85705, (602) 292-2640.

CALIFORNIA: Irvine: 17891 Cartwright Dr., Irvine, CA
92714, (714) 660-1200; **Roseville:** 1 Sierra Gate
Plaza, Roseville, CA 95678, (916) 786-9208;
San Diego: 4333 View Ridge Ave., Suite 100,
San Diego, CA 92123, (619) 278-9601;
Santa Clara: 5353 Betsy Ross Dr., Santa Clara, CA
95054, (408) 980-9000; **Torrance:** 690 Knox St.,
Torrance, CA 90502, (213) 217-7010;
Woodland Hills: 21220 Erwin St., Woodland Hills,
CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave.,
Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park
Rd., Barnes Industrial Park, Wallingford,
CT 06492, (203) 269-0074.

FLORIDA: Altamonte Springs: 370 S. North Lake Blvd,
Altamonte Springs, FL 32701, (305) 260-2116;
Ft. Lauderdale: 2950 N.W. 62nd St.,
Ft. Lauderdale, FL 33309, (305) 973-8502;
Tampa: 4803 George Rd., Suite 390,
Tampa, FL 33634, (813) 885-7411.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross,
GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin,
Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr.,
Ft. Wayne, IN 46815, (219) 424-5174;
Carmel: 550 Congressional Dr., Carmel, IN 46032,
(317) 573-6400.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 201,
Cedar Rapids, IA 52402, (319) 395-9550.

KANSAS: Overland Park: 7300 College Blvd., Lighton
Plaza, Overland Park, KS 66210, (913) 451-4511.

MARYLAND: Columbia: 8815 Centre Park Dr.,
Columbia MD 21045, (301) 964-2003.

MASSACHUSETTS: Waltham: 950 Winter St.,
Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd.,
Farmington Hills, MI 48018, (313) 553-1569.
Grand Rapids: 3075 Orchard Vista Dr. S.E.,
Grand Rapids, MI 49506, (616) 957-4200.

MINNESOTA: Eden Prairie: 11000 W. 78th St.,
Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: St. Louis: 11816 Borman Drive,
St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South,
Parkway Towers, Iselin, NJ 08830 (201) 750-1050.

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy
NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr.,
East Syracuse, NY 13057, (315) 463-9291;
Melville: 1895 Walt Whitman Rd., P.O. Box 2936,
Melville, NY 11747, (516) 454-6600;
Pittsford: 2851 Clover St., Pittsford, NY 14534,
(716) 385-6770;
Poughkeepsie: 385 South Rd., Poughkeepsie,
NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green,
Woodlawn Rd., Charlotte, NC 28210, (704)
527-0933; **Raleigh**: 2809 Highwoods Blvd., Suite 100,
Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23775 Commerce Park Rd.,
Beachwood, OH 44122, (216) 464-6100;
Beavercreek: 4200 Colonel Glenn Hwy.,
Beavercreek, OH 45431, (513) 427-6200.

OREGON: Beaverton: 6700 SW 105th St., Suite 110,
Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Blue Bell: 670 Sentry Pkwy,
Blue Bell, PA 19422, (215) 825-9500.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg.,
Suite 505, Hato Rey, PR 00918, (809) 753-8700.

TENNESSEE: Johnson City: Erwin Hwy,
P.O. Drawer 1255, Johnson City, TN 37605
(615) 461-2192.

TEXAS: Austin: 12501 Research Blvd., Austin, TX
78759, (512) 250-7655; **Richardson:** 1001 E.
Campbell Rd., Richardson, TX 75081,
(214) 680-5082; **Houston:** 9100 Southwest Frwy.,
Suite 250, Houston, TX 77074, (713) 778-6592;
San Antonio: 1000 Central Parkway South,
San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green St., Suite 200,
Murray, UT 84123, (801) 266-8972.

WASHINGTON: Redmond: 5010 148th NE, Bldg B,
Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite
150, Brookfield, WI 53005, (414) 782-2899.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center,
Nepean, Ontario, Canada, K2H9C4,
(613) 726-1970. **Richmond Hill:** 280 Centre St. E.,
Richmond Hill L4C1B1, Ontario, Canada
(416) 884-9181; **St. Laurent:** Ville St. Laurent
Quebec, 9460 Trans Canada Hwy., St. Laurent,
Quebec, Canada H4S1R7, (514) 336-1860.

ARGENTINA: Texas Instruments Argentina Viamonte
1119, 1053 Capital Federal, Buenos Aires, Argentina,
541/748-3699

AUSTRALIA (& NEW ZEALAND): Texas Instruments
Australia Ltd.: 6-10 Talavera Rd., North Ryde
(Sydney), New South Wales, Australia 2113,
2 + 887-1122; 5th Floor, 418 St. Kilda Road,
Melbourne, Victoria, Australia 3004, 3 + 267-4677;
171 Philip Highway, Elizabeth, South Australia 5112,
8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.:
Industriestrabe B/16, A-2345 Brunn/Gebirge,
2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: 11,
Avenue Jules Bondetlaan 11, 1140 Brussels, Belgium,
(02) 242-3080.

BRAZIL: Texas Instruments Electronicos do Brasil
Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424
Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E,
2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY:
Ahertajantie 3, P.O. Box 81, ESPOO, Finland, (90)
0-461-422.

FRANCE: Texas Instruments France: Paris Office, BP
67 8-10 Avenue Morane-Saulnier, 78141 Velizy-
Villacoublay cedex (1) 30 70 1003.

GERMANY (Fed. Republic of Germany): Texas
Instruments Deutschland GmbH: Haggertystrasse 1,
8050 Freising, 8161+80-4591; Kurfuerstendamm
195/196, 1000 Berlin 15, 30+882-7365; III, Hagen
43/Kibbelstrasse, .19, 4300 Essen, 201-24250;
Kirchhorsterstrasse 2, 3000 Hannover 51,
511+648021; Maybachstrabe 11, 7302 Ostfildern
2-Nelingen, 711+34030.

TEXAS INSTRUMENTS

HONG KONG: Texas Instruments Hong Kong Ltd., 8th
Floor, World Shipping Ctr., 7 Canton Rd., Kowloon,
Hong Kong, (852) 3-7351223.

IRELAND: Texas Instruments (Ireland) Limited:
7/8 Harcourt Street, Stillorgan, County Dublin, Eire,
1 781677.

ITALY: Texas Instruments Italia S.p.A. Divisione
Semiconduttori: Viale Europa, 40, 20093 Cologne
Monzese (Milano), (02) 253001; Via Castello della
Magliana, 38, 00148 Roma, (06) 5222651;
Via Amendola, 17, 40100 Bologna, (051) 554004.

JAPAN: Tokyo Marketing/Sales (Headquarters):
Texas Instruments Japan Ltd., MS Shibaura Bldg., 9F,
4-13-23 Shibaura, Minato-ku, Tokyo 108, Japan,
03-769-8700. Texas Instruments Japan Ltd.: Nissho-
Iwai Bldg. 5F, 30 Imabashi 3-chome, Higashi-ku,
Osaka 541, Japan, 06-294-1881; Daini Toyota West
Bldg. 7F, 10-27 Meieki 4-chome, Nakamura-ku,
Nagoya 450, 052-583-8691; Daiichi Seimei Bldg. 6F,
3-10 Oyama-cho, Kanazawa 920, Ishikawa-ken,
0762-23-5471; Daiichi Olympic Tachikawa Bldg. 6F,
1-25-12 Akebono-cho, Tachikawa 190, Tokyo,
0425-27-6426; Matsumoto Showa Bldg. 6F, 2-11
Fukashi 1-chome, Matsumoto 390, Nagano-ken,
0263-33-1060; Yokohama Nishiguchi KN Bldg. 6F,
2-8-4 Kita-Saiwai-cho, Nishi-ku, Yokohama 220,
045-322-6741; Nihon Seimei Kyoto Yasaka Bldg. 5F,
843-2 Higashi Shiokohjidori, Nishinotoh-in Higashi-iru,
Shiokouji, Shimogyo-ku, Kyoto 600, 075-341-7713;
2597-1, Aza Harudai, Oaza Yasaka, Kitsuki 873, Oita-ken, 09786-3-3211; Miho Plant, 2350 Kihara Miho-mura, Inashiki-gun 300-04, Ibaragi-ken,
0298-85-2541.

KOREA: Texas Instruments Korea Ltd., 28th Fl., Trade
Tower, #159, Samsung-Dong, Kangnam-ku, Seoul,
Korea 2+551-2810.

MEXICO: Texas Instruments de Mexico S.A.: Alfonso
Reyes—115, Col. Hipodromo Condesa, Mexico, D.F.,
Mexico 06120, 525/525-3860.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor
Mannai Bldg., Diplomatic Area, P.O. Box 26335,
Manama Bahrain, Arabian Gulf, 973+274681.

NETHERLANDS: Texas Instruments Holland B.V.,
19 Hogehilweg, 1100 AZ Amsterdam—Zuidoost,
Holland 20+5602911.

NORWAY: Texas Instruments Norway A/S: PB106,
Refstad 0585, Oslo 5, Norway, (2) 155090.

PEOPLES REPUBLIC OF CHINA: Texas Instruments
China Inc., Beijing Representative Office, 7-05 Citic
Bldg., 19 Jianguomenwai Dajje, Beijing, China, (861)
5002255, Ext. 3750.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor,
Ba- Lepanto Bldg., Paseo de Roxas, Makati, Metro
Manila, Philippines, 817-60-31.

PORTUGAL: Texas Instruments Equipamento
Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich,
2650 Moreira Da Maia, 4470 Maia, Portugal,
2-948-1003.

**SINGAPORE (+ INDIA, INDONESIA, MALAYSIA,
THAILAND)**: Texas Instruments Singapore (PTE) Ltd.,
Asia Pacific Division, 101 Thompson Rd. #23-01,
United Square, Singapore 1130, 350-8100.

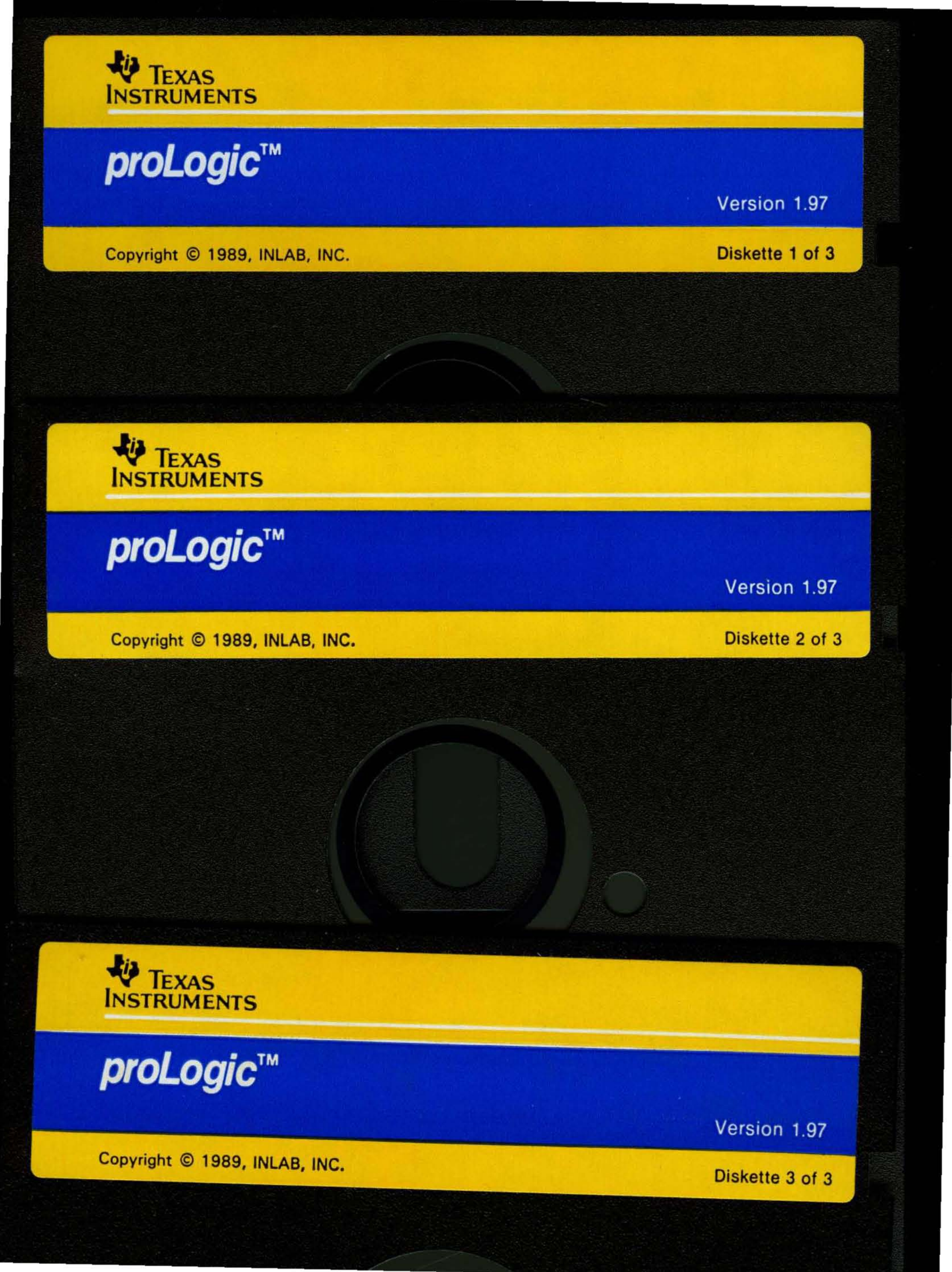
SPAIN: Texas Instruments Espana, S.A.: C/Jose
Lazaro Galdiano No. 6, Madrid 28036, 1/458.14.58.

SWEDEN: Texas Instruments International Trade
Corporation (Sverigefilialen): S-164-93, Stockholm,
Sweden, 8 - 752-5800.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse
6, CH-8953 Dietikon (Zuerich) Switzerland,
1-740 2220.

TAIWAN: Texas Instruments Supply Co., 9th Floor
Bank Tower, 205 Tun Hwa N. Rd., Taipei, Taiwan,
Republic of China, 2 + 713-9311.

UNITED KINGDOM: Texas Instruments Limited:
Manton Lane, Bedford, MK41 7PA, England, 0234
270111.



TEXAS INSTRUMENTS

proLogic™

Version 1.97

Copyright © 1989, INLAB, INC.

Diskette 1 of 3

TEXAS INSTRUMENTS

proLogic™

Version 1.97

Copyright © 1989, INLAB, INC.

Diskette 2 of 3

TEXAS INSTRUMENTS

proLogic™

Version 1.97

Copyright © 1989, INLAB, INC.

Diskette 3 of 3

TEXAS
INSTRUMENTS

Printed in U.S.A.
0989—10