Lattice™
Semiconductor
Corporation

Using GAL® Development Tools

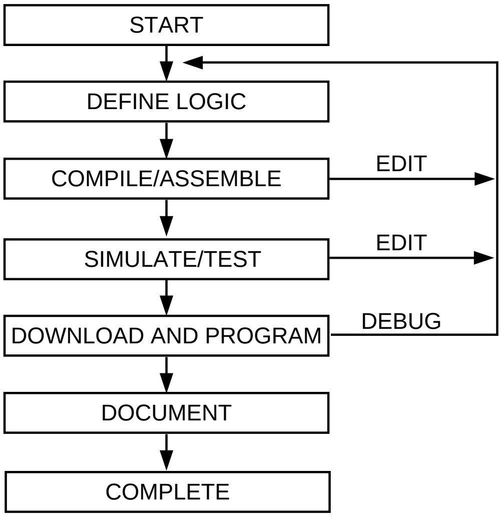
GAL Hardware and Software Tools

Lattice Semiconductor specializes in the design and manufacture of high-speed E^{2}CMOS® programmable logic devices. While we distribute the ABEL compiler with our Synario software solutions, we've left the task of developing GAL design software and programming hardware to the respective third-party experts in those fields.

Such universal tool suppliers provide support for all major devices, from a variety of manufacturers. If you're just starting out with programmable logic and plan to purchase development tools, rest assured that industry-standard hardware and software will handle GAL device development. If you're already using standard tools for PLD development, the move to GAL devices won't require sophisticated or expensive upgrades; current third-party development tools support Lattice GAL devices to their full extent. At most, an upgrade to the current revision of the support tool may be required.

Lattice's Applications Department remains on call to assist you in the task of logic development using third-party tools. Our engineers, trained on a variety of standard equipment, are prepared to answer any questions you may have. In addition, they are able to use the tools to more fully exploit the unique benefits of GAL devices. Here we provide the basis for getting started with GAL devices. As you proceed with the development of your applications, call us — we'd like to hear how it's going.

Figure 1. PLD Design Flow



The typical PLD design flow, shown in Figure 1, begins with a design specification, iterates the logic to achieve proper functionality, and ends with a 'download' of the information to a programming fixture that patterns the device for the system. Critical to the accuracy and ultimate success of the PLD design process is the use of logic development tools to minimize the chance of error and improve design efficiency.

Software Tools

In the early days of programmable logic devices, fuse maps were entered by hand on a piece of paper with a fuse map table, and then manually transferred into the hardware programmers. This basic concept of fuse map generation is still valid for modern devices, such as Lattice's GAL devices or any other PLDs, but the software tools have greatly advanced from those early days. Although the final result of any software tool is the generation of a fuse map, there are many methods in which logic designs can be entered. Most third-party logic compiler software environments offer Boolean equation, truth table and state machine design entry methods. These three basic design entry methods vastly improved the efficiency and accuracy of logic design. To further improve efficiency of design entry, newer software packages offer schematic entry, macro library, timing waveform and hardware description language (HDL) design entry methods. By combining these multiple design entry methods within each third-party software package, system design engineers are able to select the method that suits his or her logic design. Accuracy of the logic implementation is further improved by the ability to perform functional and timing simulation within the software.

Software packages, such as Synario/ABEL from Data I/O, CUPL from Logical Devices, PLDesigner from Minc and PLD 386+ from OrCAD, provide various combinations of the above mentioned design entry methods. With these software packages, the required logic design can be fully designed, simulated, and debugged using software, before any hardware is built. All of these software packages perform a similar function. They process and synthesize the design idea entered by the specified method, convert this result into an intermediate file, such as netlist or PLA file, and finally generate the fuse map file for the programmer. As part of this process, a documentation file is also created. As we move into the future with higher density PLDs, third-party supplied universal software tools are becoming an integral part of the hardware.

Hardware Tools

The same arguments as those expressed above for universal software tools apply to universal hardware. Hardware developed by third parties is more flexible and provides a future growth path to the user. While Lattice recommends the use of In-System Programmable™ devices for almost any application, third-party programming hardware for standard GAL device programming is quite mature and of high quality.

Universal programming hardware allows the programming of a variety of devices without the aid of custom fixtures or manufacturers' adapters. Since the GAL programming algorithm requires no abnormal voltages or timings, as some one-time programmable technologies do, most all hardware manufacturers support GAL devices on existing models.

Patterning the PLD is the process of providing it with the data (the JEDEC file) to perform a specific custom function, and applying the appropriate series of voltage pulses. 'Support' by the hardware manufacturer refers to his ability to provide the appropriate voltage pulses and timings for a given PLD. After that, patterning a device merely requires downloading the JEDEC file.

Downloading is the process of 'teaching' the hardware programmer the pattern that is necessary to program a device. This data can come from a pre-patterned device (or 'master'), from a computer via direct connection or modem or from an attached peripheral, such as a floppy disk. If the file is transferred in JEDEC format, as most are, a checksum is calculated and verified at the end of the data transfer to ensure that no data was dropped or garbled during transmission. Most programmers have either a single button or simple command string that puts the hardware into the download mode.

The programming of the GAL device is controlled by the programming hardware. Since the GAL device uses a nonvolatile, reprogrammable E^2CMOS technology, the device can be erased. In fact, the device is automatically erased as the first step in the programming algorithm.

The patterning of the GAL device array is done using a parallel-programming scheme, which keeps the total programming time to well under a second. The algorithm is so efficient that it programs devices nearly 50% faster than typical bipolar PLD algorithms, and an order of magnitude faster than UV-CMOS approaches. During this programming time, both the logic array and the architecture matrix are patterned. Finally, an analog verify of each and every cell in the GAL device takes place, to ensure that the cell is fully programmed and will retain data for a minimum of 20 years.

It is worth noting here that GAL devices offer a security cell that can be programmed to prevent examination (or further verification) of the pattern in the programmable arrays — a feature provided so that a proprietary design can be obscured from competitive or enemy eyes.

Somewhat ironically, the GAL device security cell is itself erasable. It can only be erased, however, in conjunction with an array 'Bulk Erase,' during which all bits are cleared at once. This allows the designer or manufacturing person to reuse previously secured devices — a feature never before available in PLDs.

Debugging and Pattern Revisions

GAL devices bring extensive advantages to the manufacturing and design engineering areas, due to their unique combination of E^2CMOS technology, generic architecture, and unmatched quality levels. Only GAL devices are instantly erasable in a standard hardware programming fixture. As mentioned, erasure takes place automatically just prior to the re-patterning of the array. No time-consuming trips to a UV lamp are necessary, as with UV-erasable PLDs. Both the GAL device's logic array and device architecture configuration are fully reprogrammable and reconfigurable. In addition, the erasable GAL device is assembled in a low-cost plastic package, not an expensive quartz-windowed package. Pattern revisions can be recorded in the device's electronic signature, allowing the traceability, tracking and verification of every device. Finally, inventories are kept to a minimum, thanks to the generic 'one device fits all' macrocell approach.

The Design Process

By choosing generic, compiler-based software, generic hardware and generic silicon (Lattice GAL devices), the biggest decisions in the design process have already been made. The choice of the appropriate programmable logic device has traditionally been a difficult first step in starting a design, since with bipolar PLDs, you must guess which one of the dozens of architectures has the right combination of outputs, I/Os and registers. If your choice is wrong, you must guess again. The Lattice GAL concept simplifies the approach, requiring that you merely count the number of inputs and outputs, then select a speed/power option. The development software automatically and dynamically allocates the inputs, I/Os, registers and so on.

The following design example shows how to implement two basic logic gates in a GAL device. The specific syntax is that of ABEL-HDL; however, other generic software (CUPL) has similar syntax and functions. In this cursory ‘walk-through,’ segments of code are presented as they would appear on the screen of a personal computer running Synario/ABEL software. The manufacturers of the software would, of course, be glad to provide a more comprehensive tutorial.

All the design processes are envoked from the Synario/ABEL design environment. The Synario Project Navigator allows the user to select devices, edit source files and schematics, and keep track of simulation vectors and results in a single design tracking environment.

Once you open a new design file and select a device, as shown in Figure 2, fields are provided for optional information, such as the design title and source file names.

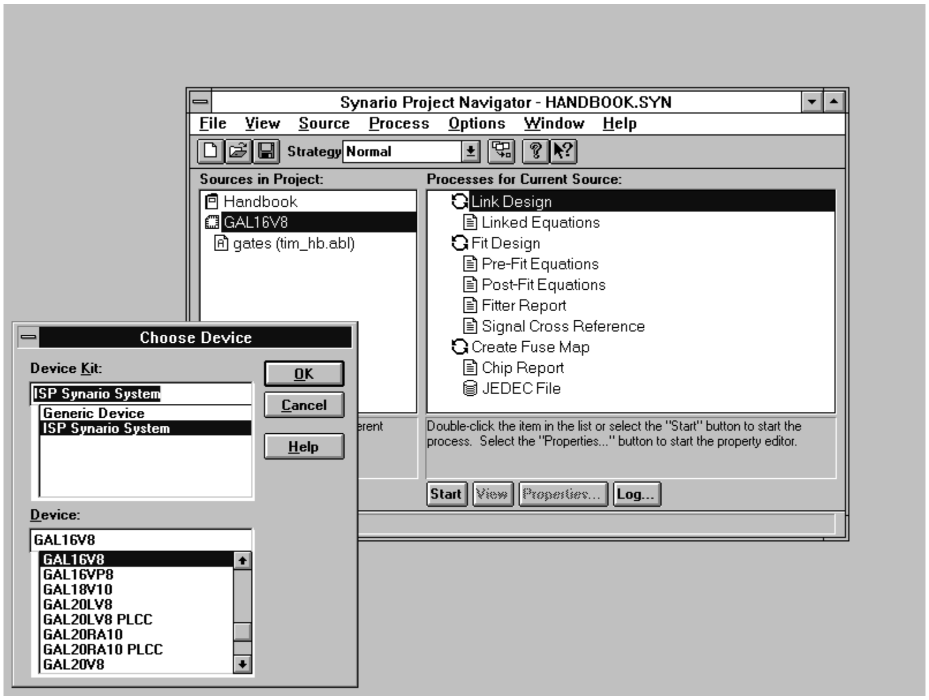
The device, its pinout, pin labels, and intermediate variables need to be specified next. Use names that are convenient for you to reference, since the software doesn’t care what you call a pin, as long as you are consistent:

```
  gates device 'p16v8';
"**** inputs ****
        A, !B  PIN 1,2;
"**** outputs ****
        Y, !Z  PIN 18,19 ISTYPE
'COM,INVERT';
"**** intermediate definitions ****
        C,X,H,L=.C.,.X.,1,0;
```

It’s a good idea to specify pin names in a format consistent with the actual pin state. In the above pin definitions, signals A and Y are active-high, while B and Z are active-low. We have chosen to indicate active-low data signals by prefixing the labels with exclamation points in the definition statement. The use of an active-high variation of these signals in subsequent design statements will automatically be resolved by the software compiler.

Entry of the logic functions is next. This entry is in the form of Boolean equations, truth-table, state machine and schematic-entry formats.

Figure 2. Synario Project Navigator and Device Selection Dialog Box



Strategy: Normal

Double-click the item in the list or select the "Start" button to start the process. Select the "Properties..." button to start the property editor.

Device Kit: ISP Synario System

Device: GAL16V8

Here, the Boolean equation-entry format is used to create an AND function on Y (pin 18) and an XOR function on Z (pin 19). Since Z has been defined as an active low signal, however, we will actually end up with XNOR on pin 19:

```
"**** logic equations ****
equations
        Y = A & B;
        Z = A & B # !A & !B;
```

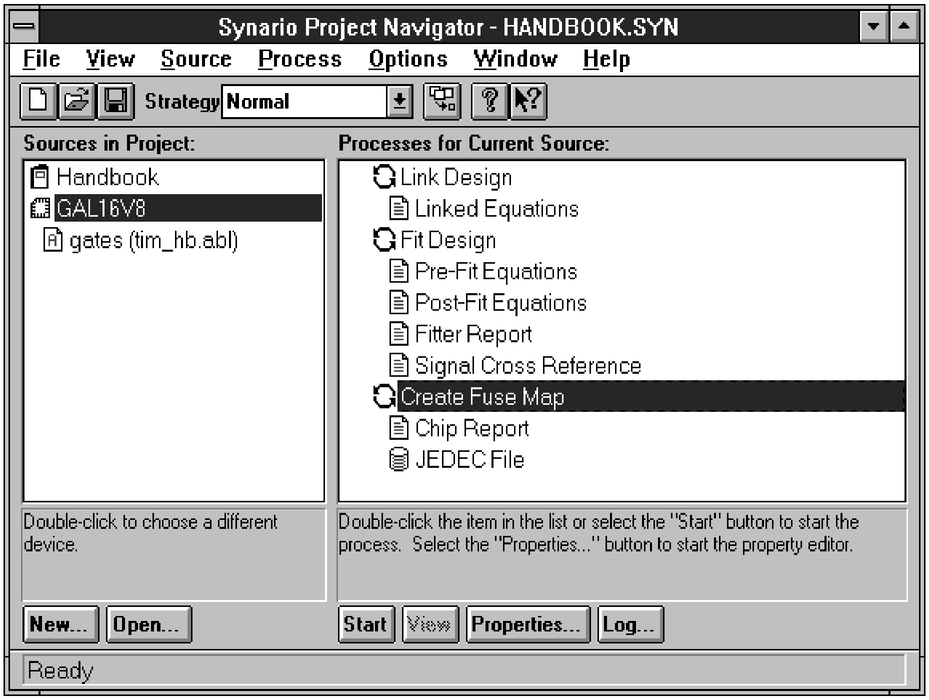
The operators used in the ABEL language are '!' for invert, '&' for the AND function and '#' for the OR function. The equations are written exactly as needed. All of the inversions for active-low inputs and outputs will be automatically resolved, a routine procedure for compiler software. Although these are simple equations, had they been complex ones that needed automatic reduction to a specific number of product terms for a given PLD, the software would have performed the reduction, as well.

When a design source file is complete, the Compile Logic Menu options compile the source file where various input file formats are converted to equation format. Next, the Reduce Logic Menu option minimizes the equations.

The last step of the process is to generate the JEDEC fusemap under the Create Fuse Map Menu. JEDEC, a standards organization with representatives from major semiconductor companies on its committees, has approved a standard for the interchange of PLD data. The JEDEC file is used as the medium of transfer from the development computer environment to that of the hardware device programmer. Included in the file are control bits that determine the status of security cells or fuses, test vectors, and data-transmission checksums. (The JEDEC standard is available from Lattice Semiconductor upon request.) A portion of the JEDEC file for our example is reproduced here:

```
QP20* QF2194* QV8* F0*
 X0*
NOTE Table of pin names and numbers*
NOTE PINS A:1 B:2 Y:18 Z:19*
L0000 10011111111111111111111111111111*
L0032 01101111111111111111111111111111*
L0256 10011111111111111111111111111111*
L2048 01000000*
L2128
1111111111111111111111111111111111111111111111111111111111111111*
L2192 1*
C13DA*
```

Figure 3. Linking and Fuse Map Creation in Synario



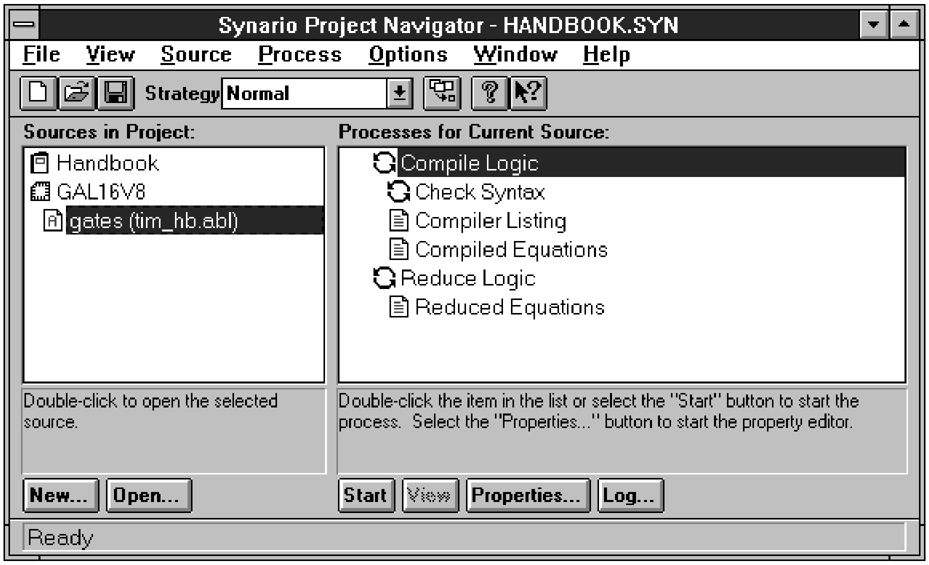
Strategy Normal

Double-click to choose a different device.

Double-click the item in the list or select the "Start" button to start the process. Select the "Properties..." button to start the property editor.

Ready

Figure 4. Compilation and Logic Optimization in Synario



Strategy: Normal

Double-click to open the selected source.

Double-click the item in the list or select the "Start" button to start the process. Select the "Properties..." button to start the property editor.

Ready

Test vectors, which indicate the stimulus and response for a PLD, serve primarily to validate the functionality of a design source file. The ABEL compiler simulates the source file so that only properly functioning patterns are programmed into a PLD for system debug. In our basic gates example, the simulation routine provides the expected data:

```
"**** test vector definition ****
test_vectors
([A,B] -> [Y,Z]);
 [0,0] -> [L,X]; "**** test AND gate ****
 [0,1] -> [L,X];
 [1,0] -> [L,X];
 [1,1] -> [H,X];
 [0,0] -> [X,H]; "**** test XNOR gate ****
 [0,1] -> [X,L];
 [1,0] -> [X,L];
 [1,1] -> [X,H];
end
```

While some PLD manufacturers claim that test vectors are also necessary for verifying functionality of the integrated circuit after programming, Lattice E^{2}CMOS GAL devices are fully tested and guaranteed to yield 100% all of the time.

Once the JEDEC fusemap is generated, the design is ready to be programmed into a device. Synario/ABEL also generates a documentation file (.REP) for the design which can be viewed under the Chip Report Menu option. The purpose of the file is to provide a hard-copy documentation of the final (reduced) equations, the cell map or 'fuse plot,' and a chip-pinout diagram, if desired (see Listing 1).

Example: Two-Story Elevator Controller

This example is designed to step the reader through the process of creating and implementing a logic design using GAL devices. Whether a novice or intermediate user of PLDs, you are encouraged to familiarize yourself with this section and the GAL application notes, which provide examples of how to implement basic functions such as decoders, shifters, multiplexers, counters, etc.

Here we will be building a two-story elevator control unit. The function of the unit is to monitor the state of the call buttons, respond to calls for service, and display the status of the elevator by means of floor and direction displays. The operating control function requires a small state machine and a latch function, while the display logic uses only combinational circuits.

Our elevator travels between two floors. Arriving at a floor in response to a call for service, the elevator opens its doors, pauses, then closes them automatically. If the up or down button is pushed, the elevator travels to the other floor. A microswitch mounted on the car informs the controller that the elevator has arrived at a new floor.

Once the elevator arrives at a floor to discharge passengers, it opens its doors, pauses to let the passengers out, then closes the doors and assumes its wait position. A

Listing 1. Synario/ABEL Documentation File

```
SYNARIO  -  Device Utilization Chart

Tutorial Using a GAL16V8
        Lattice Semiconductor
--------------------------------------------------------------------------------
Module                   : 'gates'
--------------------------------------------------------------------------------
Input files:
    ABEL PLA file        : gates.tt3
    Vector file          : gates.tmv
    Device library       : P16V8AS.dev

Output files:
    Report file          : gates.doc
    Programmer load file : gates.jed

P16V8AS Programmed Logic:
--------------------------------------------------------------------------------
Y     =  (  A & !B );
Z     = !(  A & !B
        #   !A & B );

P16V8AS Chip Diagram:
--------------------------------------------------------------------------------

                                   P16V8AS

                         +---------\       /---------+
                         |          \     /          |
                         |           -----           |
                       A |  1                   20   | Vcc
                         |                           |
                       B |  2                   19   | !Z
                         |                           |
                         |  3                   18   | !Y
                         |                           |
                         |  4                   17   |
                         |                           |
                         |  5                   16   |
                         |                           |
                         |  6                   15   |
                         |                           |
                         |  7                   14   |
                         |                           |
                         |  8                   13   |
                         |                           |
                         |  9                   12   |
                         |                           |
                     GND | 10                   11   |
                         |                           |
                         |                           |
                         `---------------------------'
                        SIGNATURE: N/A
```

Listing 1. Synario/ABEL Documentation File, Continued

```
P16V8AS Resource Allocations:
--------------------------------------------------------------------------------
        Device        | Resource  |   Design    |    Part     |
       Resources      | Available | Requirement | Utilization | Unused
======================|===========|=============|=============|==============
                      |           |             |             |
Dedicated input pins  |    10     |      2      |      2      |      8 ( 80 %)
Combinatorial inputs  |    10     |      2      |      2      |      8 ( 80 %)
Registered inputs     |     -     |      0      |      -      |      -
                      |           |             |             |
Dedicated output pins |     2     |      2      |      0      |      2 (100 %)
Bidirectional pins    |     6     |      0      |      2      |      4 ( 66 %)
Combinatorial outputs |     8     |      2      |      2      |      6 ( 75 %)
Registered outputs    |     -     |      0      |      -      |      -
Two-input XOR         |     -     |      0      |      -      |      -
                      |           |             |             |
Buried nodes          |     -     |      0      |      -      |      -
Buried registers      |     -     |      0      |      -      |      -
Buried combinatorials |     -     |      0      |      -      |      -


P16V8AS Product Terms Distribution:
--------------------------------------------------------------------------------
            Signal             |    Pin   | Terms | Terms | Terms
             Name              | Assigned | Used  |  Max  | Unused
===============================|==========|=======|=======|=======
Y                              |   18     |   1   |   8   |   7
Z                              |   19     |   2   |   8   |   6


     ==== List of Inputs/Feedbacks ====

Signal Name                    | Pin      | Pin Type
============================== |==========|=========
A                              |    1     | INPUT
B                              |    2     | INPUT


P16V8AS Unused Resources:
--------------------------------------------------------------------------------
 Pin   |  Pin   |   Product    | Flip-flop
Number |  Type  |    Terms     |   Type
=======|========|=============|==========
    3  |  INPUT |      -      |    -
    4  |  INPUT |      -      |    -
    5  |  INPUT |      -      |    -
    6  |  INPUT |      -      |    -
    7  |  INPUT |      -      |    -
    8  |  INPUT |      -      |    -
    9  |  INPUT |      -      |    -
   11  |  INPUT |      -      |    -
   12  |  BIDIR | NORMAL  8   |    -
   13  |  BIDIR | NORMAL  8   |    -
   14  |  BIDIR | NORMAL  8   |    -
   15  | OUTPUT | NORMAL  8   |    -
   16  | OUTPUT | NORMAL  8   |    -
   17  |  BIDIR | NORMAL  8   |    -
```

call for service at the floor where the elevator is resting will result in the doors being opened.

A free-running clock controls the elevator's operation, toggling every 5 to 10 seconds to allow a brief pause during each arrival and departure activity. While this slow clock rate is appropriate for the timing of the elevator doors and car movement, it is far too slow to capture a time-independent call for service. As such, a latch function that captures data instantly (actually, within 25 ns for the GAL16V8B-25) is designed using two of the GAL device macrocells.

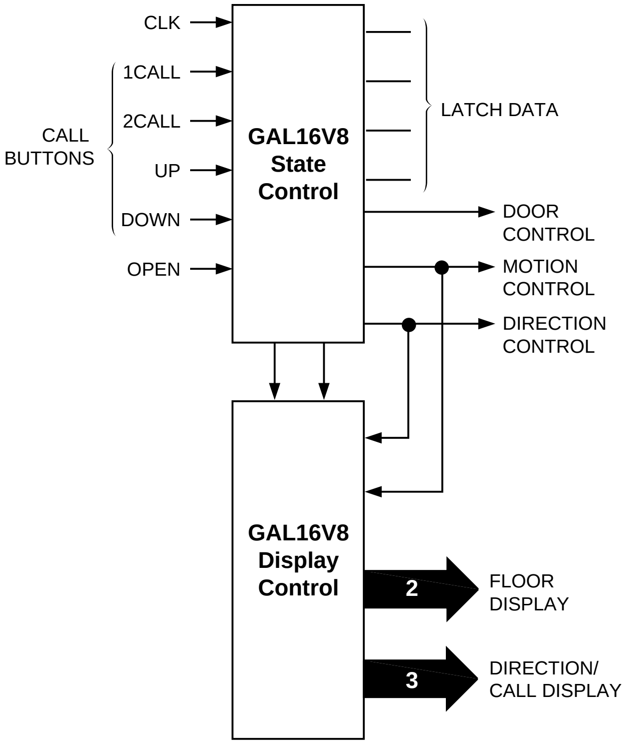
As shown in Figure 5, the total elevator control unit uses two GAL16V8s — one to perform the actual control function, the other to handle the display.

The control of the elevator consists of two basic functions: the call-button latches and the state machine. The latches, constructed from the GAL device's available AND and OR gates (instead of using the on-chip D-type register), are instantaneous and not dependent on a clock for holding data. The truth table of the S-R latch used is shown in Table 1. As shown in the truth table, the latch is set by applying a logic 1 to SET, and reset by applying a logic 1 to RESET. Applying a logic 0 to both inputs causes a hold state, while applying a logic 1 to both is undefined for this type of latch. The various call signals (UCALL, DCALE, OCALL, 1CALL, 2CALL) are applied to the two latches to command the elevator to travel to the requested floor.

Table 1. SR Latch Truth Table

Set	Reset	Output	$\overline{\text{Output}}$
0	0	Hold	Hold
0	1	0	1
1	0	1	0
1	1	Not Valid	

Figure 5. Block Diagram



The first step in this GAL implementation is to translate the functional operation of the elevator (described in the text in the preceding paragraphs) to a logical format. This is realized through the use of a state-transition diagram, which literally describes all the allowed stable states (on floor two, doors open, etc.) that our elevator can be in. An unacceptable state, for example, would be resting between floors.

Figure 6 shows the state-transition diagram. Inside each state circle, the diagram indicates the state name (top half) and the condition of each of the state variables: DOOR, MOTION, and DIRECTION. The transitions out of the state are shown with the logic level requirements to make the transition. Also shown is the destination of each of the transfers.

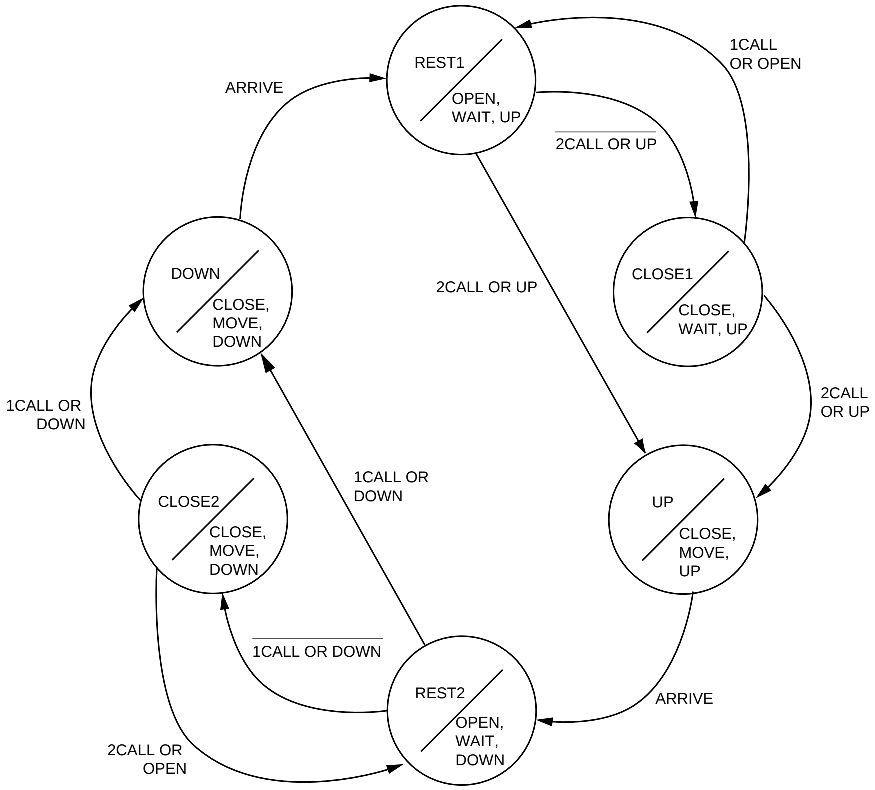
The latched signals L1CALL and L2CALL are used to start the states changing. The ARRIVAL input tells the car when to stop its motion. The normal wait state of the elevator is either CLOSE1 or CLOSE2 (not moving with door closed).

The information is then transferred from the transition diagram to the ABEL state-machine syntax, shown in Listing 2. Notice the use of defaults in the state syntax to indicate what state should be selected (or held) if none of the criteria for exit is met. There is also an identifiable 1-to-1 correspondence from the state transition diagram to the state-machine syntax. The portion of the documentation file which includes reduced equations is shown in Listing 3. Notice that the compiler automatically chose the proper polarity to fit the reduced equations into the GAL device, using DeMorgan's Law: pins 12, 13, and 14 are inverted, relative to the other output pins.

Display Design

The source file for the up/down arrow display is shown in Listing 4. The UPARROW is active only when the car is moving up. DNARROW is true only when the car is

Figure 6. State-Transition Diagram



moving down. The common bar, SEGARROW, is active during any call, in any direction. This signal is also active when an unserviced call is active. As such, the SEGARROW signal is a call waiting indicator that acknowledges a call button being pushed. The logic equations for the arrow functions are self-explanatory; its input signals come from the controller.

The floor indicator is a simplified decoder. A truth table input format is used for the design. Notice that a floor is always indicated, and that the change occurs when the direction bit changes. This bit is constrained by the state machine to change only when the car arrives at a floor. The documentation file showing the reduced equations is reproduced in Listing 5.

Listing 2. Design Input File for Control Section

```
module elev_ctl
Title 'Two Story Elevator Control Logic Example Using a GAL16V8
        Lattice Semiconductor
        ABEL Source File'

"**** inputs ****
CLK,!OE                 PIN 1,11;
CALL1,CALL2             PIN 2,3;        "Buttons on Floors
" Up ,Down ,Open
UCALL,DCALL,OCALL       PIN 4,5,6;      "Buttons in Elevator
ARRIVE                  PIN 7;          "Floor Arrival Sensor

"**** outputs ****
DOOR            PIN 12 ISTYPE 'REG_D,INVERT'; "0=Open, 1=Close
MOTION          PIN 13 ISTYPE 'REG_D,INVERT'; "0=Wait, 1=Move
DIRECTION       PIN 14 ISTYPE 'REG_D,INVERT'; "0=Up  , 1=Down

L1CALL,L1CALL_  PIN 16,17 ISTYPE 'COM,INVERT'; "Latched call to 1st floor
L2CALL,L2CALL_  PIN 18,19 ISTYPE 'COM,INVERT'; "Latched call to 2nd floor

"**** state definitions ****
CONTROL = [DOOR,MOTION,DIRECTION];

REST1   =       ^B000;
CLOSE1  =       ^B100;
UP      =       ^B110;
REST2   =       ^B001;
CLOSE2  =       ^B101;
DOWN    =       ^B111;

"**** intermediate definitions ****
C,X,H,L=.C.,.X.,1,0;

FLOOR1 =  DIRECTION.FB;
FLOOR2 = !DIRECTION.FB;

"**** logic equations ****
equations

L1CALL  = !(  L1CALL_ #  CALL1 #  (FLOOR1 &  OCALL)
                               #  (FLOOR2 &  DCALL));
L1CALL_ = !(  L1CALL  #  (!DOOR.FB & !MOTION.FB & !DIRECTION.FB));
L2CALL  = !(  L2CALL_ #  CALL2 #  (FLOOR2 &  OCALL)
                               #  (FLOOR1 &  UCALL));
L2CALL_ = !(  L2CALL  #  (!DOOR.FB & !MOTION.FB &  DIRECTION.FB));

CONTROL.CLK = CLK;

"**** state machine definition ****
state_diagram CONTROL
state REST1: if (L2CALL) then UP;
                else CLOSE1;
state CLOSE1: if (L2CALL) then UP;
              else if (L1CALL) then REST1;
              else CLOSE1;
state UP: if (ARRIVE) then REST2;
          else UP;
state REST2: if (L1CALL) then DOWN;
             else CLOSE2;
state CLOSE2: if (L1CALL) then DOWN;
              else if (L2CALL) then REST2;
              else CLOSE2;
state DOWN: if (ARRIVE) then REST1;
            else DOWN;

end
```

Listing 3. Expanded Product Terms for Control Section

```
L1CALL     = (  !DIRECTION.Q & !L1CALL_ & !CALL1 & !OCALL
              #   DIRECTION.Q & !L1CALL_ & !CALL1 & !DCALL );

L1CALL_    = !(  DIRECTION.Q & DOOR.Q & MOTION.Q
               #   L1CALL );

L2CALL     = (  DIRECTION.Q & !OCALL & !L2CALL_ & !CALL2
              #   !DIRECTION.Q & !L2CALL_ & !CALL2 & !UCALL );

L2CALL_    = !(  !DIRECTION.Q & DOOR.Q & MOTION.Q
               #   L2CALL );

DOOR.D   = (  !DIRECTION.Q & !DOOR.Q & MOTION.Q & !L1CALL & L2CALL
            #   DIRECTION.Q & !DOOR.Q & MOTION.Q & L1CALL & !L2CALL
            #   !DOOR.Q & !MOTION.Q & ARRIVE ); " ISTYPE 'INVERT'
DOOR.C   = (  CLK );

MOTION.D   = (  !DIRECTION.Q & MOTION.Q & !L1CALL
              #   DIRECTION.Q & MOTION.Q & !L2CALL
              #   !DOOR.Q & !MOTION.Q & ARRIVE ); " ISTYPE 'INVERT'
MOTION.C   = (  CLK );

DIRECTION.D   = (  DIRECTION.Q & MOTION.Q
                 #   !DIRECTION.Q & !DOOR.Q & !MOTION.Q & ARRIVE
                 #   DIRECTION.Q & !DOOR.Q & !ARRIVE ); " ISTYPE 'INVERT'
DIRECTION.C   = (  CLK );
```

Listing 4. Design Input File for Display Section

```
module elev_dsp
Title ‘Two Story Elevator Display Logic Example Using a GAL16V8
        Lattice Semiconductor
        ABEL Source File’


“**** inputs ****
L1CALL,L2CALL           PIN 2,3;        “Call Status
MOTION,DIRECTION        PIN 4,5;        “Control Status


“**** outputs ****                          arrow display diagram

UPARROW         PIN 12 ISTYPE ‘COM,INVERT’; “       ^
SEGARROW        PIN 13 ISTYPE ‘COM,INVERT’; “       |
DNARROW         PIN 14 ISTYPE ‘COM,INVERT’; “       v

SEG1,SEG2,SEG12 PIN 15,16,17 ISTYPE ‘COM,INVERT’; “LED Segment Drivers
“       segment display diagram
“                2
“               —-
“                  | 12
“                2 |
“               —-
“            2 |   | 1
“              |   |
“               —-
“                2
```

Listing 4. Design Input File for Display Section, Continued

```
“**** intermediate definitions ****
C,X,H,L=.C.,.X.,1,0;

“**** logic equations ****
equations

UPARROW = MOTION & DIRECTION;
SEGARROW= L1CALL # L2CALL;
DNARROW = MOTION & !DIRECTION;

“**** truth table definition ****
truth_table
([DIRECTION, MOTION] -> [SEG1, SEG2, SEG12]);
 [      0  ,  0    ] -> [  1 ,   0 ,   1  ] ;
 [      0  ,  1    ] -> [  1 ,   0 ,   1  ] ;
 [      1  ,  0    ] -> [  0 ,   1 ,   1  ] ;
 [      1  ,  1    ] -> [  0 ,   1 ,   1  ] ;


end
```

Listing 5. Expanded Product Terms for Display Section

```
UPARROW  =  ( MOTION & DIRECTION );
SEGARROW = !(!L1CALL & !L2CALL );
DNARROW  =  ( MOTION & !DIRECTION );
SEG1     = !( DIRECTION );
SEG2     = !(!DIRECTION );
SEG12    = !(0);
```

E^2CMOS, GAL, ispGAL, ispLSI, pLSI, pDS, Silicon Forest, UltraMOS, Lattice Logo, L with Lattice Semiconductor Corp. and L (Stylized) are registered trademarks of Lattice Semiconductor Corporation (LSC). The LSC Logo, Generic Array Logic, In-System Programmability, In-System Programmable, ISP, ispATE, ispCODE, ispDOWNLOAD, ispGDS, ispStarter, ispSTREAM, ispTEST, ispTURBO, Latch-Lock, pDS+, RFT, Total ISP and Twin GLB are trademarks of Lattice Semiconductor Corporation. ISP is a service mark of Lattice Semiconductor Corporation. All brand names or product names mentioned are trademarks or registered trademarks of their respective holders.

Lattice Semiconductor Corporation (LSC) products are made under one or more of the following U.S. and international patents: 4,761,768 US, 4,766,569 US, 4,833,646 US, 4,852,044 US, 4,855,954 US, 4,879,688 US, 4,887,239 US, 4,896,296 US, 5,130,574 US, 5,138,198 US, 5,162,679 US, 5,191,243 US, 5,204,556 US, 5,231,315 US, 5,231,316 US, 5,237,218 US, 5,245,226 US, 5,251,169 US, 5,272,666 US, 5,281,906 US, 5,295,095 US, 5,329,179 US, 5,331,590 US, 5,336,951 US, 5,353,246 US, 5,357,156 US, 5,359,573 US, 5,394,033 US, 5,394,037 US, 5,404,055 US, 5,418,390 US, 5,493,205 US, 0194091 EP, 0196771B1 EP, 0267271 EP, 0196771 UK, 0194091 GB, 0196771 WG, P3686070.0-08 WG. LSC does not represent that products described herein are free from patent infringement or from any third-party right.

The specifications and information herein are subject to change without notice. Lattice Semiconductor Corporation (LSC) reserves the right to discontinue any product or service without notice and assumes no obligation to correct any errors contained herein or to advise any user of this document of any correction if such be made. LSC recommends its customers obtain the latest version of the relevant information to establish, before ordering, that the information being relied upon is current.

LSC warrants performance of its products to current and applicable specifications in accordance with LSC's standard warranty. Testing and other quality control procedures are performed to the extent LSC deems necessary. Specific testing of all parameters of each product is not necessarily performed, unless mandated by government requirements.

LSC assumes no liability for applications assistance, customer's product design, software performance, or infringements of patents or services arising from the use of the products and services described herein.

LSC products are not authorized for use in life-support applications, devices or systems. Inclusion of LSC products in such applications is prohibited.

LATTICE SEMICONDUCTOR CORPORATION
5555 Northeast Moore Court
Hillsboro, Oregon 97124 U.S.A.
Tel.: (503) 681-0118
FAX: (503) 681-3037
http://www.latticesemi.com

November 1996