CHAPTER 8

Programmable Logic and Storage Devices

As technology advances, the density, complexity and size of field-programmable gate arrays (FPGAs) provide an attractive, cost-efficient, and increasingly important alternative to semicustom application-specific integrated circuits (ASICs). Mask charges for cell-based ASICs can cost from $250K to $500K, eliminating these devices from the low-volume end of the market. The opportunity to realize large circuits in FPGAs has created pressure for a change in the method by which circuits are designed for FPGA-based applications. Designers who use schematic-entry tools can be productive and efficient when designs are small, but the trend is toward larger and larger designs targeted for FPGAs. The language-based design methodology that has served the ASIC design flow has become essential to FPGA-based design flows, because it is the key to meeting ever-shrinking windows of opportunity for new products. As a result, FPGA vendors have been led to improve their support of language-entry tools for FPGAs, and recognize that more and more designers are shifting from schematic entry tools to language-entry tools. This chapter will emphasize a design flow for FPGAs that is entirely Verilog-based.

The technologies available for implementing digital circuits range from the discrete gates and standard integrated circuits (ICs) used in low-density/low-performance applications, to cell-based and full-custom ICs for high-density/high-performance circuits. Standard integrated circuits can be manufactured cheaply, but they implement very limited, basic functionality at low levels of integration. Production of customized logic, having a small market, creates an inventory risk because the quantities that could be sold do not warrant the expense of their development and production, and IC manufacturers cannot afford to stock multiple variants of specialized functional units, as they do with standard parts. Inevitable progress in technology alone would render their
inventory worthless before their investment could be recovered. Our focus in this chapter will be on programmable logic devices, which lie between the two extremes of density and performance that characterize standard parts and full-custom circuits.

Programmable logic devices (PLDs) were born out of a necessity created by two conflicting realities: large, dense, high-performance circuits cannot be built practically or economically from discrete devices, and dedicated ICs cannot be produced economically to satisfy a diversity of low-volume applications. The resolution of these forces lies in programmable logic devices.

Although read-only memories (ROMs), programmable logic arrays (PLAs), programmable array logic (PALs), complex PLDs (CPLDs), FPGAs, and mask-programmable gate arrays (MPGAs) are all programmable, we will use the term PLD to indicate the low-density structures that were introduced to implement two-level combinational logic: PLAs, PALs, and similar vendor-named devices. PLDs are distinguished by their having a regular structure of identical basic functional units with fixed architecture. Certainly, MPGAs and standard-cell-based designs can be considered to be programmable because an application-specific program determines the metalization layers interconnecting the underlying transistors to form functional units in the case of an FPGA, and the placement and routing of standard cells can be considered to “program” a cell-based wafer. We consider MPGAs to be in the broader family of programmable devices, but we do not include standard-cell-based designs. Why?

MPGAs are formed from a regular array of transistors. Unprogrammed MPGAs have an identical structure. They are programmed by adding layers of metal interconnects to compose and connect macros with a desired functionality. On a local basis, the interconnect might establish, for example, the connectivity that forms a NAND gate, and on a global basis establish the functionality of an adder. The architecture of the basic functional units remains unchanged, but the interconnection fabric is unique to the application. In contrast, standard cell layouts do not have a fixed, basic architecture of functional units. A standard-cell-based design has regularity in the structure of its layout channels, but the functional units themselves are not uniform and do not have an architecturally determined placement. One cell may implement an inverter, another a flip-flop. Nor is a cell itself programmed. The overall architecture of a cell-based design is completely flexible within the constraints imposed by layout routing channels and a library of cells. No cell pattern need be replicated in a cell-based layout. For these reasons, we make a distinction between programming that overlays an interconnection fabric on a given fixed architecture, and programming that establishes an architecture of functional units, as is the case with standard cells. We mean the former case when we use the term PLD.

Storage devices, such as ROMs, are considered to be PLDs because they can implement combinational logic by storing the values of a function at memory locations that are addressed by the inputs of the function. These implementations, of necessity, implement the full truth table of the function. Memory-implmented combinational logic may be inefficient, because minimization techniques are not used to implement a full truth table of a function, and device resources might not be fully utilized.
8.1 Programmable Logic Devices

Programmable logic devices\(^1\) have a fixed architecture but their functionality is programmed for a specific application, either by the manufacturer or by the end user. PLDs whose architecture is programmed by the manufacturer are referred to as mask-programmable logic devices (MPLDs); those that are programmed by the end user are referred to as field-programmable logic devices (FPLDs). The architecture of the basic functional unit of a PLD is fixed, and is not customized by the user. Consequently, the development and production costs of PLDs can be amortized over a larger base of customers, and the range of applications for the devices can be very broad. This reduces production and inventory risks for the manufacturer and unit costs for the consumer, while at the same time allowing advances in processing technology to be incorporated into an evolving product line. The design cycle of a system that uses a PLD can be very short because PLDs can be manufactured, tested, and placed in inventory in advance of their being chosen as a technology for an application. Because the devices are premanufactured, they are suitable for rapid prototyping of a design.

Three basic characteristics distinguish PLDs from each other: (1) an architecture of identical basic functional units, (2) a programmable interconnection fabric, and (3) a programming technology. The first type of PLD that we will consider has the AND-OR plane structure shown in Figure 8-1. This type of architecture is used to implement ROMs, PLAs, and PALs. It implements Boolean expressions in SOP form: The AND plane forms product terms selectively from the inputs, and the OR plane forms outputs from sums of selected product terms. A programmable interconnect fabric joins the two planes, so that the outputs implement sum-of-product expressions of the inputs.

Whether and how a plane can be programmed determines the particular type of PLD that is implemented by the overall structure.

8.2 Storage Devices

The architecture used to implement PLDs lends itself to implementation of storage devices. Storage devices can be read-only or random-access, depending on whether the contents of a memory cell can be written during normal operation of the device. Read-only memory (ROM) is a device programmed to hold certain contents, which remain unchanged during operation and after power is removed from the device. In contrast, the contents of a random-access memory (RAM) can be changed during operation, and they vanish when power is removed. There is another major distinction between ROMs and RAMs: The circuit of a ROM is structurally modified to program the device prior to its use. In contrast, the circuit of a RAM is not programmed—it is fixed. Only the contents of a RAM are programmed. This occurs dynamically, during normal read and write operations of the circuit.

\(^1\)There are many more vendors and families of devices than we can cover in a book of this type. See www.e-insite.net/ednmag for EDN Access's annual directory of PLDs, CPLDs, and FPGAs.
8.2.1 Read-Only Memory (ROM)

A $2^n \times m$ ROM consists of an addressable array of semiconductor memory cells organized as $2^n$ words of $m$ bits each. A read-only memory has $n$ inputs, called "address lines," and $m$ outputs, called "bit lines." The AND-plane of the structure shown in Figure 8-2 serves as an address decoder and is nonprogrammable. The address decoder implements a full decode of the $n$ inputs, and each pattern of input bits addresses a unique decoded output, called a "word line." Each input address word selects one of the $2^n$ memory words to assert a word line, and each cell of a word stores 1 bit of information. Consequently, each word line corresponds to a minterm of a Boolean expression.

ROMs can be manufactured in a variety of technologies: bipolar, complementary metal-oxide semiconductor (CMOS), n-channel MOS (nMOS), and p-channel MOS (pMOS). A mask-programmed ROM implemented in nMOS technology has the circuit structure shown in Figure 8-3. The bit lines form the output word, and $n$-channel
link transistors connect the word lines to the bit lines. A bit line is normally pulled up to $V_{DD}$, but when a word line is pulled high by the address decoder, the n-channel transistors that are attached to it will be turned on. This action pulls the corresponding bit lines down. The pattern of link transistors attached to a given word line determines the pattern of 1s and 0s that appear on the bit lines for the applied input address word. This pattern is determined by the customized mask set for the device. Given the three-state output inverters, the presence of a link transistor corresponds to a stored 1 at the location of the memory cell. The mask set ensures that transistors will be fabricated only at locations that require a link.

The information stored in a ROM can be read under normal operation of a host circuit, but not written. The outputs of a ROM are normally three-stated, so that the device can be connected to a shared bus serving multiple devices. In commercial ROMs, an additional chip-select input allows multiple devices to be connected to a
common bus, each selectable by its unique address. If a ROM has been selected, a pattern of 0s and 1s at its address inputs causes one and only one word line to be asserted.

A $2^n \times m$ ROM can store $m$ different functions of $n$ variables (i.e., truth table storage). Figure 8-4 illustrates a $16 \times 8$ ROM having a 4-bit address word and a total of 16 memory words of 8 bits each. Commercial ROMs are available in a range of organization and densities, as shown in Table 8-1.

A ROM is a nonvolatile memory because the stored information remains when power is removed from the device. Mask-programmable ROMs are manufactured with a fixed, nonerasable memory pattern, usually for high-volume applications. Their non-recurring engineering (NRE) cost is relatively high compared to a field-programmable ROM because the mask set that programs the chip is customized to a particular end user’s application. The mask set can be produced in about a 4-week cycle. Mask-programmed ROMs are used in applications in which a system needs stored data and has no need to alter the data in ordinary use. For example, they are used as data tables that hold the codes for characters that are to be displayed on a cathode ray tube (CRT) monitor screen of a computer system, and hold the bootstrap program that executes immediately when a personal computer is powered on. They are widely used in electronic point-of-sale terminals in retail stores, instrumentation, domestic appliances, industrial equipment, video games, and security systems.

![Schematic symbol for a $16 \times 8$ ROM.]

**FIGURE 8-4** Schematic symbol for a $16 \times 8$ ROM.

| Table 8-1 Organization and density of commercial ROMs |
|---------------------------------|--------|
| **Organization**              | **Density** |
| $32K \times 8$                | 256K bit |
| $64K \times 8$                | 512K bit |
| $128K \times 8$               | 1M bit   |
| $256K \times 8$               | 2M bit   |
| $512K \times 8$               | 4M bit   |
| $1024K \times 8$              | 8M bit   |
| $64K \times 16$               | 1M bit   |
| ...                           |         |
| $256K \times 16$              | 4M bit   |
| $512K \times 16$              | 8M bit   |
8.2.2 Programmable ROM (PROM)

A field-programmable ROM (PROM) is one that can be programmed (once) by an end user with a special apparatus called a PROM programmer. PROMs are said to be one-time programmable (OTP) or write-once memory (WOM). PROMs are nonvolatile and nonerasable. Usually manufactured in a bipolar technology, a PROM initially has a pull-up device at every crosspoint between a word line and the internal bit line. The pull-up device (diode or transistor) is also connected to a metal fusible link, as shown in Figure 8-5. A PROM programmer selectively applies a voltage (10-30 V) to cause current sufficient to vaporize the link, thereby disconnecting the pull-down device from the word line, and permanently causing a 1 to appear in that cell when it is decoded by a word line. The output of the bit line is the inverted content of the memory cell.

The bit line outputs of a PROM are driven by three-state inverters, and each inverter input is connected to ground by a pull-down resistor and is also connected to the internal bit line. In the absence of a signal on a word line, the bit line will be at ground potential, and the output will be high. The enable line in the circuit of Figure 8-5 is active-low, and a high minterm line pulls a bit line output low. Cells having a blown link are not affected by their minterm line, and their output remains at 1, due to the action of the pull-down resistor. Note that a bit-line can be pulled down by one or more

![Figure 8-5 Circuit structure for a fusible-link bipolar PROM.](image-url)
cells. In this scheme, the presence of a link transistor implies a 0 in the output word when the wordline is decoded. The programming is permanent (i.e., there is no way to restore the blown links and create a different program). A program can be modified however, if the modification affects only links that have not yet been blown.

### 8.2.3 Erasable ROMs

The architecture of an erasable PROM is similar to that of a PROM, but it uses a floating-gate nMOS transistor as the link device between a word line and the bit lines (Figure 8-6). A floating-gate transistor has an additional gate inserted between the operational gate and the channel. This gate is surrounded by a high-impedance dielectric material, and is insulated from the operational gate. When special circuitry (not shown) applies a sufficiently high voltage (e.g., 21 V) to the operational gate the insulator breaks down and a negative charge is pulled from the channel and becomes trapped on the floating gate when the programming voltage is removed. The effect of the trapped charge is to turn off the transistor by depleting the channel of carriers, which effectively raises the threshold voltage of the transistor and breaks the link between the word line and the bit line, allowing the bit line to float high and remain high independently of the

---

**FIGURE 8-6** Circuit structure for a floating-gate EEPROM.
affected word line. Subsequent reads of the cell are 1. In this scheme, the presence of a programmed link transistor (i.e., one that has trapped charge) implies a 0 in the output word when the word line is decoded.

There are two types of erasable ROMs: those that are erasable by ultraviolet (UV) light and those that are erasable by electricity. The former, called an EPROM or UV-EPRROM, have a quartz opening and rely on a mechanism by which the UV light at a specific wavelength causes a temporary breakdown of the insulation of the floating gate and allows photocurrents to remove the trapped charge and effectively erase the stored information. The latter type, called an EEPROM (electrically erasable PROM), use electrical pulses to break down the insulated floating gate and erase the stored pattern. Application of a high negative voltage to a minterm line will remove the trapped charge from the floating gate. The UV-erase mechanism of an EPROM is nonselective (also referred to as “bulk erase”); all of the memory contents are reprogrammed to 1. A EEPROM, however, has additional circuitry providing a selective erase capability, allowing individual words to be selectively erased and reprogrammed.

EPROMs are commonly used in the debug phase of firmware development for microprocessor-based systems. They require 5 to 20 minutes of exposure to UV light to accomplish an erase. ROMs are substituted for production after the program is correct because their packages omit the quartz window and are cheaper. EEPROMs are attractive because they can be programmed in-circuit, can be erased with low current, and do not require the additional hardware and expense of a PROM programmer or a UV source.

Volatility and fatigue are two important considerations in applications of ROM technology. In the absence of UV light, an EPROM is guaranteed to hold 70% of its charge for at least 10 years [1]. The insulating material in an EEPROM is thinner than that for an EPROM, and can deteriorate, so EEPROMs have a limited number of write/erase cycles, typically $10^2$ to $10^5$. EEPROMs that have exceeded their fatigue limit may fail to hold a charge on the floating gate or may trap charge on the gate. Because they can be erased electrically they erases much faster than an ordinary EPROM, making them suitable for prototype code development. They are also used in system applications that do not require a high number of write/erase cycles over the useful life of a product, such as storage of default configuration data in a personal computer [1]. EEPROMs are also available with low in-circuit programming voltages (e.g., Atmel AT49LV1024).²

8.2.4 ROM-Based Implementation of Combinational Logic

ROMs are commonly used in applications that require a truth table for combinational logic. They are an attractive technology because a ROM can be programmed to implement any of $2^n$ different functions of $n$ inputs, and a single ROM can implement any of those functions at any of its bit lines (standard logic would require a new circuit structure for each different function). A ROM-based design can be modified by simply replacing the ROM, without altering the external circuitry. The complexity of the logic

being implemented does not have an impact on the effort to program the device, as it would in the case of discrete or building-block logic. ROMs are usually faster than multiple large and medium scale integrated (LSI/MSI) devices and other PLDs in moderately sized circuit applications, and often they are faster than an FPGA or custom LSI chip in a comparable technology. On the other hand, for moderately complex functions, a ROM-based circuit is usually more expensive, consumes more power, and may run more slowly than a circuit that uses multiple LSI/MSI devices and PLDs or a small FPGA [1]. Their full address decoding circuitry ultimately limits ROMS to applications that have no more than 20 inputs. Like other semiconductor devices, ROMs benefit from advances in technology that are leading to cheaper and denser devices.

8.2.5 Verilog System Tasks for ROMs

Verilog has two file input-output (I/O) system tasks that can be used to load memory data from a text file, reducing the effort required to initialize a large memory, as an alternative to writing the individual words within the ROM model. A single ROM model can serve a variety of applications by substituting text files. The tasks $readmem and $readmemh load to specified locations in a memory the contents of a text file formatted as binary or hexadecimal words, respectively (see “Selected System Tasks and Functions” at the companion web site).

Example 8.1

The truth table of the 2-bit comparator presented in Example 4.4 is shown in Figure 8-7 with a symbolic diagram of the fuse links required to program a ROM-based implementation of the circuit with active-low enabled, three-stated outputs. Note that the output column $D0$ is unused, and that the pattern of links accounts for the inverted outputs of the device.

The Verilog model ROM_16_x_4 illustrates how to declare a memory of 16 words, each having a width of 4 bits, and how to load the memory from a text file of data.

```verilog
module ROM_16_x_4 (ROM_data, ROM_addr);
    output [3:0] ROM_data;
    input [3:0] ROM_addr;
    reg [3:0] ROM [15:0];

    assign ROM_data = ROM [ROM_addr];

    initial $readmemh("ROM_Data_2bit_Comparator.txt", ROM, 0, 15);
endmodule
```

The contents of a binary-formatted text file for the 2-bit comparator would be listed from address 0 to address 15 as:

```
001x
010x
010x
010x
100x
001x
010x
```
**FIGURE 8-7** Truth table and PROM fuse map for a 2-bit comparator.

A useful tip: in the Silos III simulation environment for Verilog (see www.sir.ucad.com), the text file to be read by $readmemb or $readmemh is expected to be located in the same directory (folder) in which the project is located, unless a pathname is specified to a different location. The simulation results in Figure 8-8 display the contents of the ROM as a word and as individual bits, illustrating that the unused bit is displayed in Verilog's 4-valued logic as an unknown logic value (denoted by x). The actual fuse map would have to specify a 1 or 0 for the bit.

*End of Example 8.1*
8.2.6 Comparison of ROMs

A variety of ROMS are manufactured by several commercial vendors. Table 8-2 compares representative devices and lists some typical performance attributes. The indicated trend of the complexity and cost is qualified by the fact that the unit cost of mask-programmed ROMs can be quite low, depending on the volume of parts that are produced. The performance characteristics are a moving target, linked to advances in process technology.

8.2.7 ROM-Based State Machines

ROMs provide a convenient implementation of a state machine, and can be an economical implementation if the attributes of the device match the application. The ROM-based state machine shown in Figure 8-9 uses a 2^n × m ROM to store the next-state and output functions of a state machine. The state of the machine is stored in a set of D-type flip-flops, because they typically require fewer outputs from the ROM than would a J-K flip-flop.
### TABLE 8-2  Comparison of (a) ROM types and (b) performance attributes.

<table>
<thead>
<tr>
<th>Device</th>
<th>Programming Mode</th>
<th>Erase Mode</th>
<th>Complexity and Cost</th>
<th>Example</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPROM</td>
<td>In-circuit</td>
<td>In-circuit</td>
<td></td>
<td>Intel 2864</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Byte-by-byte</td>
<td>Byte-by-byte</td>
<td></td>
<td>AT91LV1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8K × 8 nMOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64K × 16 nMOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70 ns**</td>
<td></td>
</tr>
<tr>
<td>FLASH</td>
<td>In-circuit</td>
<td>In-circuit</td>
<td></td>
<td>Intel 2732</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bulk or sector</td>
<td></td>
<td>AT27BV400</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4K × 8 nMOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>45 ns</td>
<td></td>
</tr>
<tr>
<td>EPROM</td>
<td>Out-of-circuit</td>
<td>Out-of-circuit</td>
<td></td>
<td>TMS47C256</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bulk, UV Light</td>
<td></td>
<td>32K × 8 CMOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AT27BV400</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>256K × 16 or 512K × 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>150 ns</td>
<td></td>
</tr>
</tbody>
</table>

*Requires high volume to offset NRE

**Programming time: 500 ms

***One-time programmable

---

#### (b)

<table>
<thead>
<tr>
<th>Type</th>
<th>Technology</th>
<th>Read cycle</th>
<th>Write cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>NMOS, CMOS</td>
<td>10-200 ns</td>
<td>4 weeks</td>
</tr>
<tr>
<td>ROM</td>
<td>Bipolar</td>
<td>&lt; 100 ns</td>
<td>4 weeks</td>
</tr>
<tr>
<td>PROM</td>
<td>Bipolar</td>
<td>&lt; 100 ns</td>
<td>10-50 μs/byte</td>
</tr>
<tr>
<td>EPROM</td>
<td>NMOS, CMOS</td>
<td>25-200 ns</td>
<td>10-50 μs/byte</td>
</tr>
<tr>
<td>EEPROM</td>
<td>NMOS</td>
<td>50-200 ns</td>
<td>10-50 μs/byte</td>
</tr>
</tbody>
</table>


---

The method for designing a ROM-based state machine is simplified because the truth table is implemented directly, without minimization. The size of the array depends on the number of inputs, not on the complexity of the implemented logic. We form a ROM table in which the row address represents the present state of the machine, and the contents associated with that address hold the output and the next state.

---

**Example 8.2**

A Mealy-type state machine describing a binary coded decimal (BCD)-to-Excess_3 code converter was developed by manual methods in Example 3.2. A Verilog model of a ROM memory and of the machine are listed below. The ROM model is external to the state machine model. Its contents are written immediately by the initial (single-pass) behavior that executes when a simulation begins. The listing identifies the contents
stored at each ROM address. The comments identify the state associated with each address, and hold the output and next state. A continuous assignment updates the address of the ROM (ROM_addr) whenever the state or the input of the machine change, ensuring that the machine is of the Mealy type. The testbench specifies a simple input sequence for the purpose of illustrating the machine's behavior, shown in Figure 8-10,
which matches the behavior of the manually designed gate-level machine shown in Figure 3-23. In this example the contents of the ROM are listed within the ROM, rather than in an external file.

```
module ROM_BCD_to_Excess_3 (ROM_data, ROM_addr);
output [3:0] ROM_data;
input [3:0] ROM_addr;
reg [3:0] ROM [15:0];
assign ROM_data = ROM[ROM_addr];

// initial begin
initial begin
  ROM[0] = 4'b1001; // S_0 0 000 1 001
  ROM[1] = 4'b1111; // S_1 0 001 1 111
  ROM[2] = 4'b1000; // S_6 0 010 1 000
  ROM[3] = 4'b1110; // S_4 0 011 1 110
  ROM[4] = 4'bxxxx; // not used
  ROM[5] = 4'b0011; // S_2 0 101 0 011
  ROM[6] = 4'b0000; // S_5 0 110 0 000
  ROM[7] = 4'b1110; // S_3 0 111 0 110
  ROM[8] = 4'b1010; // S_0 1 000 0 101
  ROM[9] = 4'b0011; // S_1 1 001 0 011
  ROM[10] = 4'b0000; // S_6 1 010 0 000
  ROM[11] = 4'b0010; // S_4 1 011 0 010
  ROM[12] = 4'bxxxx; // not used
  ROM[13] = 4'b1011; // S_2 1 101 1 011
  ROM[14] = 4'b1000; // S_5 1 110 1 000
  ROM[15] = 4'b1110; // S_3 1 111 1 110
end
endmodule
```

```
module BCD_to_Excess_3_ROM (ROM_addr, B_out, ROM_data, B_in, clk, reset);
output [3:0] ROM_addr;
output B_out;
input [3:0] ROM_data;
input B_in, clk, reset;
reg [2:0] state;
wire [2:0] next_state;
wire B_out;
assign next_state = ROM_data [2:0];
assign B_out = ROM_data [3];
assign ROM_addr = {B_in, state};

always @(posedge clk or negedge reset)
  if (reset == 0) state <= 0; else state <= next_state;
endmodule
```
module test_BCD_to_Excess_3b_Converter();
wire B_out, clk;
wire [3:0] ROM_addr, ROM_data;
reg B_in, reset;

BCD_to_Excess_3_ROM M1 (ROM_addr, B_out, ROM_data, B_in, clk, reset);
ROM_BCD_to_Excess_3 M2 (ROM_data, ROM_addr);
clock_gen M3 (clk);

initial begin #1000 $finish; end

initial begin
#10 reset = 0; #90 reset = 1;
end

initial begin
#0 B_in = 0;
#100 B_in = 0;
#100 B_in = 0;
#100 B_in = 1;
#100 B_in = 0;
end
endmodule

End of Example 8.2

8.2.8 Flash Memory

Flash memory devices are similar to EEPROMs, but have additional built-in circuitry to selectively program and erase the device in-circuit, without the need for a special programmer. They have widespread application in modern technology for cell phones, digital cameras, set-top boxes, digital TV, telecommunications, nonvolatile data storage, and microcontrollers. Flash memory is cost-competitive with a magnetic disk for capacities under 5 MB. Its low consumption of power makes it an attractive storage medium for laptop and notebook computers. Flash memories incorporate additional circuitry too, allowing simultaneous erasing of blocks of memory, for example 16 Kbytes to 64 Kbytes. Intel's StrataFlash memory\(^3\) (3-Volt technology), shown in Figure 8-11, encodes the threshold voltage levels of transistors to achieve storage of multiple bits per cell (1 to 3), achieving reduced cell area and die size for a given density. An internal state machine controls the charge placement to achieve a target threshold voltage. Like EEPROMs, flash memories are subject to fatigue, typically having about \(10^5\) block erase cycles.

\(^3\)See www.intel.com.
FIGURE 8-11 Block diagram of Intel's StrataFlash memory.
8.2.9 Static Random Access Memory (SRAM)

Read-only memories are limited to applications that require retrieval, but not storage, of information during ordinary operation. Computers and other digital systems perform many operations that retrieve, manipulate, transform, and store data, and therefore need read/write memories. For example, an application program must be retrieved from a relatively slow storage medium, such as a floppy disk or a CD-ROM, and moved on demand to a location where it can be accessed quickly by the processor. ROMs are not used to store large application programs, and they cannot dynamically store the data generated by a program’s execution. Storage registers and register files support fast, random storage, but cannot be used for mass storage because they are implemented with flip-flops and occupy too much physical area in silicon to support applications that generate and store vast amounts of data. Small register files may be integrated in an ASIC or an FPGA to avoid having to access an external (slower) memory device.

RAM is faster and occupies less area than a register file, and it serves the function of providing fast storage and retrieval of large amounts of data during the operation of a computer (e.g., a video frame buffer). The name random indicates that RAMs allow data to be written to or read from any storage location in any order. Most RAMs are volatile—the information they contain vanishes after power is removed from the device. A newer and emerging technology, nonvolatile RAM, will be discussed later in this chapter.

There are two basic types of RAMs: static and dynamic. Static RAMs (SRAMs) are implemented with a transistor-capacitor storage cell structure that does not require refresh; dynamic RAMs (DRAMs) are slower, use fewer transistors, and occupy less physical area, but they require refresh circuitry to retain stored data. They provide the densest storage devices, but their contents must be refreshed every few milliseconds; therefore DRAMs require additional supporting circuitry. SRAMs are used as fast-cache memory in a computer.

The circuit in Figure 8-12 shows the basic structure of an SRAM cell. A pair of inverters are connected in a closed loop and their outputs are tied to pass transistors attached to Bit line and its complement, Bit line_bar. SRAMs commonly use the 6-transistor circuit shown in Figure 8-13. The gate of each pass transistor is connected to the word line of the circuit. Suppose that Word enable is de-asserted, that the stored content of the cell has cell = 1 and cell_bar = 0, and that the inputs are changed to Bit line = 0 and Bit line_bar = 1. When Word enable is asserted, cell is driven to 0 and cell_bar is driven to 1. The feedback structure forces the output of one inverter to be the complement of the output of the other inverter.

The values of Bit line and Bit line_bar control the read and write operations. An array of such storage cells is configured with sense amplifiers that are used to read the contents of a cell. Data are written to the cell by precharging Bit line and Bit line_bar.

---

4In contrast, note that data are read serially from a tape storage media.
5Other schemes use as few as four transistors by replacing the p-channel pull-up transistors with depletion-load devices that function as resistors and compensate for leakage current.
null
Example 8.3

The level-sensitive Verilog description, \textit{RAM\_static}, models a simple RAM cell, without accounting for propagation delays. With active-low signals denoted by the suffix \textunderscore \textit{b}, level-sensitive behavior is modeled here by a single continuous assignment declaration with nested conditional operators decoding the status of \textit{CS\_b} and \textit{WE\_b}. If \textit{CS\_b} is not asserted the output is in the three-state mode (has the value Verilog logic value \textit{z}). If \textit{CS\_b} and \textit{WE\_b} are asserted (low), the cell is in transparent mode, and \textit{data\_out} follows \textit{data\_in}; if \textit{CS\_b} is asserted and \textit{WE\_b} is de-asserted, the cell is latched. The contents of the cell can always be read, but a host processor would access \textit{data\_out} only when \textit{WE\_b} is de-asserted. The functional schematic in Figure 8-15(a) forms \textit{RAM\_static}; the simulation results presented in Figure 8-15(b), demonstrates the cell’s behavior.

```
module RAM\_static (data\_out, data\_in, CS\_b, WE\_b);
  output data\_out;
  input data\_in;
  input CS\_b; // Active-low chip select control
  input WE\_b; // Active-low write control
  wire data\_out = (CS\_b == 0) \&\& (WE\_b == 0) \&\& data\_in : data\_out : 'bz;
endmodule
```

The Verilog description is synthesizable, and is implemented by a single four-input lookup table (LUT) in a Xilinx chip. The maximum combinational path delay after the logic has been placed and routed in a Xilinx XC3S10XL chip is 6.756 ns (the delay from the input pad for \textit{WE\_b} to the pad for \textit{data\_out}).

---

\footnote{A \textit{wire} declaration with an assignment to an expression implements the implicit combinational logic of the expression. The declaration is equivalent to a separate declaration of a \textit{wire} and a continuous-assignment statement.}

\footnote{This information is contained in the Post-Route Timing Report generated by the implementation process within the Xilinx design flow.}

End of Example 8.3
Example 8.4

The Verilog model of an SRAM cell can be modified to incorporate a single bidirectional port for use in a bus-based architecture. An additional active-low signal, \(OE_b\) (output enable) is added to the block diagram symbol (see Figure 8-16) and controls the datapaths through the three-state I/O buffers. The datapath is reduced from two signal ports to one port, which renders a great savings of package pins and total area if the data port is a wide vector. The structure of the model is shown in Figure 8-17, where the latch is implemented by a mux with feedback. The data paths for a write operation
FIGURE 8-16 SRAM cell block diagram symbol with a bidirectional data port interface to a shared bus.

FIGURE 8-17 SRAM cell with bidirectional data port configured to write external data through a bidirectional data port to the internal cell ($WE_b = 0, OE_b = 1$).

are shown. Output enable ($OE_b$) is asserted (low) during a read operation, and write enable is asserted (low) during a write operation. If $WE_b$ is asserted and $OE_b$ is not, the value at $data$ is transparent through $latch_out$, and is held when $WE_b$ is de-asserted (i.e., written to the cell). Conversely, when $WE_b$ is not asserted and $OE_b$ is asserted, the content of the cell can be read through $data$, as shown by the data paths in Figure 8-18.

The Verilog model of the RAM cell with bidirectional data port, $RAM_{static\_BD}$, is given below.

```verilog
module RAM_static_BD (data, CS_b, OE_b, WE_b);
    input data;    // Bi-directional data port
    input CS_b;    // Active-low chip select
    input OE_b;    // Active-low output enable
    input WE_b;    // Active-low write enable

module RAM_static_BD (data, CS_b, OE_b, WE_b);
    input data;    // Bi-directional data port
    input CS_b;    // Active-low chip select
    input OE_b;    // Active-low output enable
    input WE_b;    // Active-low write enable
```

1Continuous assignments are used here to illustrate another style for modeling level-sensitive behavior. The default type of the target of the assignment is a $wire$. (Some tools might require an explicit declaration of type.)
assign latch_out = ((CS_b == 0) & (WE_b == 0) & (OE_b == 1))
? data: latch_out;

assign data = ((CS_b == 0) & (WE_b == 1) & (OE_b == 0))
? latch_out : 'b0;

endmodule

![SRAM cell with bidirectional data port](image)

**FIGURE 8-18** SRAM cell with bidirectional data port: configured to read the cell contents via the bidirectional data port \((WE_b = 1, OE_b = 0)\).

Two additional modes are possible. With \(CS_b\) asserted (low), the control lines could be \(WE_b = 0, OE_b = 0\), and \(WE_b = 1, OE_b = 1\). The configurations that result are shown in Figure 8-19. The cell is latched in both cases; its contents are not affected by the external data path, and \(latch\_out\) does not affect \(data\). The contents of the cell are not available at \(data\).

The functional schematic of \(RAM\_static\_BD\), created by the Xilinx ISE synthesis tool, is shown in Figure 8-20. The schematic consists of a latch with additional logic to steer the I/O datapaths through a bidirectional data port. The synthesized and implemented circuit has \(data\) mapped to an I/O block (IOB) configured for bidirectional operation in a Xilinx XCS10XL chip. The maximum combination path delay is 8.178 ns (from \(CS_b\) to \(data\)). The slight increase in delay reflects the presence of the additional logic for the bidirectional data port (compared to the delay of the stand-alone cell).

The interface between \(RAM\_static\_BD\) and a bidirectional shared bus is illustrated in Figure 8-21, and the structure of the testbench for verifying \(RAM\_static\_BD\) is shown in Figure 8-22. A separately declared register variable, \(bus\_driver\), drives the bidirectional bus and sends data to \(RAM\_static\_BD\). The Verilog testbench, \(test\_RAM\_static\_BD\), uses a continuous assignment to assign the value of \(bus\_driver\) to \(data\_bus\) if \(OE_b\) is asserted during a write operation, and to disconnect \(bus\_driver\)
FIGURE 8-19 SRAM cell with bidirectional data port: configured for latched data and not reading or writing, (a) with \( WE_{b} = 0, OE_{b} = 0 \) and (b) with \( WE_{b} = 1, OE_{b} = 1 \).

FIGURE 8-20 SRAM cell with bidirectional data port: preoptimization functional schematic created by Xilinx ISE tools.
otherwise. Note that `data_bus` has two drivers, `bus_driver` from the testbench, and `data`, the value driven through the bidirectional port of `RAM_static_BD`. The assignments from `bus_driver` to `data_bus` must be synchronized by `WE_b` and `OE_b` to avoid bus contention (i.e., so that the bus has only one driver at a time). The bidirectional nature of the testbench is illustrated in Figure 8-22, which shows `RAM_static_BD` instantiated within the testbench, `test_RAM_static_BD`. The signals `OE_b`, `WE_b`, and `CS_b` are declared as register variables in the testbench.

```
module test_RAM_static_BD();
// Demonstrate write / read capability.
reg bus_driver;
reg CS_b, WE_b, OE_b;

wire data_bus = ((WE_b == 0) && (OE_b == 1)) ? bus_driver : '1bz;
```
RAM_static_BD M1 (data_bus, CS_b, OE_b, WE_b);

initial #4500 $finish;
initial begin
  CS_b = 1; bus_driver = 1; OE_b = 1;
  #500 CS_b = 0;
  #500 WE_b = 0;
  #100 bus_driver = 0;
  #100 bus_driver = 1;
  #300 WE_b = 1; #200 bus_driver = 0;
  #300 OE_b = 0; #200 OE_b = 1;
  #200 OE_b = 0; #300 OE_b = 1; WE_b = 0;
  #200 WE_b = 1; #200 OE_b = 0; #200 OE_b = 1;
  #500 CS_b = 1;
  #500 bus_driver = 0;
end

begin
  #3600 WE_b = 1; OE_b = 1;
  #200 WE_b = 0; OE_b = 0;
end
endmodule

The simulation results in Figure 8-23 show a sequence of values for CS_b, WE_b, and OE_b to demonstrate the modes of operation of RAM_static_BD. In the transparent mode, with WE_b = 0 and OE_b = 1, the value of data is determined by bus_driver, and latch_out is the same as data; when WE_b de-asserts, the value of data is latched (i.e., data are written to the cell). When OE_b is asserted, with WE_b de-asserted, the value of latch_out appears at data and at data_bus. When OE_b and WE_b are simultaneously asserted or de-asserted, the bus is not driven. The bus could be used by another client.

End of Example 8.4

Large SRAMs cannot be implemented practically as a simple array structure for two important reasons. Large SRAMs require wide input decoders, and the footprints of long rectangular arrays might not be as convenient as square arrays for physical layout in silicon. As an alternative, large SRAMs arrays are reorganized into nearly rectangular block structures using two levels of decoding.

11 The output of the cell can also be latched by de-asserting CS, but this is not the ordinary way to end a write cycle.
Example 8.5

A 32K × 8 SRAM can be organized in the structure shown in Figure 8-24, where the array has been partitioned into 8 blocks of size 512 × 64. A 32K memory requires a 15-bit address. The lower 6 bits of the address are passed to a bank of 8 muxes, each having a 64-bit wide datapath. The 6-bit address selects 1 bit from each datapath to form an 8-bit output word, Data Out. These same 6-bits steer Data In to 1 of 64 input lines connected to each of the 8 memory blocks. The upper 9 bits of the address are decoded by combinational logic to select one 64-bit word in each of the 8 blocks. In this reorganized structure, the address decoders have a practical size because they decode fewer outcomes, and the overall structure is nearly square, having a height of 512 cells and a width of 512 cells.

End of Example 8.5
Example 8.6

The alternative architecture shown in the block diagram in Figure 8-25 for a large SRAM has a bidirectional data port, where the column decoder, row decoder, and column I/O circuitry are represented by functional blocks adjacent to a $128 \times 128$ array of memory cells holding 2048 8-bit words. The upper 7 bits of the address word decode the 128 rows of the array, and the lower 4 bits of the address decode the 16 columns of words. The three-state devices that gate the bidirectional datapaths are not shown, but are contained in the column I/O circuitry. The Verilog model of the SRAM, \texttt{RAM}\_\texttt{2048} \_\texttt{8}, will be based on the organization of the data cells shown in Figure 8-26, where the address organization leads naturally to a row-by-row sequential access, beginning at the upper rightmost cell and proceeding ultimately to the lower leftmost cell. The model will include timing parameters, describing the propagation delays of the device, and timing checks to detect violations of operational constraints during simulation.

The Verilog model \texttt{RAM}\_\texttt{2048} \_\texttt{8} implements the structure illustrated in Figure 8-26. The companion testbench, \texttt{t\_RAM\_static\_2048} \_\texttt{8}, includes a behavior that writes a pattern of walking 1s through each column of the memory, successively, and another behavior that reads back the patterns stored in memory. Patterns are included in the testbench for simulating with and without delay (the \texttt{specify} ... \texttt{endspecify} block containing timing parameters and path delays can be commented out from the code).\footnote{The delay values used are for illustration and do not represent the fastest devices that are available with the most advanced technology.}
The simulation results shown in Figure 8-27 show the patterns written in column 9, starting at row 104, for zero-delay simulation. The three-state action of the bus and the bidirectional datapath causes data_bus to have the value xz_11 in the displayed waveforms. The results in Figure 8-28 show the patterns read back from the same locations. When nonzero propagation delays are included in the model, the simulation results in
Figure 8-27 RAM_2048_8: simulation results for writing a walking 1s pattern to memory with zero delay.

Figure 8-29 are obtained for writing and reading data from memory. The testbench includes write_probe, which reports the value that is stored in memory at the rising edge of WE_b and provides a check on the latching activity of the model.

```verilog
module RAM_2048_8 (data, addr, CS_b, OE_b, WE_b);
    parameter word_size = 8;
    parameter addr_size = 11;
    parameter mem_depth = 128;
    parameter col_addr_size = 4;
    parameter row_addr_size = 7;
    parameter Hi_Z_pattern = 8'bzzzz_zzzz;
    input [word_size-1:0] data;
    input [addr_size-1:0] addr;
    input CS_b, OE_b, WE_b;
    reg [word_size-1:0] data_int;
```
**FIGURE 8-28**  RAM_2048_8 simulation results for reading back a walking ls pattern from memory with zero delay.

```plaintext
reg [word_size-1:0] RAM_col0 [mem_depth-1:0];
reg [word_size-1:0] RAM_col1 [mem_depth-1:0];
reg [word_size-1:0] RAM_col2 [mem_depth-1:0];
reg [word_size-1:0] RAM_col3 [mem_depth-1:0];
reg [word_size-1:0] RAM_col4 [mem_depth-1:0];
reg [word_size-1:0] RAM_col5 [mem_depth-1:0];
reg [word_size-1:0] RAM_col6 [mem_depth-1:0];
reg [word_size-1:0] RAM_col7 [mem_depth-1:0];
reg [word_size-1:0] RAM_col8 [mem_depth-1:0];
reg [word_size-1:0] RAM_col9 [mem_depth-1:0];
reg [word_size-1:0] RAM_col10 [mem_depth-1:0];
reg [word_size-1:0] RAM_col11 [mem_depth-1:0];
reg [word_size-1:0] RAM_col12 [mem_depth-1:0];
reg [word_size-1:0] RAM_col13 [mem_depth-1:0];
reg [word_size-1:0] RAM_col14 [mem_depth-1:0];
reg [word_size-1:0] RAM_col15 [mem_depth-1:0];

wire [col_addr_size-1:0] col_addr = addr[col_addr_size-1:0];
wire [row_addr_size-1:0] row_addr = addr[row_addr_size-1:col_addr_size];
```
FIGURE 8.29  RAM2048: simulation results for (a) writing to, and (b) reading from memory a walking 1s pattern with nonzero propagation delay.

```
assign data = ((CS_b == 0) && (WE_b == 1) && (OE_b == 0))
? data_int: Hi_Z_pattern;
always @ (data or col_addr or row_addr or CS_b or OE_b or WE_b)
begin
  data_int = Hi_Z_pattern;
  if ((CS_b == 0) && (WE_b == 0))  // Priority write to memory
    case (col_addr)
      0: RAM_col0[row_addr] = data;
      1: RAM_col1[row_addr] = data;
      2: RAM_col2[row_addr] = data;
      3: RAM_col3[row_addr] = data;
      4: RAM_col4[row_addr] = data;
      5: RAM_col5[row_addr] = data;
      6: RAM_col6[row_addr] = data;
      7: RAM_col7[row_addr] = data;
      8: RAM_col8[row_addr] = data;
      9: RAM_col9[row_addr] = data;
```
FIGURE 8-29 Continued

10: RAM_col10[row_addr] = data;
11: RAM_col11[row_addr] = data;
12: RAM_col12[row_addr] = data;
13: RAM_col13[row_addr] = data;
14: RAM_col14[row_addr] = data;
15: RAM_col15[row_addr] = data;
endcase

else if ((CS_b == 0) & (WE_b == 1) & (OE_b == 0)) // Read from memory
    case (col_addr)
0: data_int = RAM_col0[row_addr];
1: data_int = RAM_col1[row_addr];
2: data_int = RAM_col2[row_addr];
3: data_int = RAM_col3[row_addr];
4: data_int = RAM_col4[row_addr];
5: data_int = RAM_col5[row_addr];
6: data_int = RAM_col6[row_addr];
7: data_int = RAM_col7[row_addr];
8: data_int = RAM_col8[row_addr];
9: data_int = RAM_col0[row_addr];
10: data_int = RAM_col10[row_addr];
11: data_int = RAM_col11[row_addr];
12: data_int = RAM_col12[row_addr];
13: data_int = RAM_col13[row_addr];
14: data_int = RAM_col14[row_addr];
15: data_int = RAM_col15[row_addr];
endcase
end

/* Comment out the model for a zero delay functional test.

specify

// Parameters for the read cycle
specparam t_RC = 10; // Read cycle time
specparam t_AA = 8; // Address access time
specparam t_ACS = 8; // Chip select access time
specparam t_CLZ = 2; // Chip select to output in low-z
specparam t_OE = 4; // Output enable to output valid
specparam t_OLZ = 0; // Output enable to output in low-z
specparam t_CHZ = 4; // Chip de-select to output in hi-z
specparam t_OHZ = 3.5; // Output disable to output in hi-z
specparam t_OH = 2; // Output hold from address change

// Parameters for the write cycle
specparam t_WC = 7; // Write cycle time
specparam t_CW = 5; // Chip select to end of write
specparam t_AW = 5; // Address valid to end of write
specparam t_AS = 0; // Address setup time
specparam t_WP = 5; // Write pulse width
specparam t_WR = 0; // Write recovery time
specparam t_WHZ = 3; // Write enable to output in hi-z
specparam t_DW = 3.5; // Data set up time
specparam t_DH = 0; // Data hold time
specparam t_OW = 10; // Output active from end of write

// Module path timing specifications
(addr -> data) = t_AA; // Verified in simulation
(CS_b -> data) = (t_ACS, t_ACS, t_CHZ); // Verified in simulation
(OE_b -> data) = (t_OE, t_OE, t_OHZ);

// Timing checks (Note use of conditioned events for the address setup,
// depending on whether the write is controlled by the WE_b or by CS_b.

// Width of write/read cycle
$width (negedge addr, t_WC);

// Address valid to end of write
$setup (addr, posedge WE_b && CS_b == 0, t_AW);
$setup (addr, posedge CS_b && WE_b == 0, t_AW);
// Address setup before write enabled
$ssetup (addr, negedge WE_b && CS_b == 0, t_AS);
$ssetup (addr, negedge CS_b && WE_b == 0, t_AS);

// Width of write pulse
$width (negedge WE_b, t_WP);

// Data valid to end of write
$ssetup (data, posedge WE_b && CS_b == 0, t_DW);
$ssetup (data, posedge CS_b && WE_b == 0, t_DW);

// Data hold from end of write
$hold (data, posedge WE_b && CS_b == 0, t_DH);
$hold (data, posedge CS_b && WE_b == 0, t_DH);

// Chip sel to end of write
$ssetup (CS_b, posedge WE_b && CS_b == 0, t_CW);
$width (negedge CS_b && WE_b == 0, t_CW);

endspecify

/*
*/

endmodule

/*****************************************************************************/
testbench /******************************************************************************/

module test_RAM_2048_8();

parameter word_size = 8;
parameter addr_size = 11;
parameter mem_depth = 128;
parameter num_col = 16;
parameter col_addr_size = 4;
parameter row_addr_size = 7;
parameter initial_pattern = 8'b00000_0001;
parameter Hi_Z_pattern = 8'bzzzz_zzzz;

reg [word_size -1 : 0] data_to_memory;
reg CS_b, WE_b, OE_b;

integer col, row;
wire [col_addr_size -1:0] col_addr = col;
wire [row_addr_size -1:3] row_addr = row;
wire [addr_size -1:0] addr = (row_addr, col_addr);

parameter t_WPC = 8; // Write pattern cycle time (Exceeds min)
parameter t_RPC = 12; // Read pattern cycle time (Exceeds min)
parameter latency_Zero_Delay = 5000;
parameter latency_Non_Zero_Delay = 18000;
//parameter stop_time = 7200; // For zero-delay simulation
parameter stop_time = 45000; // For non-zero delay simulation
// Three-state, bi-directional I/O bus

wire [word_size-1:0] data_bus = ((CS_b == 0) & (WE_b == 0) & (OE_b == 1))
? data_to_memory : Hi_Z_pattern;

wire [word_size-1:0] data_from_memory = ((CS_b == 0) & (WE_b == 1) & (OE_b == 0))
? data_bus : Hi_Z_pattern;

RAM_2048_8 M1 (data_bus, addr, CS_b, OE_b, WE_b); // UUT

initial #stop_time $finish;
/*
// Zero delay test: Write walking ones to memory
initial begin
    CS_b = 0;
    OE_b = 1;
    WE_b = 1;
    for (col=0; col <= num_col-1; col = col +1) begin
        data_to_memory = initial_pattern;

        for (row = 0; row <= mem_depth-1; row = row + 1) begin
            #1 WE_b = 0;
            #1 WE_b = 1;
            data_to_memory =
            {data_to_memory[word_size-2:0],data_to_memory[word_size-1]};
        end
    end
end

// Zero delay test: Read back walking ones from memory
initial begin
    #latency_Zero_Delay;
    CS_b = 0;
    OE_b = 0;
    WE_b = 1;
    for (col=0; col <= num_col-1; col = col +1) begin
        for (row = 0; row <= mem_depth-1; row = row + 1) begin
            #1;
        end
    end
end
*/

// Non-Zero delay test: Write walking ones to memory
// Writing controlled by WE_b
initial begin
    CS_b = 0;
    OE_b = 1;
    WE_b = 1;

    for (col = 0; col <= num_col - 1; col = col + 1) begin
        data_to_memory = initial_pattern;
        for (row = 0; row <= mem_depth - 1; row = row + 1) begin
            #(t_WPC/8) WE_b = 0;
            #(t_WPC/4);
            #(t_WPC/2) WE_b = 1;
            data_to_memory =
                {data_to_memory[word_size-2:0], data_to_memory[word_size-1]};
            #(t_WPC/8);
        end
    end
end

// Non-Zero delay test: Read back walking ones from memory
initial begin
    #latency_None_Zero_Delay;
    CS_b = 0;
    OE_b = 0;
    WE_b = 1;
    for (col = 0; col <= num_col - 1; col = col + 1) begin
        for (row = 0; row <= mem_depth - 1; row = row + 1) begin
            #t_RPC;
        end
    end
end

//--
reg [word_size-1:0] write_probe;
always @ (posedge M1.WE_b)
case (M1.col_addr)
    0: write_probe = M1.RAM_col0[M1.row_addr];
    1: write_probe = M1.RAM_col1[M1.row_addr];
    2: write_probe = M1.RAM_col2[M1.row_addr];
    3: write_probe = M1.RAM_col3[M1.row_addr];
    4: write_probe = M1.RAM_col4[M1.row_addr];
    5: write_probe = M1.RAM_col5[M1.row_addr];
    6: write_probe = M1.RAM_col6[M1.row_addr];
    7: write_probe = M1.RAM_col7[M1.row_addr];
    8: write_probe = M1.RAM_col8[M1.row_addr];
    9: write_probe = M1.RAM_col9[M1.row_addr];
    10: write_probe = M1.RAM_col10[M1.row_addr];
    11: write_probe = M1.RAM_col11[M1.row_addr];
    12: write_probe = M1.RAM_col12[M1.row_addr];
endcase
End of Example 8.6

The structure of `test_RAM_2048_8` is shown in Figure 8-30. The unit under test, `RAM_2048_8`, and the testbench both include bidirectional three-state I/O. The active-low write enable signal, `WE_b`, has priority over the active-low output-enable signal, `OE_b` (i.e., if `WE_b = 0`, the output is in the high-impedance condition independently of `OE_b`). This precludes bus contention by not allowing simultaneous reading and writing.

The timing parameters incorporated in the model for `RAM_2048_8` govern the transitions of the output waveforms in response to changes of the input waveforms and establish operational constraints that must be satisfied for correct operation of the device. For example, if the address is not stable when `CS_b` and `WE_b` are low, multiple memory cells can be affected while the device is in the transparent/write mode. The address access time is a key parameter that dictates the rate at which the memory can be read. Table 8-3 lists parameters describing the write cycle of a static RAM, and Table 8-4 describes the read cycle.

![Figure 8-30 test_RAM_2048_8: structure of the testbench for writing and reading patterns of walking 1s through a shared bi-directional bus.](image)

Note that we have simplified the schematic by showing a single three-state buffer instead of an actual configuration having a buffer on each bit line of each bus.
### TABLE 8.3 Parameters for the write cycle of a static RAM.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{WT}$</td>
<td>Write cycle time: Specifies the minimum period for successive writing of data to memory.</td>
</tr>
<tr>
<td>$t_{CW}$</td>
<td>Chip select to end of write: Specifies the minimum interval between the falling edge of $CS_b$ and the rising edge of $WE_b$.</td>
</tr>
<tr>
<td>$t_{AV}$</td>
<td>Address valid to end of write: Specifies the minimum interval between a change in the address and the end of write (the rising edge of $WE_b$).</td>
</tr>
<tr>
<td>$t_{AS}$</td>
<td>Address setup time before write: Specifies the width of the interval over which the address must be stable prior to the falling edge of $WE_b$.</td>
</tr>
<tr>
<td>$t_{WP}$</td>
<td>Write pulse width: Specifies the minimum width of the write pulse.</td>
</tr>
<tr>
<td>$t_{WR}$</td>
<td>Write recovery time: Specifies the minimum interval between the rising edge of $WE_b$ and the end of the write cycle.</td>
</tr>
<tr>
<td>$t_{W}H$</td>
<td>Write enable to output in high-z: Specifies the minimum interval between the falling edge of $WE_b$ and the output entering the high-impedance state.</td>
</tr>
<tr>
<td>$t_{DS}$</td>
<td>Data setup time: Specifies the minimum width of the interval over which the data must be stable prior to the rising edge of $WE_b$.</td>
</tr>
<tr>
<td>$t_{DH}$</td>
<td>Data hold time after end of write: Specifies the minimum interval that the data must be stable after the rising edge of $WE_b$.</td>
</tr>
<tr>
<td>$t_{OW}$</td>
<td>Output active from end of write: Specifies the earliest time that the output is available after the rising edge of $WE_b$.</td>
</tr>
</tbody>
</table>

The timing parameters of a write cycle are illustrated in Figure 8-31. Two cases must be considered: (1) the operation controlled by $WE_b$ with $CS_b = 0$ (the device is selected) and $OE = 1$ (the read cycle is not active), and (2) the operation controlled by $CS_b$ with $WE_b = 0$ (write is enabled), and $CS_b = 0$.

In the former case (shown in Figure 8-31(a)), the address must be stable and the chip must be selected before the falling edge of $WE_b$. The write cycle occurs over an interval of width $t_{WC}$, which includes the times at which the address lines may be changed. The address setup time, $t_{AS}$, establishes the minimum time between the stable address and the falling edge of $WE_b$. This constraint ensures that the address-decoding circuitry is stable before the write is attempted. The enable input of a transparent latch must satisfy a minimum pulselwidth constraint ($t_{WP}$); similarly, the time from chip select to the end of the write cycle ($t_{CW}$) must also satisfy a pulselwidth constraint ($t_{CW}$). While $WE_b$ is low the device is in the transparent mode and the three-state device driving $data_{in}$ is in the high-impedance state. The device enters this state, with a delay specified by $t_{W}H$, when $WE_b$ is asserted. The data to be written to the SRAM must satisfy a setup time constraint ($t_{DW}$) and a hold time constraint ($t_{DH}$) relative to the rising edge of $WE_b$. Note: Figure 8-31a is drawn to illustrate the rising edge of $CS_b$ occurring after the rising edge of $WE_b$. The address must be stable for an interval.

---

1. We will consider timing constraints in more detail in Chapter 11.
2. If the rising edge of $CS$ occurs before the rising edge of $WE$ the timing constraints must be applied relative to the rising edge of $CS$. 
TABLE 8-4 Parameters for the read cycle of a static RAM.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{RC}$</td>
<td>Read-cycle time: Specifies the minimum period for successive reading of data from memory.</td>
</tr>
<tr>
<td>$t_{AA}$</td>
<td>Address access time: A key performance parameter specifying the minimum interval between a change in the address and the availability of valid data retrieved from memory.</td>
</tr>
<tr>
<td>$t_{ACS}$</td>
<td>Chip select access time: Specifies the minimum interval between assertion of chip select and the availability of valid data from memory, assuming that $OE_b = 0$ and $WE_b = 1$ before $CS_b = 0$.</td>
</tr>
<tr>
<td>$t_{CLZ}$</td>
<td>Chip select low z: Specifies the minimum interval between assertion of chip select and the output leaving the high-impedance state.</td>
</tr>
<tr>
<td>$t_{OE}$</td>
<td>Output enable to output valid: Specifies the minimum interval between the falling edge of $OE_b$ and the availability of valid data from memory.</td>
</tr>
<tr>
<td>$t_{OLZ}$</td>
<td>Output enable to output in low Z: Specifies the minimum interval between the falling edge of $OE_b$ and the output leaving the high-impedance state.</td>
</tr>
<tr>
<td>$t_{CHZ}$</td>
<td>Chip deselect to output in high Z: Specifies the minimum interval between the rising edge of $OE_b$ and the output entering the high-impedance state.</td>
</tr>
<tr>
<td>$t_{OHZ}$</td>
<td>Output disable to output in high Z: Specifies the minimum interval between the rising edge of $OE_b$ and the output entering the high-impedance state.</td>
</tr>
<tr>
<td>$t_{OH}$</td>
<td>Output hold from address change: Specifies the minimum interval that the output remains valid after a change in the address.</td>
</tr>
</tbody>
</table>

($t_{WR}$) the write recovery time, after the rising edge of $WE_b$, and the bus becomes available after an interval ($t_{OW}$) expires from the rising edge of $WE_b$. The interval from the onset of a stable address to the end of the write cycle is represented by the parameter $t_{AW}$.

When $WE_b$ is low before the falling edge of $CS_b$, and rises after the rising edge of $CS_b$, the SRAM is controlled by $CS_b$ and is characterized by the waveforms in Figure 8-31b. In this case, the setup and hold time constraints for the data on the bus are relative to the rising (latching) edge of $CS_b$.

The two modes of the read cycle are illustrated in Figure 8-32. In Figure 8-32(a) the data is determined by the address (with $CS_b = 0$ and $WE_b = 1$, and is valid $t_{AA}$ time units after the address is stable. In Figure 8-32(b), the data becomes valid after $t_{ACS}$ time units from the falling edge of $CS_b$.

8.2.10 Ferroelectric Nonvolatile Memory

Ferroelectric materials are so named because their electrical characteristics resemble those of ferromagnetic materials. Despite the suggestion implied by their name, ferroelectric materials have nothing to do with ferromagnetics. Their similarity is primarily in the fact that certain ferroelectric materials can exhibit a significant hysteresis effect, but it is not associated with magnetic properties. Instead, the hysteresis effect in a
Ferroelectric is due to the so-called spontaneous electrical polarization of a ferroelectric material under the influence of an applied voltage. When power is removed the residual polarization behaves like a bistable memory device. Ferroelectric memories hold the promise of replacing other nonvolatile memories, such as EEPROMs in applications that require short programming time and low power consumption. Contactless smart cards, digital cameras, and utility meters are considered to be appropriate applications for this technology. EEPROMS and flash memories are also nonvolatile, and have lower power to read data than ferroelectrics. Ferroelectric memories can also be embedded with other devices. This technology is expected to mature to have
competitive circuit densities compared to other alternatives. See Sheilhose and Gulak [3] for a survey of circuits exploiting ferroelectric technology.

### 8.3 Programmable Logic Array (PLA)

PLAs were developed for integrating large two-level combinational logic circuits. Like ROMs, their architecture consists of two arrays, shown in Figure 8-33. One array implements the AND operation that forms a product term (i.e., a Boolean cube, possibly

---

**FIGURE 8-33** AND-OR plane structure of a PLA.
a minterm), and another array implements the OR operation that forms a sum of the product (SOP) terms. A PLA implements a two-level Boolean function in SOP form.

Unlike ROMs, both arrays of a PLA are programmable (mask-programmable or one-time field-programmable). However, the AND plane does not implement a full decoder, but instead forms a limited number of product terms. The programmable OR-plane forms expressions by OR-ing together product terms (cubes). An \( n \times p \times m \) PLA has \( n \) inputs, \( p \) product terms (outputs of the AND plane), and \( m \) output expressions (from the OR plane). A \( 16 \times 48 \times 8 \) PLA has 48 product terms. A 16-input ROM would have \( 2^{16} = 65,536 \) input patterns decoded as minterms and available to form the outputs. A PLA would have 8 outputs formed from the 48 product terms (not necessarily minterms).

A PLA implements general product terms, not just minterms or maxterms. Because it has limited AND-plane resources, minimal SOP forms must be found so that device resources might accommodate an application's requirements for product terms. PLA minimization algorithms led to development of widely used synthesis algorithms having general application to ASICs [4].

The circuit structure of a PLA implemented in CMOS technology is shown in Figure 8-34.

![Diagram of PLA circuit structure](image)
The AND-OR plane structure shown implements NOR-NOR logic, which reverts to equivalent AND-OR logic with inverted inputs and three-stated inverters at the outputs. Each input is available as a literal in complemented and uncomplemented form. A programmable link in the AND plane determines whether the associated input literal (or its complement) is connected to a buffered word line.

Programming determines whether inputs have a link to word lines and whether word lines connect to the output lines. A word line may be linked to an input or its complement, but not both. Each word line is connected to a pull-up resistor (active device). The aggregate of linked input literals and complements of input literals forms a Boolean cube at the word line to which the links are attached. Unconnected inputs have no effect on a word line. In the absence of an asserted and connected input literal (or its complement) a word line is pulled up. In the absence of an asserted (high) level on its linked word lines, a column line is pulled high. An asserted input turns on a connected link transistor in the AND plane and pulls the word line to ground by overriding its pull-up resistor. A column line is asserted (high) if all of its connected word lines are de-asserted (low). An asserted word line turns on a connected link transistor in the OR plane, causing its connected word line to be pulled down. A column line is low if any of its connected word lines is asserted (high). If any word line is asserted (high), a connected column line is pulled down. A column line is asserted (high) only if all of its connected word lines are de-asserted (low).

To see that the circuit shown in Figure 8-35 exhibits wired-AND logic at its word lines, note that

\[
W1 = A'B' \\
W1' = (A + B) \\
W2 = C'D' \\
W2' = (C + D')
\]

![Figure 8-35 Wired-OR logic of a PLA](image-url)
Similarly, the column lines exhibit wired-\textit{OR} behavior, where \( W \) is low if \( W1 \) or \( W2 \) is high; otherwise \( W \) is high (pull-up):
\[
W' = W1 + W2 \\
W = (W1 + W2)' \\
Y = W' = W1 + W2 = A'B' + C'D' \\
Y = (A + B)' + (C + D)'
\]

The overall structure is that of \textit{NOR-NOR} logic, with
\[
Y' = [(A + B)' + (C + D)']
\]
The equivalent circuit is shown in Figure 8.36(a), and an equivalent \textit{OR-AND} structure is shown in Figure 8.36(b).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{8.36.png}
\caption{Equivalent circuit structures for PLA logic: (a) \textit{NOR-NOR} logic, and (b) \textit{OR-AND} logic with inverted inputs.}
\end{figure}

\subsection{8.3.1 PLA Minimization}

The area of a PLA depends primarily on the number of word lines (i.e., distinct product terms), so it is advantageous to find ways to reduce the number of product terms by sharing logic as much as possible. One approach would be to use Karnaugh maps or other minimization methods to reduce each Boolean expression. However, minimization of individual Boolean functions does not necessarily produce an optimal PLA implementation. Minimization of a set of Boolean functions, as an aggregate, can exploit don't-cares and opportunities to share logic because a product term that is generated to form one output expression can be used in another output expression that uses the same term. Alternatively, a common factor in a product of sums form can be shared by multiple functions that have the same factor.

\subsubsection*{Example 8.7}

Consider the three Boolean functions shown below, with their K-maps shown in Figure 8.37. Before minimization, the implementation would require 13 product terms (word lines) to support the cubes of the three functions.

\[
f_1(a, b, c, d) = \Sigma m(1, 6, 7, 9, 13, 14, 15) \\
f_2(a, b, c, d) = \Sigma m(6, 7, 8, 9, 13, 14, 15) \\
f_3(a, b, c, d) = \Sigma m(1, 2, 3, 9, 10, 11, 12, 13, 14, 15)
\]
After each function is individually minimized, the total number of cubes is 8, a savings of nearly 40%. To minimize the functions as an aggregate, a task easily done by modern synthesis tools, we re-cover the functions and identify common cubes by considering pairwise and threewise intersections, as shown in Figure 8-38. The final result needs only five word lines, having eliminated an additional four word lines.

**End of Example 8.7**
Systematic manual minimization of multiple output functions is feasible for up to three functions, with a maximum of four inputs [5]. Otherwise, a computer-based approach is needed (e.g., espresso [4] and MIS-II [6-8]).

The tabular format shown in Figure 8.39 can be used to specify the functionality of a PLA. Table rows correspond to PLA rows (wordlines). Table columns list inputs and functions indicating whether an input is in a cube, and whether a cube is in a function. Inputs are coded as 1 (care-on), 0 (care-off), and—(don’t-care). Outputs are coded as 1 (contains the wordline), or 0 (does not contain the wordline).

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f₁</th>
<th>f₂</th>
<th>f₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>abd</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bc'd</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>b'c</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bc</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>abd</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIGURE 8.39** Tabular format for specifying the structure of a PLA.

ROMs require canonical data (i.e., a complete truth table), but PLAs require only minimum SOP Boolean forms. The cubes in a minimized PLA table may cover multiple minterms, and a given input vector may assert multiple output functions. For a given input vector, the cubes are formed by AND-ing the complemented and uncomplemented literals in a row; the outputs are determined by column-wise OR-ing the cube (word line) entries having a 1. A simplified representation of a PLA is shown in Figure 8.40. The filled circles indicate whether a literal or its complement is used in a cube and whether a cube is used in an expression.

### 8.3.2 PLA Modeling

An application for a PLA must be compatible with the limited number of product terms (word lines) that can fit within the device. PLAs are used to implement the next-state and output-forming logic of large state machines that control more complex sequential machines, such as computers. PLAs are a more attractive implementation than ROMs for large state machines because the area of a PLA can be minimized and tailored to an application.

Verilog includes a set of system tasks for modeling multiinput, multioutput PLAs. PLAs implement two-level combinational logic by a array structure of AND, NAND, OR, and NOR logic array planes. The “personality” file, or matrix, of the PLA specifies the physical connections of transistors forming the product of input terms (cubes) and the sums of those products to form the Boolean expressions of the outputs. See “Selected System Tasks and Functions” at the companion web site.
Example 8.8

The statements below illustrate calls to Verilog's built-in PLA system tasks describing synchronous and asynchronous arrays and planes:

\[
\begin{align*}
\text{async\&array} & \quad (\text{PLA\_mem}, \{\text{in0, in1, in2, in3, in4, in5, in6, in7}\}, \{\text{out0, out1, out2}\}); \\
\text{sync\&or\&plane} & \quad (\text{PLA\_mem}, \{\text{in0, in1, in2, in3, in4, in5, in6, in7}\}, \{\text{out0, out1, out2}\}); \\
\text{async\&and\&array} & \quad (\text{PLA\_mem}, \{\text{in0, in1, in2, in3, in4, in5, in6, in7}\}, \{\text{out0, out1, out2}\}); \\
\text{async\&and\&array} & \quad (\text{PLA\_mem}, \{\text{in0, in1, in2, in3, in4, in5, in6, in7}\}, \{\text{out0, out1, out2}\});
\end{align*}
\]

The outputs of the asynchronous arrays are updated whenever an input signal changes value or whenever the personality matrix of the PLA changes during simulation. The synchronous types are updated when evaluated in a synchronous behavior. Both forms update their outputs with zero delay.

The personality matrix of a PLA specifies the cubes that form the inputs to the PLA and the expressions forming the outputs of the PLA. The data describing the personality is stored in a memory whose width accommodates the inputs and outputs of the PLA and whose depth accommodates the number of outputs.

There are two ways to load data into the personality matrix: (1) using the \$readmemb task, read the data from a file, and (2) load the data directly with procedural assignment statements. Both methods can be used at any time during a simulation to reconfigure the PLA dynamically.
Two formats may be used to describe the contents of an array: array and plane. The array format stores either a 1 or 0 in memory to indicate whether a given input is in a cube, and whether a given cube is in an output. For example, the array format shown below indicates that the cube in \( \text{in1} \) & \( \text{in2} \) & \( \text{in3} \) is formed and used in \( \text{out1} \), but not in \( \text{out2} \). The cube \( \text{in1} \) & \( \text{in3} \) is used in \( \text{out2} \).

\[
\begin{array}{cccccc}
\text{in1} & \text{in2} & \text{in3} & \text{out1} & \text{out2} \\
1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

End of Example 8.8

Example 8.9

Suppose we want to implement the logic of the following Boolean equations with a PLA:

\[
\begin{align*}
\text{out0} &= \text{in0} \& \text{in1} \& \text{in2} \& \text{in3} + \text{in4} \& \text{in5} \& \text{in6} \& \text{in7} + \text{in1} \& \text{in3} \& \text{in5} \& \text{in7} \\
\text{out1} &= \text{in1} \& \text{in3} \& \text{in5} \& \text{in7} + \text{in4} \& \text{in5} \& \text{in6} \& \text{in7} \\
\text{out2} &= \text{in0} \& \text{in2} \& \text{in4} \& \text{in6} + \text{in4} \& \text{in5} \& \text{in6} \& \text{in7} + \text{in1} \& \text{in3} \& \text{in5} \& \text{in7}
\end{align*}
\]

The expressions use four distinct cubes:

\[
\begin{align*}
\text{in0} & \text{in1} \text{in2} \text{in3} \\
\text{in0} & \text{in2} \text{in4} \text{in6} \\
\text{in4} & \text{in5} \text{in6} \text{in7} \\
\text{in1} & \text{in3} \text{in5} \text{in7}
\end{align*}
\]

The personality data of the PLA is shown below, and is placed in a text file, \textit{PLA_data.txt}. The data indicate the presence of a literal by a 1, and the absence of a literal by 0, listed in ascending order of the inputs. There is one row for each cube, a column for each input, and the last three columns indicate whether a row cube is present in each of the three output functions.
The Verilog model PLA_array describes a PLA that forms three output functions of eight Boolean input variables. The personality of the array is stored in the array of words PLA_mem, whose width corresponds to the width of the personality matrix and whose depth is determined by the number of Boolean expressions that will be formed as outputs. Hence, the array of words has a width of 11 bits and a depth of three words.

```verilog
module PLA_array (in0, in1, in2, in3, in4, in5, in6, in7, out0, out1, out2);
    input [in0, in1, in2, in3, in4, in5, in6, in7];
    output out0, out1, out2;
    reg out0, out1, out2;
    reg [0: 10] PLA_mem [0: 2]; // 3 functions of 8 variables

    initial begin
        $readmem "PLA_data.txt", PLA_mem;
        $async$and$array
            (PLA_mem, [in0, in1, in2, in3, in4, in5, in6, in7], [out0, out1, out2]);
    end
endmodule
```

**End of Example 8.9**

The PLA in Example 8.4 is configured by the *initial* behavior at the beginning of a simulation. The simulator reads the file PLA_data.txt and loads the data into the declared memory, PLA_mem. Note that the inputs and outputs are declared in ascending order. When an input to the module changes value the array is evaluated to form updated values of out0, out1, and out2.

The array format requires that the complement of a literal be provided separately as an input if it is needed to form a cube. On the other hand, the plane format encodes the personality matrix, according to the format in Table 8-5, which was adopted from the Espresso format developed at the University of California at Berkeley [4].

<table>
<thead>
<tr>
<th>Table Entry</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The complemented literal is used in the cube.</td>
</tr>
<tr>
<td>1</td>
<td>The literal is used in the cube.</td>
</tr>
<tr>
<td>x</td>
<td>The worst case of the input is used.</td>
</tr>
<tr>
<td>z</td>
<td>Don't care; the input has no significance.</td>
</tr>
<tr>
<td>?</td>
<td>Same as z.</td>
</tr>
</tbody>
</table>
Example 8.10

Suppose the logic to be implemented in a PLA is described by the following statements:

\[
\begin{align*}
\text{out0} &= \text{in0} \& \sim \text{in2}; \\
\text{out1} &= \text{in0} \& \text{in1} \& \sim \text{in3}; \\
\text{out2} &= \sim \text{in0} \& \sim \text{in3};
\end{align*}
\]

In the plane (Espresso) format, the personality of the PLA is described by:

\[
\begin{align*}
4'b1?0? \\
4'b11?0 \\
4'b0?0
\end{align*}
\]

The rows correspond to the outputs and are listed in descending order. A row defines the conditions of the inputs that assert that output. For example, the inputs 1000 and 1101 will both assert the first output. A Verilog description of the PLA is given below.

```verilog
module PLA_plane (in0, in1, in2, in3, in4, in5, in6, in7, out0, out1, out2);
input in0, in1, in2, in3, in4, in5, in6, in7;
output out0, out1, out2;
reg out0, out1, out2;
reg [0:3] PLA_mem [0:2]; // 3 functions of 4 variables
reg [0:4] a;
reg [0:3] b;

initial begin
    $async$and$array
    (PLA_mem, [in0, in1, in2, in3, in4, in5, in6, in7], {out0, out1, out2});
    PLA_mem [0] = 4'b1?0?; // Load the personality matrix
    PLA_mem [1] = 4'b11?0;
    PLA_mem [2] = 4'b0?0;
end
endmodule
```

End of Example 8.10

8.4 Programable Array Logic (PAL)

PALs emerged after PLAs, and simplified the dual-array structure by fixing the OR plane and allowing only the AND plane to be programmed. Each output is formed from a specified number of word lines, and each word line is formed from a small number of product terms. One of the more popular devices, the PAL16L8, has the structure shown in Figure 8-41. The device has 16 inputs and 8 outputs; its package has 20 pins,

\[^{16}\text{Note: PAL is a trademark of Applied Micro Devices (AMD).}\]
including power and ground. Each input is available in true or complemented form. There are eight 7-input OR gates connected to word lines from the AND plane. Each word line can be connected to any input or its complement. An eighth word line in each group controls a three-state inverter that is driven by the group’s OR gate. Each output implements a sum of products expression from at most seven terms. The device has only 20 pins, so six of the pins are bidirectional. The AND gate (not shown) that is associated with each word line is permanently connected to an OR gate and cannot be shared with any other OR gate, but six of the outputs are connected to three-state inverters and can be fed back to the AND array to be shared with other AND gates, which accommodates expressions having more than seven product terms. A bidirectional pin also makes it possible for the device to implement a transparent latch by combina-
tional feedback. PLD-based latches have application as address decoder/latches in microprocessor systems [1]. Modern PAL devices are manufactured with registered outputs and selectable output polarity.

Early PAL devices were implemented in bipolar technology; like ROMs, they were programmed by vaporizing metal links. Contemporary devices are implemented in CMOS technology with floating-gate link transistors.
8.5 Programmability of PLDs

ROMs, PLAs, and PLDs are implemented in similar array structures. Table 8-6 compares the options that are presented for programming the devices. PLAs provide the greatest flexibility and are used for large, complex, combinational logic circuits.

8.6 Complex PLDs (CPLDs)

As technology has evolved, more dense and complex devices have been developed to implement large structures (e.g., up to 1024 functions) of field-programmable combinational and sequential logic, and are referred to as complex PLDs, or CPLDs. The high-level architecture of a typical CPLD (shown in Figure 8-42) is formed as a structured array of PLD blocks that have a programmable on-chip interconnection fabric. Aside from increased performance, these architectures overcome the limitation of

<table>
<thead>
<tr>
<th>Programmable Block</th>
<th>AND Plane</th>
<th>OR Plane</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>PLA</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>PAL</td>
<td>P</td>
<td>NP</td>
</tr>
</tbody>
</table>

NA = not applicable, P = programmable, NP = not programmable.

FIGURE 8-42 High-level architecture of a CPLD.
conventional PLDs, which have a relatively small number of inputs. CPLDs have wide
inputs, but not at the expense of a dramatic (i.e., exponential) increase in area. The
device area of a conventional PLD will be scaled by a factor of $2^n$ if its input dimension is
scaled by a factor of $n$. An array of identical interconnected PLDs will accommodate
the increased dimension of the input space too, but the cell area increases by a factor of
only $n$, in addition to the area required by the interconnection fabric. Thus, CPLDs are
distinguished by having wide fan-in AND gates. Large CPLDs do not connect the output
of every macrocell to an output pin, but they typically have 100% connectivity
between macrocells.

Each PLD block of a CPLD has a PAL-like internal structure that forms combinational
logic functions of its inputs. The outputs of the macrocells in the PLDs can be
programmed to route to the inputs of other logic blocks to form more complex,
multilevel logic beyond the limitations of a single logic block. Some CPLDs are electrically
erasable and reprogrammable (EPLD). CPLDs are suited for wide fan-in And-Or logic
structures, and exploit a variety of programming technologies: SRAM/transmission gates,
EPROM (floating-gate transistors), and antifuses.\(^7\)

8.7 Altera MAX 7000 CPLD

Altera PLDs use CMOS technology with floating-gate EEPROM configuration mem-
ory cells to establish routing. Their CPLD families include the MAX 5000, 7000, and
9000 series. The architecture of Altera’s 7000\(^6\) series devices (EPM7032, EPM7064,
and EPM7096 Devices) is shown in Figure 8-43.

The structure consists of an array of logic array blocks (LABs), a programmable
interconnect array (PIA), and an array of programmable I/O blocks. Each LAB has 36
inputs and 16 outputs and contains 16 macr0cells, each containing combinational logic
and a flip-flop for either combinational or sequential operation. The PIA is a global
bus that establishes connectivity between multiple LABs, dedicated inputs, and I/O
pins. The PIA provides full connectivity with predictable timing between the logic-cell
outputs and the inputs to the LABs. The level of connectivity reduces the density of
the structure compared to the denser channel-based architectures used in FPGAs.

The I/O control blocks establish connectivity between the I/O pins and the PIA
and LABs. The (dedicated) global inputs clock (GCLK) and active-low clear (GCLnR)
connect to all macrocells. The outputs are enabled by active-low signals OE1n and
OE2n, which connect to all I/O control blocks. From 8 to 16 LAB outputs can be
programmed to route to I/O pins, and from 8 to 16 I/O pins can be programmed to route
through the I/O control block to the PIA. Each LAB contains a finer-grain array of pro-
grammable macrocells, each having the same basic architecture, shown in Figure 8-44.

A macrocell consists of a logic array (programmable AND plane), a product-
term select matrix that drives an OR gate, and a programmable flip-flop. The program-
ma ble array functions as a mini-PAL, forming product terms (not minterms), which are
OR-ed to form an expression. Each macrocell has up to 36 inputs from the PIA and up

\(^7\)Antifuses are programmable low-resistance electrical links.

\(^6\)See www.altera.com for device data sheets and additional resources about PLDs.
FIGURE 8-43 Altera 7000 Series: architecture for the EPM7032, EPM7064, and EPM7096 devices.

FIGURE 8-44 Altera 7000 Series: macrocell architecture for the EPM7032, EPM7064, and EPM7096 devices.
to 16 additional inputs formed as expander signals (discussed below). Each macrocell generates and provides five product terms to its product-term select matrix (i.e., each macrocell alone can form an expression having up to five cubes). The product-term select matrix can steer a product term to the input of an OR gate, an XOR gate, a logic expander, or to the preset, clear clock or enable input of a flip-flop.

The flip-flop of each macrocell can be individually programmed to implement a D, T, or J-K flip-flop, or be programmed to implement an set–reset (S-R) latch for use in sequential machines. For example, a macrocell can be converted to a T-type flip-flop to give more efficient implementations of counters and adders. (Connect the output of the flip-flop to one input of an XOR gate, and drive the other input by the toggle signal (recall that a T-type flip-flop has the characteristic equation: \( Q^+ = T \oplus Q \)).)

The flip-flop of each macrocell can be synchronized in three ways: (1) by a global clock, (2) by a product term, or (3) by either of the previous modes but gated by a term from the product-term select matrix. Note that the product-term signal could be generated by the logic of the macrocell or could come directly from an I/O pin. Likewise, the clear input can be programmed to be either the global clear signal or a product term from the product-term select matrix. The preset input is a product term from the product-term select matrix. The signal formed by the combinational logic is routed by a programmable register bypass switch to either the I/O control block (combinational output) or to the D input of the flip-flop (registered output). The actions of clear and preset are asynchronous.

The macrocells of the Altera EPM7000E and EPM7000S devices have the architecture shown in Figure 8.45. The circuit is nearly identical to that of the EPM7032,
EPM7064, and EPM7096 devices, but it has two global clocks and a programmable fast select mux/switch in the datapath to the $D$ input of the programmable flip-flop. The fast select switch bypasses the PIA and the macrocell's combinational logic to connect an I/O pin directly to the $D$ input of the flip-flop, for fast (2.5 ns) setup times. The true or complemented value of either $GCLK1$ and $GCLK2$ can be the two clocks.

### 8.7.1 Shareable Expander

There are two types of logic expanders that let a macrocell exploit the unused product terms of other macrocells in the same LAB to efficiently synthesize fast, complex, logic with more than five product terms.

Shareable expanders increase the number of literals that can form an expression. One unused product term of each macrocell in a LAB can be inverted and fed back to the logic array as a shareable expander, where it can be used by any other macrocell $AND$ gate in the same LAB. This creates a more complex three-level $NAND-AND-OR$ structure by replacing a literal with an entire product term from another macrocell. Sharing product terms in a LAB can reduce the overall resources required to implement logic. Figure 8-46 shows how a shared expander is formed in a LAB.

**FIGURE 8-46** Altera 7000 Series: Shared Expander.
8.7.2 Parallel Expander

The output of each macrocell ordinarily consists of the sum of only its five product terms. This accommodates most, but not all logic. Parallel expanders (Figure 8-47) increase the number of cubes that can form an expression by chaining (successively OR-ing) the output of a macrocell with the output of a neighboring macrocell. Each macrocell's five terms can be summed with up to 15 more product terms provided by parallel expanders from neighboring macrocells in the same LAB. The parallel expanders are allocated in groups of up to five additional product terms. The 16 macrocells of a LAB are organized into two groups of 8 cells each for the purpose of lending or borrowing parallel expanders (e.g., macrocells 8 down to 1, and macrocells 16 down to 9). A macrocell borrows parallel expanders from lower-number macrocells in the same group of 8. For example, macrocell 1 in LAB 1 can only lend a parallel expander; and macrocell 8 can only borrow a parallel expander from macrocell 7, macrocells 7 and 6, or from macrocells 7, 6, and 5. A macrocell needing 20 product terms can obtain 5 more terms from each of three neighboring macrocells, plus its own 5 terms, for a total of 20 terms.

FIGURE 8-47 Altera 7000 Series: parallel expander.
To summarize the shared resources in a LAB: (1) shareable expanders make one unused product term from a macrocell available to any or all macrocells in the same LAB, and (2) parallel expanders chain the outputs of neighboring macrocells in a LAB to form an expression summing up to 20 product terms.

### 8.7.3 I/O Control Block

Each I/O pin can be programmed as a dedicated input or output pin, or as a bidirectional pin under the control of global signals $OE1n$ and $OE2n$. If the enable signal of the three-state buffer in Figure 8-48 is programmed to VCC by the block's logic, the output of the macrocell is hard wired to drive the I/O pin. If the enable signal is programmed to ground the buffer is three-stated (disconnected) and the I/O pin functions as an input pin, providing an external signal to the PIA. Otherwise, either $OE1n$ or $OE2n$ can dynamically control the bidirectional operation of the circuit, providing connection to a shared bus.

### 8.7.4 Timing Considerations

Altera's architecture uses a PIA to connect LABs and I/O pins. All signals are available throughout the device. As a result, timing is predictable, an advantage over channel-based architectures, which have routing-dependent delays. The tradeoff is that a channel-routed architecture can achieve a denser implementation by having a sparser interconnect fabric.

### 8.7.5 Device Resources

A CPLD vendor's synthesis tool optimizes the Boolean logic describing a design, partitions the results to fit into LABs, allocates the functional units to LABs, and establishes the programming needed to configure the device. The tool attempts to optimize the design's utilization of resources, subject to speed constraints imposed by the application and the available devices.

![Diagram](image.png)

**Figure 8-48** Altera 7000 Series: the I/O control block for the EPM7032, EPM7064, and EPM7096 devices.
8.7.6 Other Altera Device Families

Altera manufactures an in-system programmable (ISP) version of the 7000 series devices, called the 7000S series. It eliminates the need for a special programmer box. The MAX 9000 series devices have more routing resources and greater density than the 7000 series. Their Flex 6000, Flex 8000, Flex 10, and Flex 20 families of devices use SRAM-based technology (i.e., RAM-programmed transmission gates) to form a channel-based interconnection fabric, rather than EEPROM technology, and use lookup tables (LUTs) to implement logic. We will consider these devices with FPGAs.

8.8 XILINX XC9500 CPLDs

The Xilinx XC9500 family of CPLDs are flash-based (EEPROM) and in-system programmable. The devices are organized as an array of functional blocks in a PAL-like structure with wide AND gates and fast flip-flops. Each function block contains up to 18 independent macrocells (see Figure 8-49), and can accommodate 54 inputs and drive 18 outputs (depending on the packaging). A FastCONNECT switch-matrix technology ensures that an application can be fully routed even when the device utilization is high. I/O blocks (IOBs) buffer the inputs and outputs to the device and also receive the global clock and S-R signals. The output buffers have a programmable slew rate.

![Diagram of Xilinx XC9500 architecture](image-url)
The PAL-like architecture of a function block receives 54 complemented and uncomplemented inputs and can form up to 90 product terms from the inputs. A "product-term allocator" allocates up to 90 product terms to each macrocell in the function block to form an SOP expression. Each macrocell can receive five direct product terms from the AND array, and up to 10 more product terms can be made available from other uncommitted product terms in other macrocells in the same functional block, with a negligible increase in delay. Moreover, partial sums of products can be combined over several macrocells to produce expressions with more than 18 product terms.

Each macrocell can be independently configured for combinational or registered functionality, and receives global clock, output enable, and S-R signals. The pin-to-pin delays of the XC9500 device family are short, and support high system clock rates up to 150 MHz. They should be used in high fan-in state machines, in which speed is a dominant constraint.

The architecture of a macrocell is shown in Figure 8-50. The flip-flop can be configured as a D- or T-type flip-flop, with synchronous or asynchronous S-R operation. The register’s clock can be any of three global clocks, or a product term. The register can also be bypassed to provide direct output. Each macrocell has five direct inputs from the AND-array, which can be used to implement combinational functions or control inputs (clock, clock enable, set-reset, and output enable).

**FIGURE 8-50** Xilinx XC9500: Macrocell architecture.
Table 8-7: Xilinx XC9500: significant device characteristics.

<table>
<thead>
<tr>
<th>XC9500 CPLDs</th>
<th>XC9536XV</th>
<th>XC9572XV</th>
<th>XC95144XV</th>
<th>XC95288XV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macrocels</td>
<td>36</td>
<td>72</td>
<td>144</td>
<td>288</td>
</tr>
<tr>
<td>Usable Gates</td>
<td>800</td>
<td>1,600</td>
<td>3,200</td>
<td>6,400</td>
</tr>
<tr>
<td>Registers</td>
<td>36</td>
<td>72</td>
<td>144</td>
<td>288</td>
</tr>
<tr>
<td>Tpd (ns)</td>
<td>3.5</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>TSU (ns)</td>
<td>2.8</td>
<td>3.1</td>
<td>3.1</td>
<td>3.7</td>
</tr>
<tr>
<td>TCO (ns)</td>
<td>1.8</td>
<td>2.0</td>
<td>2.0</td>
<td>2.5</td>
</tr>
<tr>
<td>f_sys (MHz)</td>
<td>278</td>
<td>250</td>
<td>250</td>
<td>222</td>
</tr>
</tbody>
</table>

Characteristics of the XC9500 device family are shown in Table 8-7. Note that the propagation delay parameters are fixed for a particular device, allowing predictable performance independently of the internal placement and routing of the design.

8.9 Field-Programmable Gate Arrays

CPLDs are characterized by an array of PAL-like blocks of combinational logic implementing wide-input SOP expressions. They have predictable timing and a crossbar type of interconnection fabric, and are suited for low- and medium-density applications. FPGAs have a more complex and register-rich tiled architecture of functional units and a flexible channel-based interconnection fabric. Featuring flash-based reconfigurability, CPLDs can be reprogrammed a limited number of times, but FPGAs have no practical limit on their reconfigurability. FPGAs are suitable for medium- and high-density applications. They differ in two significant ways: CPLDs because (1) their performance is dependent on the routing that is implemented in the device for a particular application, and (2) their functionality is implemented by lookup tables rather than by PAL-like wide-input AND gates.

Mask-programmable gate arrays are fabricated in a foundry, where final layers of metal customize the wafer to the specifications of the end user. FPGAs are sold as fully fabricated and tested generic products. Their functionality is determined by programming done in the field by the customer and/or end user. FPGAs allow designers to turn a design into working silicon in a matter of minutes, making rapid prototyping a reality.

FPGAs are distinguished on the basis of several features: architecture, number of gates, mechanism for programming, program volatility, the granularity and robustness of a functional/logical unit, physical size (footprint), pinout, time-to-prototype, speed, power, and the availability of internal resources for connectivity [9, 10]. We will focus on the dominant technology: SRAM-based FPGAs, which lose their programming when power is removed from the part.

SRAM-based FPGAs have a fixed architecture that is programmed in the field for a particular application. A typical, basic architecture, as shown in Figure 8-51, consists of (1) an array of programmable functional units (FUs) for implementing combinational...
and sequential logic, (2) a fixed, but programmable, interconnection fabric, which establishes the routing of signals, (3) a configuration memory, which programs the functionality of the device, and (4) I/O resources, which provide an interface between the device and its environment. The performance and density of FPGAs have advanced with improvements in process technology. Today’s leading-edge devices include block memory as well as distributed memory, robust interconnection fabrics, global signals for high-speed synchronous operation, and programmable I/O resources matching a variety of interface standards.

Volatile FPGAs are configured by a program that can be downloaded and stored in static CMOS memory, called a configuration memory. The contents of the static RAM are applied to the control lines of static CMOS transmission gates and other devices to: (1) program the functionality of the functional units, (2) customize configurable features, such as slew rate, (3) establish connectivity between functional units, and (4) configure I/O/bidirectional ports of the device. The configuration program is downloaded to the FPGA from either a host machine or from an on-board PROM. When power is removed from the device the program stored in memory is lost, and the device must be reprogrammed before it can be used again.

The volatility of a stored-program FPGA is a double-edged sword—the FPGA must be reprogrammed in the event that power is disrupted, but the same generic part can serve a boundless variety of applications, and it can be reconfigured on the same circuit board under the control of a processor. One of the programs that can be executed by an FPGA can even test the host system in which it is embedded. The ease of reprogramming a stored-program FPGA supports rapid-prototyping, enabling
design teams to compete effectively in an environment characterized by narrow and ever-shrinking windows of opportunity. Time-to-market is critical in many designs, and FPGAs provide a path to early entry. FPGAs can be reconfigured remotely, via the Internet, allowing designers to repair, enhance, upgrade, or completely reconfigure a device in the field.

### 8.9.1 The Role of FPGAs in the ASIC Market

The architectural resources of FPGAs match the general need for computational engines with memory, datapaths, and processors. The flexibility of an FPGA adds a dimension beyond what is available in masked-programmed devices, because mask-programmed devices cannot be reprogrammed. The same FPGA can be programmed to implement a variety of processors. Expensive, high-risk mask sets for ASICs have made flexibility an important consideration. On the other hand, FPGAs cost more per unit gate, and consume more power.

Table 8-8 summarizes key distinctions between FPGA technology, cell-based and mask-programmed ASIC technology, and standard parts. The myriad applications for ASICs and FPGAs require customization to address the needs of the market. The diversity, rewards, evolving technology, and short lifespan of the market preclude producing and stockpiling standard parts to meet these needs without unacceptably high risk. The lower volumes demanded by individual, specialized applications provides a smaller base over which to amortize the costs of development and production, so units costs for FPGAs are higher than for standard parts, and for high-volume, mask-programmed ASICs, but their NRE costs are significantly lower.

The early technology of MGAs used a fixed array of transistors and routing channels. Routing was a major issue in early devices, and frequently led to incomplete utilization of the available transistors. Today, multilevel metal routing (five and six layers) in a sea-of-gates technology is commonplace, with high utilization of resources. MGAs are preprocessed to the point of customizing the final metal layers to a particular application. The customization/metallization steps connect individual transistors to form gates, and interconnect gates to implement logic. This technology provides a much quicker turnaround than cell-based and full-custom technologies, because only the final metalization step is customized, but not as fast as that for an FPGA. Depending on the foundry, an FPGA can be turned around in a few days to several weeks. On the other hand, designs can be implemented, programmed, and reprogrammed virtually instantaneously in an SRAM-based FPGA while the part is mounted in an emulator or in its target host application. But FPGAs will always be slower and less dense than a comparable MPG because of the additional circuitry and delays introduced by their programmable interconnect.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Functionality</th>
<th>Relative Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Part</td>
<td>Supplier-defined</td>
<td>Low</td>
</tr>
<tr>
<td>FPGA</td>
<td>User-defined</td>
<td>Higher</td>
</tr>
<tr>
<td>ASIC</td>
<td>User-defined</td>
<td>Low</td>
</tr>
</tbody>
</table>
FPGAs are fully tested by the manufacturer before they are shipped, so the designer’s attention is focused on the creativity of the design, not on testing for manufacturing defects. Designers can quickly correct design flaws and reconfigure the part to a different functionality in the field [10]. FPGAs address a market that cannot be met by mask-programmed technologies, which are one-time write. Mask-programmed technologies do not support reconfiguration, and corrections are costly. The risk of an FPGA-based design is significantly higher than for an FPGA because a design flaw requires retooling of the final masks, with attendant costs, and lost time to reenter the fabrication-process queue.

FPGAs have a broad customer base for amortizing the NRE of most of the processing steps compared to cell-based and full-custom solutions. Gate arrays are widely used to implement designs that have a high content of random logic, such as state-machine controllers.

FPGAs require the direct support of a foundry, and the completion of a design can depend on the schedule of the foundry's other customers. FPGAs are fully manufactured and tested in anticipation of being shipped immediately to a buyer.

The software interface between the designer and FPGA technology is simple, and it is now readily and cheaply available on PCs and workstations that support schematic and HDL entry. Programmable logic technology continues to grow in density at exponential rates as compared with other technologies, such as dynamic random-access memories (DRAMs) [10]. The speed of parts is growing at a linear rate and is now at a level that supports system-level integration.

Standard cell-based technology uses a library of predesigned and precharacterized cells that implement gates. The design of the individual cells in a library is labor-intensive, as efforts are made to achieve a dense, area-efficient layout. Consequently, a cell library has a high NRE cost, which a foundry must amortize over a large customer base during the lifetime of the underlying process technology. The mask set of a standard cell library is fully characterized and verified to be correct. Place and route tools select, place and interconnect cells in rows on a chip to implement functionality. The structure is semiregular because the cell heights are fixed, while the width of cells may vary, depending on the functionality being implemented. Placement and routing are customized for each application. Place and route are done automatically to achieve dense configurations that meet speed and area constraints. Cell-based technology requires a fully customized mask set for each application. Consequently, volume must be sufficient to offset high production and development costs and ultimately drive an economically low unit cost.

8.9.2 FPGA Technologies

State-of-the-art FPGAs can now implement the functionality of over a million (two-input equivalent) gates on a single chip. Three basic types of FPGAs are available: antifuse, EPROM, and SRAM-based. The capacity and speed of these parts continues to evolve with process improvements that shrink minimum feature sizes of the underlying transistors.

Antifuse devices\(^\text{19}\) are programmed by applying a relatively high voltage between two nodes to break down a dielectric material. This eliminates the need for a memory

\(^{19}\)See [actel.com](http://www.actel.com) for more information about antifuse devices.
to hold a program, but the one-time write configuration is permanent. When an antifuse is formed a low-resistance path is irreversibly created between the terminals of the device. The antifuse itself is relative small, about the size of a via, and over a million devices can be distributed over a single FPGA [11]. The significant advantage of this technology is that the on-resistance and parasitic capacitance of an antifuse are much smaller than for transmission gates and pass transistors. This supports higher switching speeds and predictable timing delays along routed paths.

EPROM and EEPROM-based technology uses a charged floating gate, programmed by a high voltage. Devices based on these technologies are reprogrammable and nonvolatile and can be programmed offline while imbedded in the target system.

SRAM-based FPGA technology uses CMOS transmission gates to establish interconnect. The status of the gates is determined by the contents of the SRAM configuration memory.

There are multiple vendors of SRAM-based FPGA products (e.g., Xilinx, Altera, ATMEL, and Lucent). The architecture of these FPGAs is similar to that of an FPGA, with block structures of logic and routing channels. Bidirectional and multiply driven wires are included. Devices are advertised on the basis of gate counts, but the actual use of the gates on a device depends on the router’s ability to exploit the resources to support a given design.

The complexity of logic cells in an FPGA functional unit is based on competing factors. If the complexity of a cell is low (fine-grained, such as the Actel Act-1 part), the time and resources required for routing may be high. On the other hand, if the complexity is high, there will be wasted cell area and logic. An example of a fine-grained architecture would be one that is based on two-input NAND gates or muxes, as opposed to a large-grain architecture using four-input NAND gates or muxes. The former uses considerably more routing resources.

8.10 Altera Flex 8000 FPGAs

Altera’s architecture for its FPGAs is illustrated by the now-mature technology of its flexible logic element matrix (Flex) 8000 family of devices, and its closely related, and more advanced, Flex 10 and Flex 20 devices. The basic architecture shown in Figure 8-52 has a grid of LABs, each consisting of eight independently programmable logic elements (LEs) providing an efficient fine-grained logic structure. A logic element has a four-input LUT, a programmable register, and dedicated carry and cascade chains. (The LEs will be discussed in more detail with the Flex 10 device family). The grid of LABs has access to continuous channels of FastTrack Interconnect. The ends of the interconnect channel are connect to input–output elements (IOEs), each having a bidirectional I/O buffer and a flip-flop for registering either the input or the output. Four signals are common to each LE in a LAB: two of which can be used as clocks and two of which can be used for clear/preset control; each of these can be driven by a dedicated input, an I/O pin, or an internal signal from the LAB’s local interconnect. Each device can have up to four low-skew global clock, clear, or preset control signals.

The Flex 8000 family is targeted for applications in register-intensive digital signal processing, wide datapath manipulation and data transformations (bus interfaces,
coprocessor functions, transistor-transistor logic (TTL) integration, and high-speed controllers.

8.11 Altera Flex 10 FPGAs

The push for speed in all electronic equipment has driven designers to relentlessly integrate as much of the design as possible. Consequently, advanced FPGAs now have distributed or block memory on the same chip. PLDs, even when accompanied by a register cell, are not efficient implementations of memory. Consequently, advanced FPGAs combine two structures, one for logic and one that implements embedded memory efficiently. Altera’s Flex 10 devices are intended to support system-on-programmable chip integration in a single device that has logic implementing the equivalent of up to 200,000 typical gates (i.e., two-input NAND gates). The Flex 10 features dual-port on-chip memory and operates at speeds compatible with 33 MHz and 66 MHz IEEE PCI local bus specifications. Table 8-9 identifies some significant physical features of the device family\(^\text{20}\) over a range of low- and high-density of gates.

The Flex 10 architecture shown in Figure 8-53 is similar to that of the Flex 8000 devices shown in Figure 8-52, but the Flex 10 family includes block RAM. The architecture

\(^{20}\text{The Flex 10 family is available in 5, 3.3, and 2.5 V technology. Additional features (e.g., performance and packaging) of the Flex 10 device family are described in data sheets available at www.altera.com.}\)
TABLE 8-9  Altera Flex 10K FPGA: significant architectural features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>EPF10K10</th>
<th>***</th>
<th>EPF10K250A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical gates</td>
<td>10,000</td>
<td>250,000</td>
<td></td>
</tr>
<tr>
<td>Max # system gates</td>
<td>31,000</td>
<td>310,000</td>
<td></td>
</tr>
<tr>
<td># Logic elements (LEs)</td>
<td>576</td>
<td>12,160</td>
<td></td>
</tr>
<tr>
<td>Logic Array Blocks (LAB)</td>
<td>72</td>
<td>1520</td>
<td></td>
</tr>
<tr>
<td>Embedded Array Blocks (EAB)</td>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total RAM bits</td>
<td>6,144</td>
<td>40,960</td>
<td></td>
</tr>
<tr>
<td>Max # user I/O pins</td>
<td>150</td>
<td>470</td>
<td></td>
</tr>
</tbody>
</table>

* Logic and RAM, excluding JTAG circuitry.

FIGURE 8-53  Altera Flex 10 CPLD: top-level architecture.
consists of an array of embedded array blocks (EABs), an array of LABs, and an interconnection fabric. Several LABs and an EAB occupy a row; each LAB contains eight logic elements and local interconnect, as in the architecture of the Flex 8000. Each EAB has 2K of RAM memory. FastTrack Interconnect channels connect the LABs and EABs and provide global interconnect without making use of switch matrixes, with the result that delays do not depend on the routing. The IOEs can be programmed to serve as input, output or bidirectional signal paths, with programmable slew rate to reduce switching noise (only time-critical paths need a high skew rate—the others can be programmed for low-slew operation to reduce noise).

The internal architecture of a LAB is shown in Figure 8-54. Each LE has four inputs from the local interconnect (which may include bits that have been fed back from the outputs of the logic elements in the LAB (one bit can be fed back from each of the eight logic elements). Each logic element also has four control signals that are programmed to be formed locally or obtained globally. The outputs of a LAB can be connected to the row and column interconnect.

The LE of the Flex 10 family is shown in Figure 8-55. Each element contains an LUT that can implement any function of four variables. The output of the element can be direct or registered. With four bits of data input each, two LEs can implement a full
adder (one for the sum bit and one for the carry out bit), with the carry-in and carry-out bits linked to those of adjacent LEs.

Dedicated pins support carry-in/out and cascade-in/out connectivity for wide datapaths and functions of more than four variables. The cascade chain can be programmed to AND the outputs of logic elements to form a function of more than four variables, or to OR the outputs, which forms a sum of cubes, each of which can be a function of up to four variables (Figure 8-56).

Each row of the Flex 10 device has an embedded memory array block, referred to as an EAB (Figure 8-57). The memory (2,048 bits) within an EAB can be programmed to various configurations as RAM or ROM. The EAB local interconnect serves as a local datapath for both the address and input data of the memory. The output of the memory is passed directly to either a row or column interconnect, directly or through a register. Complex functions can be implemented in an EAB in one logic level, avoiding the additional routing delays that would result from implementing the function in a series of chained LEs or by aggregating several units of distributed RAM in an FPGA.

The configuration of a logic array and an embedded memory array in the Flex 10 device family combines the flexibility of a PLD with the density and efficiency of an embedded gate array. The more recent and advanced Flex 10E family of devices shown in Table 8-10 has significantly greater memory capacity. Its EAB blocks have independently configurable dual-port capability, enabling them to support dual-clock applications, such as dual-clock FIFOs.
FIGURE 8-56  Cascaded chains in the Altera Flex 10 CPLD: (a) an AND chain, and (b) an OR chain.

FIGURE 8-57  Altera Flex 10 FPGA with embedded RAM/ROM block.
TABLE 8-10 Altera Flex 10KE FPGA: significant architectural features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>EPF10K30E</th>
<th>EPF10K50E</th>
<th>EPF10K100B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical gates (1)</td>
<td>30,000</td>
<td>50,000</td>
<td>100,000</td>
</tr>
<tr>
<td>Maximum no. of system gates</td>
<td>119,000</td>
<td>199,000</td>
<td>158,000</td>
</tr>
<tr>
<td>No. of logic elements</td>
<td>1,728</td>
<td>2,880</td>
<td>4,992</td>
</tr>
<tr>
<td>EABs</td>
<td>6</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Total RAM bits</td>
<td>24,576</td>
<td>40,960</td>
<td>24,576</td>
</tr>
<tr>
<td>Maximum no. of user I/O pins</td>
<td>220</td>
<td>254</td>
<td>191</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>EPF10K100E</th>
<th>EPF10K130E</th>
<th>EPF10K200S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical gates (1)</td>
<td>100,000</td>
<td>130,000</td>
<td>200,000</td>
</tr>
<tr>
<td>Maximum no. of system gates</td>
<td>257,000</td>
<td>342,000</td>
<td>513,000</td>
</tr>
<tr>
<td>No. of logic elements</td>
<td>4,992</td>
<td>6,656</td>
<td>9,984</td>
</tr>
<tr>
<td>EABs</td>
<td>12</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Total RAM bits</td>
<td>49,152</td>
<td>65,536</td>
<td>98,304</td>
</tr>
<tr>
<td>Maximum no. of user I/O pins</td>
<td>338</td>
<td>413</td>
<td>470</td>
</tr>
</tbody>
</table>

8.12 Altera Apex FPGAs

Advances in process technology have led to denser and faster field-programmable devices. The complexity of applications and the demand for higher levels of integration has produced leading edge devices with on-chip block memory, support for high-speed synchronous operation (phase lock loops [PLLs]), and support for a wide assortment of interfaces to the external environment. The high-density Altera APEX family of Altera FPGAs combines four architectural elements to support complete system-level integration on a single chip. The arrangement partitions an implementation into control and datapath functionality and associates functionality with different architectural elements. The structure in Figure 8-58 has LUTs to support register-intensive datapath and digital signal processing (DSP) functions, and product-term integrators to support high-speed complex (multilevel) combinational logic in control logic and state machines. Embedded memory blocks, referred to as ESBs (embedded system blocks) can support a variety of memory functions (e.g., FIFOs, dual-port RAMs, and content-addressable memory).

The architecture is organized as a series of MegaLABs, shown in Figure 8-59, each consisting of from 16 to 24 LABs, an ESB, and the Flex 8000 architecture's channel-based FastTrack Interconnect fabric.

Significant architectural features of the APEX family are shown in Table 8-11. The I/O elements of the device are located at the end of each row and column of the interconnect. Each element contains a bidirectional I/O buffer, and a register. Global clocks support high-speed registered data transfers. Four dedicated input pins and two
FIGURE 8-58 Altera architecture: Apex 20KE FPGA.

FIGURE 8-59 Altera architecture: MegaLAB structure.

TABLE 8-11 Altera APEX 10KE CPLD: significant architectural features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>EP20K30E</th>
<th>***</th>
<th>EP20K1500E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical gates</td>
<td>30,000</td>
<td></td>
<td>1,500,000</td>
</tr>
<tr>
<td>Maximum no. of system gates</td>
<td>113,000</td>
<td></td>
<td>2,392,000</td>
</tr>
<tr>
<td>No. of logic elements (LEs)</td>
<td>1,200</td>
<td></td>
<td>51,840</td>
</tr>
<tr>
<td>Embedded system blocks</td>
<td>12</td>
<td></td>
<td>216</td>
</tr>
<tr>
<td>Maximum RAM bits</td>
<td>53,248</td>
<td></td>
<td>442,368</td>
</tr>
<tr>
<td>Maximum macrocells</td>
<td>192</td>
<td></td>
<td>3,456</td>
</tr>
<tr>
<td>Maximum no. of user I/O pins</td>
<td>246</td>
<td></td>
<td>808</td>
</tr>
</tbody>
</table>
dedicated clock pins on each device provide high-speed, low-skew control signals. The four dedicated inputs drive four global signals, which can be used in synchronous control operations.

8.13 Altera Chip Programmability

Altera’s Max+PLUS II development system provides a turnkey, desktop environment supporting rapid prototyping of FPGA-based designs. The system supports schematic and HDL-based (Verilog, VHDL, AHDL) design entry, logic synthesis, simulation, timing analysis, and device programming.

8.14 XILINX XC4000 Series FPGA

Xilinx launched the world’s first commercial FPGA in 1985, with the vintage XC2000 device family. The XC3000 and XC4000 families soon followed, setting the stage for today’s Spartan and Virtex device families. Each evolution of devices brought improvements in density, performance, voltage levels, pin counts, and functionality. The XC4000, Spartan, and Spartan/XL devices have the same basic architecture; Table 8-12 displays the evolution of density and operating voltage among three product families.

8.14.1 Basic Architecture

The basic architecture of the XC3000, XC4000, Spartan, and Spartan/XL device family consists of an array of configurable logic blocks (CLBs), a variety of local and global routing resources, and IOBs, programmable I/O buffers, and a SRAM-based configuration memory, as shown in Figure 8-60.

8.14.2 XC4000 Configurable Logic Block

Each CLB consists of an LUT, multiplexers, registers, and paths for control signals. Each CLB of the Xilinx 3000 series had a single programmable LUT implementing two combinational logic functions of five inputs (5-ns CLB delay). The outputs of the functions could be programmed to pass through a register or pass directly to the output of the CLB. A registered output of a function could also be routed internally to the input of the LUT, allowing a group of CLBs to implement register-rich logic for a state machine or a data pipeline.

The CLBs of the successor to the XC3000, the XC4000 device family, are more versatile than those of the XC3000. In the architecture shown in Figure 8-61, note that each CLB contains three function generators (F, G, and H). Each is based on an LUT with 5-ns delay independent of the function begin implemented. Two of the function

21The flip-flops can be set individually or globally.
TABLE 8-12  Migration of density and operating voltage for Xilinx part families.

<table>
<thead>
<tr>
<th>Part</th>
<th>Maximum No. of System Gates</th>
<th>System Performance</th>
<th>Operating Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan</td>
<td>40K</td>
<td>&gt;80 MHz</td>
<td>5V</td>
</tr>
<tr>
<td>Spartan/XL</td>
<td>40K</td>
<td></td>
<td>3.3</td>
</tr>
<tr>
<td>Spartan-II</td>
<td>200K</td>
<td>&lt;200 MHz</td>
<td>2.5</td>
</tr>
</tbody>
</table>

generators (F and G) can generate any arbitrary function of four inputs, and the third (H) can generate any Boolean function of three inputs. The H-function block can get its inputs from the F and G LUTs, or from external inputs. The three function generators can be programmed to generate: (1) three different functions of three independent sets of variables (one with four inputs and one with three inputs—one function must be registered within the CLB), (2) an arbitrary function of five variables, (3) an arbitrary function of four variables together with some functions of six variables, and (4) some functions of nine variables.

FIGURE 8-60  Basic architecture of Xilinx 3000, 4000, Spartan, and Spartan X/L devices.
Each CLB in the XC4000 series has two storage devices that can be configured as edge-triggered flip-flops with a common clock, or in the XC4000X they can be configured as flip-flops or as transparent latches with a common clock (programmed for either edge and separately invertible) and enable. The storage elements can get their inputs from the function generators, or from the \( D_n \) input. The other element can get an external input from the \( HI \) input. The function generators can also drive two outputs directly (\( X \) and \( Y \)) and independently of the outputs of the storage elements. All of these outputs can be connected to the interconnect network. The storage elements are driven by a global S-R during power-up; the global S-R is programmed to match the programming of the local S-R control for a given storage element.

### 8.14.3 Dedicated Fast Carry and Borrow Logic

The F and G function generators of the XC4000 family have separate dedicated logic for fast carry and borrow generation, with dedicated routing to link the extra signal to the function generator in the adjacent CLB. The prebuilt carry chain within a CLB can be used to add a pair of 2-bit words in one CLB. One function generator (F) can be used to generate \( a0 + b0 \), and a second function generator (G) can generate \( a1 + b1 \). The fast carry will forward the carry to the next CLB above or below. This feature is implemented with a hard macro and the graphical editor. Fast carry and borrow logic increases the efficiency performance of adders, subtractors, accumulators, comparators, and counters.
8.14.4 Distributed RAM

The three function generators within a CLB can be used as RAM, either a $16 \times 2$ dual port RAM or a $32 \times 1$ single-port RAM. The XC4000 devices do not have block RAM, but a group of their CLBs can form an array of memory.

8.14.5 XC4000 Interconnect Resources

The XC4000 series was designed with interconnect resources to minimize the resistance and capacitance of an average routed path. A grid of switch matrixes overlays the architecture of CLBs to provide general-purpose interconnect for branching and routing throughout the device. The interconnect has three types of general-purpose interconnect: single-length lines, double-length lines, and long lines. A grid of horizontal and vertical single-length lines connect an array of switch boxes. The boxes provide a reduced number of connections between signal paths within each box, not a full crossbar switch. In the XC4000 there is a rich set of connections between single-length lines and the CLB inputs and outputs. These provide capability for nearest-neighbor and across-the-chip connection between CLBs. Each CLB has a pair of three-state buffers that can drive signals onto the nearest horizontal lines above or below the CLB.

Direct (dedicated) interconnect lines provide routing between adjacent vertical and horizontal CLBs in the same column or row. These are relatively high-speed local connections through metal, but are not as fast as a hard-wired metal connection because of the delay incurred by routing the signal paths through the transmission gates that configure the path. Direct interconnect lines do not use the switch matrixes, which eliminates the delay incurred on paths going through a matrix.  

Double-length lines traverse the distance of two CLBs before entering a switch matrix, skipping every other CLB. These lines provide a more efficient implementation of intermediate-length connections by eliminating a switch matrix from the path, thereby reducing the delay of the path.

Long lines span the entire array vertically and horizontally. They drive low-skew, high-fanout control signals. Long vertical lines have a programmable splitter that segments the line and allows two independent routing channels spanning half of the array, but located in the same column. The routing resources are exploited automatically by the routing software. There are eight low-skew global buffers for clock distribution, and the skew on a global net is less than 2 ns. The XC4000 device also has an internally generated clock.

The signals that drive long lines are buffered. Long lines can be driven by adjacent CLBs or IOBs and may connect to three-state buffers that are available to CLBs. Long lines provide three-state busses within the architecture, and implement wired-AND logic. Each horizontal long line is driven by a three-state buffer, and can be programmed to connect to a pull-up resistor, which pulls the line to a logical 1 if no driver is asserted on the line.

---

22See Xilinx documentation for the pinout conventions to establish local interconnect between CLBs.
Figure 8-62 illustrates the single- and double-length interconnect lines of the XC4000 device.

The connectivity of a CLB with neighboring switch matrixes is shown in Figure 8-63.

The programmable interconnect resources of the device connect CLBs and IOBs, either directly or through switch boxes. These resources consist of a grid of two layers of metal segments and programmable interconnect points (PIP) within switch boxes. A PIP is a CMOS transmission gate whose state (on or off) is determined by the content of a static RAM cell in the programmable memory, as shown in Figure 8-64. The connection is established when the transmission gate is on (i.e., a 1 is applied at the gate of the $n$-channel transistor and a 0 is applied at the gate of the $p$-channel transistor). The interconnect path is established without altering the physical medium, as happens in a fuse-type interconnect technology. Thus, the device can be reprogrammed by simply changing the content of the controlling memory cell.

The architecture of a PIP-based interconnection in a switch box is shown in Figure 8-65. The configuration of CMOS transmission gates determines the connection between a horizontal line and the opposite horizontal line, and the vertical lines at the connection. Each switch matrix PIP requires six pass transistors to establish full connectivity.

8.14.6 XC4000 I/O Block (IOB)

Each programmable I/O pin/pin of an XC4000 device has a programmable IOB with buffers for compatibility with TTL and CMOS signal levels. Figure 8-66 shows a simplified
FIGURE 8-63 Connectivity of a CLB with its neighboring switch matrices.

FIGURE 8-64 RAM cell controlling a PIP transmission gate.
FIGURE 8-65  Circuit-level architecture of a PIP within a switch box.

FIGURE 8-66  XC4000 series IOB.
schematic for a XC4000 programmable IOB. It can be used as an input, output or bidirectional port. An IOB that is configured as an input can have direct, latched, or registered input. In an output configuration, the IOB has direct or registered output. The output buffer of an IOB has skew and slew control. The registers available to the input and output path of an IOB are driven by separate, invertible clocks. There is a global set and reset.

The XC4000 architecture has delay elements that compensate for the delay induced when a clock signal passes through a global buffer before reaching an IOB. This eliminates the hold condition on the data at an external pin. The three-state output of an IOB puts the output buffer in a high-impedance state. The output and the enable for the output can be inverted. The slew rate of the output buffer can be controlled to minimize transients on the power bus when noncritical signals are switched. The IOB pin can be programmed for pull-up or pull-down to prevent needless power consumption and noise.

The XC4000 has four edge decoders on each side of the chip. An edge decoder can accept up to 40 inputs from adjacent IOBs and 20 inputs from on-chip. These decoders provide fast decoding of wide address paths. Multiple CLBs might be needed to support decoding when the fan-in exceeds the width of a single CLB.

The XC4000 devices have imbedded logic to support the IEEE 1149.1 (JTAG) boundary scan standard. There is an on-chip test access port (TAP) controller, and the I/O cells can be configured as a shift register. Under test, the device can be checked to verify that all the pins on a PC board are properly connected by creating a serial chain of all of the I/O pins of the chips on the board. A master three-state control signal puts all of the IOBs in high-impedance mode for board testing.

The XC4000EX series, introduced in 1996, provides a twofold increase in routing resources over the earlier members of the XC4000 family. An additional 22 vertical lines are available in each column of CLBs. Another 12 quad lines have been added to each row and column to support fast global routing, illustrated in Figure 8-67. The IOBs have a dedicated early clock and fast-capture latch. The registers have a 4-ns setup time and a 6-ns clock-to-output propagation delay.

### 8.14.7 Enhancements in the XC4000E and XC4000X Series

The XC4000E and XC4000X devices offer significant increases in speed and capacity, as well as architectural improvements over the basic XC4000 family. Devices in these families can run at system clock rates up to 80 MHz, and internal frequencies can reach 150 MHz, as a result of submicron multilayered metal processes. The XC4000XL (0.35 micron feature size) operates at 3.3 V with a system frequency of 80 MHz.

### 8.14.8 Enhancements in the Spartan Series

The Spartan series of Xilinx FPGAs has the same architecture of CLBs and IOBs as its XC4000 predecessors, but has higher performance and is targeted for high-volume, low-cost applications requiring on-chip memory and high performance (over 100 MHz system speeds).

Spartan chips can accommodate embedded soft cores, and their on-chip distributed, dual-port, synchronous RAM (SelectRAM) can be used to implement FIFOs,
shift registers, and scratchpad memories. The blocks can be cascaded to any width and depth and located anywhere in the part, but their use reduces the CLBs available for logic. Figure 8.68 displays the structure of the on-chip RAM that is formed by programming a LUT to implement a single-port RAM with synchronous write and asynchronous read. Each CLB can be programmed as a $16 \times 2$ or a $32 \times 1$ memory.

Dual-port RAMs are emulated in a Spartan device by the structure shown in Figure 8.69, which has a single (common) write port and two asynchronous read ports. A CLB can form a memory with a maximum size of $16 \times 1$.
8.15 XILINX Spartan XL FPGAs

Spartan XL chips are a further enhancement of the Spartan chips, offering higher speed and density (40,000 system gates per approximately 6000 usable gates), and on-chip, distributed SelectRAM memory. The LUTs of the devices can implement 2^2 different functions of n inputs. The devices are peripheral component interconnect (PCI) bus compliant and have eight flexible global low-skew buffers (BUFGLS) and CLB latches for clock distribution and for secondary control signals; each serves one-fourth of the devices. The input latches support fast capture of data.

2The maximum number of logic gates for a Xilinx FPGA is an estimate of the maximum number of logic gates that could be realized in a design consisting of only logic functions (no memory). Logic capacity is expressed in terms of the number of two-input NAND gates that would be required to implement the same number and type of logic functions (Xilinx application note).
The XL series is targeted for applications in which low cost, low power, low packaging, and low test cost are important factors constraining the design solution. Spartan XL devices offer up to 80-MHz system performance, depending on the number of cascaded LUTs, which reduces performance by introducing longer paths. Table 8-13 presents significant attributes of devices in the Spartan XL family.

The architecture of the 3000, 4000, Spartan, and Spartan/XL series of Xilinx FPGAs consists of an array of CLB tiles mingled within an array of switch matrixes, surrounded by a perimeter of IOBs. These devices supported only distributed memory, whose use reduces the number of CLBs that could be used for logic. The relatively small amount of on-chip memory limits these devices to applications in which operations with off-chip memory devices do not compromise performance objectives. Beginning with the Spartan II series, Xilinx supported configurable embedded block memory as well as distributed memory in a new architecture.

<table>
<thead>
<tr>
<th>Spartan/XL</th>
<th>XCS05/XL</th>
<th>XCS10/XL</th>
<th>XCS20/XL</th>
<th>XCS30/XL</th>
<th>XCS40/XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Cells&lt;sup&gt;2&lt;/sup&gt;</td>
<td>238</td>
<td>466</td>
<td>950</td>
<td>1,368</td>
<td>1,862</td>
</tr>
<tr>
<td>Maximum No. of Logic Gates</td>
<td>3,000</td>
<td>5,000</td>
<td>10,000</td>
<td>13,000</td>
<td>20,000</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>360</td>
<td>616</td>
<td>1,120</td>
<td>1,536</td>
<td>2,016</td>
</tr>
<tr>
<td>Maximum RAM Bits</td>
<td>3,200</td>
<td>6,272</td>
<td>12,800</td>
<td>18,432</td>
<td>25,088</td>
</tr>
<tr>
<td>Maximum Available I/O</td>
<td>77</td>
<td>112</td>
<td>160</td>
<td>192</td>
<td>224</td>
</tr>
</tbody>
</table>

<sup>1</sup> 20–30% of CLBs as RAM
<sup>2</sup> 1 Logic cell = four-input LUT + flip-flop

### 8.16 XILINX Spartan II FPGAs

Aside from improvements in speed (200 MHz I/O switching frequency), density (up to 200,000 system gates), and operating voltage (2.5 V), four other features distinguish the Spartan II devices from the Spartan devices: (1) on-chip block memory, (2) a novel architecture, (3) support for multiple I/O standards, and (4) delay-locked loops<sup>24</sup>.

The Spartan II device family, manufactured in 0.22/0.18 μm CMOS technology with six layers of metal for interconnect, incorporates configurable block memory in addition to the distributed memory of the previous generations of devices, and the block memory does not reduce the amount of logic and/or distributed memory that is available for the application. The availability of a large on-chip memory can improve system performance by eliminating or reducing the need to access off-chip storage.

Reliable clock distribution is the key to the synchronous operation of high-speed digital circuits. If the clock signal arrives at different times at different parts of a circuit, the device may fail to operate correctly. Clock skew reduces the available time budget.

---

<sup>24</sup>The Spartan-II devices do not support LVDS (low voltage differential signaling) and LVPECL (low voltage/power emitter-coupled logic) I/O standards.
of a circuit by lengthening the setup time at registers. It can also shorten the effective hold time margin of a flip-flop in a shift register and cause the register to shift incorrectly. At high clock frequencies, the effect of skew is more significant because it represents a larger fraction of the clock cycle time. Buffered clock trees are commonly used to minimize clock skew in FPGAs. Xilinx provides all-digital delay-locked loops (DLLs) for clock synchronization/management in high-speed circuits. DLLs eliminate the clock distribution delay and provide frequency multipliers, frequency dividers, and clock mirrors.

Spartan-II devices are suitable for applications such as implementation of the glue logic of a video capture system, and the glue logic of an ISDN modem. Device attributes are summarized in Table 8-14, and the evolution of technology in the Spartan series is evident in the data in Table 8-15.

The top-level tiled architecture of the Spartan-II device, shown in Figure 8-70, marks a new organization of the Xilinx parts. Each of four quadrants of CLBs is supported by a DLL and is flanked by a 4096-bit block of RAM, and the periphery of the chip is lined with IOBs.

Each CLB contains four logic cells, organized as a pair of slices. Each logic cell, shown in Figure 8-71, has a four-input LUT, logic for carry and control, and a D-type flip-flop. The CLB contains additional logic for configuring functions of five or six inputs.

### Table 8-14: Spartan II device attributes.

<table>
<thead>
<tr>
<th>Spartan-II FPGAs</th>
<th>XC2S15</th>
<th>XC2S30</th>
<th>XC2S50</th>
<th>XC2S100</th>
<th>XC2S150</th>
<th>XC2S200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Cells ²</td>
<td>432</td>
<td>972</td>
<td>1,728</td>
<td>2,700</td>
<td>3,888</td>
<td>5,292</td>
</tr>
<tr>
<td>Block RAM Bits</td>
<td>16,384</td>
<td>24,576</td>
<td>32,768</td>
<td>40,960</td>
<td>49,152</td>
<td>57,344</td>
</tr>
<tr>
<td>Maximum Available I/O</td>
<td>86</td>
<td>132</td>
<td>176</td>
<td>196</td>
<td>260</td>
<td>284</td>
</tr>
</tbody>
</table>

¹20-30% of CLBs as RAM
²1 Logic cell = four-input LUT + flip-flop

### Table 8-15: Comparison of the Spartan device families.

<table>
<thead>
<tr>
<th>Part</th>
<th>Spartan</th>
<th>Spartan/XL</th>
<th>Spartan-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>XC4000 Based</td>
<td>XC4000 Based</td>
<td>Virtex Based</td>
</tr>
<tr>
<td>Maximum No. of System Gates</td>
<td>5K-40K</td>
<td>5K-40K</td>
<td>15K-200K</td>
</tr>
<tr>
<td>Memory</td>
<td>Distributed RAM</td>
<td>Distributed RAM</td>
<td>Block + Distributed</td>
</tr>
<tr>
<td>I/O Performance</td>
<td>80 MHz</td>
<td>100 MHz</td>
<td>200 MHz</td>
</tr>
<tr>
<td>I/O Standards</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Core Voltage</td>
<td>5V</td>
<td>3.3</td>
<td>2.5</td>
</tr>
<tr>
<td>DLLs</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

²Parts are available with up to 14 blocks (56K bits).
The Spartan-II part family provides the flexibility and capacity of an on-chip block RAM; in addition, each LUT can be configured as a $16 \times 1$ RAM (distributed), and the pair of LUTs in a logic cell can be configured as a $16 \times 2$ bit RAM or a $32 \times 1$ bit RAM.

The IOBs of the Spartan-II family are individually programmable to support the reference, output voltage, and termination voltages of a variety of high-speed memory and bus standards (see Figure 8-72). Each IOB has three registers, which can function as D-type flip-flops or as level-sensitive latches. One register (TFF) can be used to register the signal that controls (synchronously) the programmable output buffer. A second (OFF) can be programmed to register a signal from the internal logic (alternatively, a signal from the internal logic can pass directly to the output buffer). The third device can register the signal coming from the I/O pad (alternatively, this signal can pass directly to the internal logic. A common clock drives each register, but each has an independent clock enable. A programmable delay element on the input path can be used to eliminate the pad-to-pad hold time.
FIGURE 8-71 Spartan II CLB slice (two per CLB).
8.17 XILINX Virtex FPGAs

The Virtex device series is the leading edge of Xilinx technology. It addresses four key factors that influence the solution to complex system-level and system-on-chip (SoC) designs: (1) the level of integration, (2) the amount of embedded memory, (3) performance (timing), and (4) subsystem interfaces. Process rules for leading-edge Virtex parts stand at 0.18 μm and will continue to shrink. The rules allow up to 138,240 logic cells to be packed into a single die, providing up to 10 million system gates, and the capacity to support memory-intensive (up to 3.5 M bits) system-level applications requiring high density and high performance.

The Virtex family incorporates physical (electrical) and protocol support for 20 different I/O standards, including LVDS and LVPECL, with individually programmable pins. Up to 12 digital clock managers provide support for frequency synthesis and phase shifting in synchronous applications that require multiple clock domains and high frequency I/O. The Virtex architecture is shown in Figure 8-73, and its IOB is shown in Figure 8-74.
FIGURE 8-73 Xilinx Virtex-II overall architecture.

FIGURE 8-74 Xilinx Virtex-II IOB Block.
8.18 Embeddable and Programmable IP Cores for a System on a Chip (SoC)

ASIC cores consist of intellectual property (IP) that has been designed, verified, and marketed by a vendor for re-use by other parties. Cores may be soft (software models) or hard (mask sets). The use of preimplemented and verified embedded cores in an ASIC can shorten the time-to-market of a new product by reducing the amount of circuitry that must be developed. Whether this economy is realized depends on the reliability and documentation of the embedded logic and whether system-level tools exist for integrating and testing the embedded part.

FPGA vendors have expanded their efforts beyond devices and design tools, and also provide an assortment of cores that can be embedded in a device to simplify the designer's task. For example, Xilinx offers, either directly or through partnerships with third parties, cores for basic elements (e.g., accumulators and shift registers), math functions (e.g., multipliers, multiply-and-accumulate [MAC] units, and dividers), memories (e.g., synchronous FIFO), standard bus interfaces, (e.g., PCI), processor peripherals (e.g., interrupt controller), universal asynchronous receiver and transmitters (UARTs), and a variety of networking and communication products (e.g., protocol cores). Special design kits may be required to exploit these available resources. Vendors also provide reference designs illustrating how to exploit embeddable cores.

ASIC designs are characterized by high performance, high NRE cost, and high risk. The risks are high because the cost of a mask set now ranges between $250K and $500K. A mask error and consequent re-spins of a design are prohibitive from the standpoint of cost and lost opportunity to capture market share. Embeddable programmable cores\textsuperscript{26} offer flexibility (the design can be modified), lower NRE cost, and lower risk. These hybrid devices are targeted at applications for which standards are evolving or for which NRE costs might have to be amortized over multiple variants of a product. For example, the control logic of a multiprocessor computer in a wireless network for image processing applications could be implemented in a programmable core, allowing the design to be modified to meet a dynamic marketplace. Multiple designs can be produced from a single die. Two aspects are involved here: the compatibility of the processes for manufacturing ASICs and FPGAs, and the IP that is ultimately configuring the FPGA for a specific application. The former will develop and set the stage for proliferation of the latter.\textsuperscript{27} There are two variations on the theme: embeddable programmable cores for placement in an ASIC (Actel and Adaptive Silicon) and embeddable complex ASIC cores for placement within an FPGA (Triscend, Xilinx, Lucent, Altera, Atmel, QuickLogic).

Compared to FPGAs, ASICs are relatively expensive to design and manufacture. They gain performance at the expense of flexibility. An emerging technology is that of

\textsuperscript{26}See the web links for Embeddable Programmable Cores.
\textsuperscript{27}The Virtual Socket Interface Alliance (VSIA) is an industry group that promotes technical standards for mixing and matching IP from multiple sources.
embedding an FPGA within an ASIC to gain flexibility, reduce the risk of a design, and extend the life of a design to a wider range of applications. Other programmable architectures are emerging as well. For example, Adaptive Silicon has developed a basic building block, called a Hex block, consisting of sixty-four 4-bit ALUs. Hex blocks can be tiled in rectangular patterns within a fabric of local and global interconnect. A $4 \times 4$ array of hex blocks supporting arithmetic functions will achieve a density of approximately 25,000 ASIC gates.

### 8.19 Verilog-Based Design Flows for FPGAs

The design flow for an FPGA-based target technology is shown in Figure 8-75. It relies heavily on bundled software to accomplish the synthesis, implementation, and downloading of the design into a part. The place-and-route step that plays such a dominant role in ASICs is not shown in the design flow because it is transparent to the user. Likewise, the extraction of parasitics is not shown because the fixed architecture of the devices allow their timing to be precharacterized to serve a database within the implementation tool. The simplified flow allows a designer to create design iterations and derivative designs rapidly, ultimately producing a hardware prototype.

The objective of rapid prototyping is to create a working prototype as quickly as possible to meet market conditions and to support broader testing in the host environment. Initially, the tools supporting FPGAs relied on schematic entry, but many vendors are now placing greater emphasis on supporting hardware description languages (HDLs). For example, the recently released Xilinx ISE (Integrated Synthesis Environment) tools are tailored for HDL-based entry, and support floor planning, simulation, automatic block placement and routing of interconnects, timing verification, downloading of configuration data, and readback of the configuration bit stream. The tools ultimately produce the bit-stream file that can be downloaded to the part to configure it on the host board.

### 8.20 Synthesis with FPGAs

In Chapter 6 we discussed the importance of adopting synthesis-friendly descriptive styles. A model has restricted utility if it cannot be synthesized. In addition, the models must include features that allow them to exploit the unique features of the target architecture. For example, FPGA tools must optimize the partition of memory between distributed and block memory resources. It is especially important in DSP applications that the synthesis tool minimize the use of off-chip memory in order to maximize performance.

---

26 Portions of the design that are risky and might require future change can be placed in the FPGA.

27 LSI Logic and Adaptive Silicon have been working to embed an SRAM-based FPGA in an LSI ASIC.
FPGA vendors provide libraries of macros that implement specific functionality. The designer has a choice between a packaged macro and the circuit that a synthesis engine infers from a behavioral description. If technology-specific cores are used, they should be isolated within the design's hierarchy.

FPGAs are rich in registers, so it is generally advisable to employ one-hot coding of the state of a finite-state machine. There is usually little or no gain in trying to reduce the number of CLBs by employing a sequential binary code, because additional cells will be required to form the more complicated combinational logic that results from such a scheme.

The state decoding of a FSM must cover all possible codes of the state. Otherwise, latches will be introduced into the design.\textsuperscript{30} This practice also protects against the machine entering a state from which it cannot recover. It is recommended that the designer assign the default state explicitly rather than use a tool option to do it.

\textsuperscript{30}Synthesis tools will produce reports describing device use, including a report on the number of latches and registers in the implementation. It is a good practice to review these reports to detect unwanted latches.
automatically, if for no other reason than to encourage more thoughtful consideration about the design. (Use the Verilog keyword `default` as the item decoded in a `case` statement.) Also, as discussed in Chapter 6, it is recommended that all of the register variables that are assigned value within a level-sensitive cyclic behavior be initialized at the beginning of the listed code and then assigned value within the behavior by exception—as a way to help prevent synthesis of unwanted latches in the design.

The registers in a CLB do not power up to a specific state, so it is essential that a reset or set signal port be included at the top-level module of the design and be used to drive the machine into a known state. Synchronous resets are used to minimize the possibility of a reset signal causing a metastable condition.

Decoding logic should be implemented with `case` statements rather than `if . . . then . . . else` statements, unless a priority structure is intended. The former produces parallel logic (faster); the latter tends to produce logic that is nested (e.g., priority decoder) and will be cascaded in a multilevel series structure of LUTs, resulting in a slower circuit.

A net that fans out from a flip-flop to several points in a circuit can be slow and difficult to route; it might ultimately be the source of a timing constraint violation. This problem may be solved by duplicating the flip flop so that the fan-out can be shared. The result will be that the routing step takes less time and is more likely to complete successfully, and the overall performance will be improved. The tradeoff is that the solution occupies a larger area of the chip (more CLBs are required). Candidates for this treatment are the address and control lines of large memory arrays, clock enable lines, output enable lines, and synchronous reset signals. If the driver of a high-fan-out net is asynchronous, synchronize the signal before duplicating it.31

The throughput of a design can be improved by partitioning combinational logic systematically and inserting registers at the interface between the partitions. For example, a 16-bit adder can be partitioned into two 8-bit adders and pipelined to reduce the delay of the carry chain by a factor of 2. Pipelining shortens the path that a given signal must travel during a clock cycle. Consequently, the clock can be run faster and timing violations can be eliminated. The tradeoff, which could be unacceptable, is that the pipelined datapath has latency, because the data will take one or more additional clock cycles to propagate through the circuit, depending on the number of pipeline stages that have been added.32 The second tradeoff is that the pipeline registers occupy CLBs. Thus, the physical part that implements the design must be large enough to supply the additional registers for the pipeline. Reports produced by the software indicate the use of CLBs, so they should be consulted before attempting to increase the clock speed or eliminate a timing violation by pipelining.

If the place-and-route engine within the tool is allowed complete freedom it will generate an optimal assignment of pins. This freedom is curtailed when the part must fit into the socket of a previously configured board. Ideally, the board is not configured

---

31Be aware that the Xilinx tools automatically map into the same CLB signals that end with the same numeric suffix, (e.g., `sig_1`, `sig_2`). Naming duplicated signals in this manner contradicts the effort to distribute the duplicated signals to different regions of the chip. Instead, use alphabetical labels (e.g., `a`, `b`) to compose the suffix of duplicated signals.

32Pipelining will be considered in more detail in Chapter 9.
until the FPGA has been fully designed. Constraining the pinout constrains the optimization process, and may sacrifice performance. If feasible, careful pin assignment can lead to improved routing of the design. For example, the horizontal long lines in the XC4000 based Xilinx architectures have three-state buffers, which makes them suitable for data busses. On the other hand, vertical long lines for clock enables and vertical carry chains lead naturally to a vertical orientation of the cells of registers and counters. These architectural features suggest that datapaths should be applied to the left and right sides of the part, and control lines should be applied to the top and bottom of the part when manual routing and pin assignment are necessary. The tool has maximum flexibility when no pins are preassigned, but environmental constraints may require that some pins be preassigned, before routing. However, it is recommended that the unconstrained design be routed first, to verify that it can meet timing specifications. If it does not, the constrained design will also be too slow.

Keeping a design synchronous, with a single external clock source, allows timing-driven routing tools to work more efficiently. The parts have clock enables, so there is no need for special measures to gate clocks within a design.

FPGAs are register-rich. Therefore, it is advantageous to employ one-hot encoding in state machines. This leads to simpler next-state and output logic. This form of encoding is sometimes referred to as state-per-bit encoding, because a unique single flip-flop is asserted for each state. Coding style has an impact on the results of targeting a description into an FPGA. One notable example is in the description of a sequencer. If the count sequence does not have to be binary, linear feedback shift registers may be a more attractive alternative because they require less space and route more efficiently than binary counters. Designers should be aware that flip-flops in FPGAs tend to initialize to a cleared output during power-up. A state machine would have to anticipate this condition because it is not one of the explicit one-hot codes.

REFERENCES


**RELATED WEB SITES**

- www.accellera.org Accellera
- www.actel.com Actel Corp.
- www.altera.com Altera, Inc.
- www.atmel.com Atmel Corp.
- www.opencores.org Opencores
- www.synopsys.com Synopsys, Inc.
- www.synplicity.com Synplicity, Inc.
- www.vsia.com Virtual Socket Interface Alliance
- www.xilinx.com Xilinx, Inc.

**PROBLEMS**

1. Using the ROM model given in Example 8.1, develop and verify *comp_2_ROM*, a Verilog model of a 2-bit comparator.

2. The 2-bit comparator presented in Example 8.1 has three outputs. Develop a new model that encodes the outputs in a 2-bit word. Build a testbench that will accept and decode the output of the model and assert one of three outputs corresponding to the outputs of the original model.

3. Estimate the number of memory cells that would be required to implement a 16-bit adder in a ROM.

4. Write a Verilog model of a 256 × 8 ROM that stores the product of two 4-bit unsigned binary words, as shown in Figure P8-4. Use the multiplier (mprod) and multiplicand (mcond) bits to form the address of the ROM.

5. Write a testbench and verify the Verilog model of the static RAM cell, *RAM_static*, given in Example 8.3.

6. Develop an alternative model for *RAM_static* that uses a level-sensitive cyclic (*always*) behavior instead of a continuous assignment (see Example 8.3).

7. **FPGA-Based Design Exercise: A simple ALU**

   The top-level module of a sequential machine, *alu_machine_4_bit*, is depicted in Figure P8-7a, with the input and output ports that interface the module to its
environment. The machine will be described for implementation on a FPGA prototyping board.

The machine is to operate synchronously as follows: $\text{Led\_idle}$ will indicate that the machine is in its "reset" state. When $\text{Go}$ is asserted an internal register is to be loaded with the content of $\text{Data}[3:0]$ and assert $\text{Led\_wait}$ until $\text{Go}$ is de-asserted. After $\text{Go}$ is de-asserted, $\text{Led\_rdy}$ is to assert. While $\text{Led\_rdy}$ is asserted the slide switches (on a prototyping board) may be used to set a new value for $\text{Data}[3:0]$ and/or $\text{Opcode}[2:0]$. As the slide switches are changed the effect should be apparent at $\text{Alu\_out}$. The cycle is to repeat if $\text{Go}$ is re-asserted while $\text{Led\_rdy}$ is asserted (i.e., the storage register is to be reloaded). The machine is to be synchronized by the rising edge of a clock, and have synchronous active-high reset.

**Design—Partition**

An architectural partition of $\text{ALU\_machine\_4\_bit}$ is shown in Figure P8-7b. The architecture has three functional subunits: an ALU, a storage register, and
a state-machine controller. The ALU implements an instruction set (described below), and the controller directs the operations of the machine. One input datapath of ALU
machine 4 bit is connected to the internal storage register, and the other is connected to one data port of the ALU. The output of the register is connected to the other data port of the ALU. The datapath is to be controlled by Toggle Button, a state machine that will be described below. The opcode and the input datapath of ALU machine 4 bit will be controlled by the manual slide switches on the Digilab\footnote{See www.digilent.cc} prototyping board for Xilinx Spartan-XL parts. Board LEDs will indicate the internal status of the system (i.e., Led_idle, Led_wait, and Led_ready) and are derived from the state within Toggle Button.

Our design of ALU machine 4 bit will be progressive. Two functional units, ALU 4 bit and Register, will be designed and separately verified first. The state machine controller, Toggle Button, will be designed later, along with a programmable clock generator. Then we will integrate the individually verified functional units, verify that the integrated design has the correct functionality, and achieve final presynthesis sign-off. The last step will be to synthesize the design into a working prototype on the Digilab board.
TABLE P8-7a Functional specification for a 4-bit ALU.

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Add</td>
<td>Data_A + Data_B</td>
</tr>
<tr>
<td>001</td>
<td>Sub</td>
<td>Data_A - Data_B</td>
</tr>
<tr>
<td>010</td>
<td>Not_A</td>
<td>~Data_A</td>
</tr>
<tr>
<td>011</td>
<td>Not_B</td>
<td>~Data_B</td>
</tr>
<tr>
<td>100</td>
<td>A_and_B</td>
<td>Data_A &amp; Data_B</td>
</tr>
<tr>
<td>101</td>
<td>A_or_B</td>
<td>Data_A</td>
</tr>
<tr>
<td>110</td>
<td>Ror_A</td>
<td>Data_A</td>
</tr>
<tr>
<td>111</td>
<td>Rnd_B</td>
<td>&amp;Data_B</td>
</tr>
</tbody>
</table>

Design—ALU

Using the module...endmodule encapsulation and port declarations given below, write a Verilog model of ALU_4_bit, a 4-bit ALU shown in Figure P8-7c and specified in Table P8-7a.

```verilog
module ALU_4_bit (Alu_out, Data_A, Data_B, Opcode);
    output [4: 0] Alu_out;
    input [3: 0] Data_A, Data_B;
    input [2: 0] Opcode;
    ...
endmodule
```

Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using the test plan, write a testbench, t_ALU_4_bit, that verifies the functionality of ALU_4_bit.

Verify your design for a suitable number of patterns that cover the data and opcodes of the ALU. As an example, complete Table P8-7b by specifying Alu_out for the indicated patterns. Complete the second table with patterns that you choose. Include these patterns in your testbench, along with others that you choose.

![Figure P8-7c 4-bit ALU](image)
### TABLE P8-7b
Sample data calculations for test patterns to be applied to ALU_4_bit.

<table>
<thead>
<tr>
<th>Data_A</th>
<th>1010</th>
<th>1111</th>
<th>0101</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data_B</td>
<td>0101</td>
<td>0101</td>
<td>1010</td>
<td>1111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Alu_out</th>
<th>Alu_out</th>
<th>Alu_out</th>
<th>Alu_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>00010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not_A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not_B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_and_B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_or_B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ror_A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rand_B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Note
Management appreciates your efforts to arrange the waveforms in the graphical display to enhance the utility of the information, minimizing the amount of interpretation and translation that must be done. (Use the “Set Radix” option in the Silos-III user interface.) Consider annotating the waveforms by hand or by a graphical editor tool to label opcodes etc., or define and display test parameters to indicate mnemonics for opcodes.

Next, write a Verilog model of a 4-bit storage register that has parallel load capability. The register is to be synchronized by the rising edge of a clock, and have active-high synchronous reset. Use the module encapsulation and ports given below.

```verilog
module Register (Reg_out, Data, Load, clk, reset);
    output [3: 0] Reg_out;
    input [3: 0] Data;
    input            Load, clk, reset;

... endmodule
```
Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using the test plan, write a testbench, t_Register, that verifies the functionality of Register.

Design—Programmable Clock

Using the programmable clock described in Problem 33 in Chapter 5, include the following "annotation module" in your project. The role of this module is to override the default parameters in the clock generator with those that are to be used in a given application. The annotation module uses hierarchical dereferencing, where $M$ is the instance name of the unit under test (UUT) in $t_{\text{Clock Prog}}$. Your testbench must demonstrate that this works. The example below replaces the default values of Latency, Offset, and Pulsewidth by 10, 5, and 5, respectively.

```verilog
module annotate_Clock_Prog ();
  defparam t_Clock_Prog.M1.Latency = 10;
  defparam t_Clock_Prog.M1.Offset = 5;
  defparam t_Clock_Prog.M1.Pulse_Width = 5;
endmodule
```

Design—User Interface

The limited switch resources of the prototyping board create a need for a toggle button machine. It will be used to load data into the machine's register.

The state machine described by the ASM chart in Figure P8-7d has synchronous reset and resides in $S_{\text{idle}}$ with Led_idle asserted, until Go is asserted. Then it moves to $S_{\text{1}}$, where it asserts Load for one clock cycle and then enters $S_{\text{2}}$, where it asserts Led_wait and remains until Go is de-asserted. When Go is de-asserted the machine enters $S_{\text{3}}$ and asserts Led_rdy. The machine remains in $S_{\text{3}}$ until Go is again asserted. Then it returns to $S_{\text{1}}$. This sequence of state transitions lets us load data into a register, wait in $S_{\text{2}}$ until the Go button is de-asserted, and then pause in $S_{\text{3}}$, where other actions can be taken to operate on a datapath. For example, data can be placed on the input port and the output port can be examined under the action of the opcodes. The outputs Led_idle, Led_wait, and Led_rdy indicate the status of the machine, and can be used to control LEDs on the Digilab prototyping board.

Write a Verilog model of the sequential machine Toggle_Button, using the module encapsulation and ports declared below:

```verilog
module Toggle_Button (Load, Led_idle, Led_wait, Led_rdy, Go, clk, reset);
  output Load, Led_idle, Led_wait, Led_rdy;
  input Go, clk, reset;
  reg [1:0] state, next_state;

endmodule
```
Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using the test plan, write a testbench, `t_Toggle_Button`, that verifies the functionality of `Toggle_Button`.

**Design—Integration and Verification**

Now we will integrate the functional units of the architecture for `ALU机器4位` and verify that the functionality of the integration is correct, leading to presynthesis sign-off. Instantiate `ALU_4_bit`, `Register`, and...
Toggle_Button into ALU_machine_4_bit and create a Verilog model of the internal structure represented by the architecture shown in Figure P8-7b. Use the module header and declarations shown below.

```verilog
module ALU_machine_4_bit (  
  Alu_out,
  Led_idle, Led_wait, Led_rdy,
  Data,
  Opcode,
  Go,
  clk, reset);

output [4: 0] Alu_out;
output Led_idle, Led_wait, Led_rdy;
input [3: 0] Data;
input [2: 0] Opcode;
input [1: 0] Reg_out;
wire [3: 0] M1 (Alu_out, Data, Reg_out, Opcode);
wire [2: 0] M2 (Reg_out, Data, Load, clk, reset);
wire [1: 0] M3 (Load, Led_idle, Led_wait, Led_rdy, Go, clk, reset);
endmodule
```

Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using your test plan, and the headers and instantiations shown below, write a carefully documented testbench, `t_ALU_machine_4_bit`, that implements the machine, and verify the functionality of `ALU_machine_4_bit`. Note that the testbench contains the UUT (`ALU_machine_4_bit`), and the programmable clock generator (`Clock_Prog`), plus the code you write to execute the tests.

```verilog
module annotate_ALU_machine_4_bit ();
  defparam t_ALU_machine_4_bit.M2.Latency = 10;
  defparam t_ALU_machine_4_bit.M2.Offset = 5;
  defparam t_ALU_machine_4_bit.M2.Pulse_Width = 5;
endmodule
```

```verilog
module t_ALU_machine_4_bit ();
  reg Go, reset;
  reg [3: 0] Data;
  reg [2: 0] Opcode;
  wire [4: 0] Alu_out;
  wire [2: 0] Load;
  ALU_machine_4_bit M1 (  
    Alu_out, // Instantiate UUT
```
Led_idle, Led_wait, Led_rdy,
Data,
Opcode,
Go, Load,
clk, reset);
Clock_Prog M/2 (clk);
...
// Your code goes here
endmodule

Figure P8-7c shows the results of a simple test of $ALU_{machine \_4 \_bit}$. Note the organization of the display.
Design—Prototype Synthesis and Implementation

The final steps are to synthesize \texttt{ALU\_machine\_4\_bit} into a Xilinx FPGA, download the design to the Digilab prototype board, and demonstrate that the prototype functions correctly, concluding with final sign-off. The ports of the module, the pads of the FPGA, and the I/O resources of the board must be integrated. The first step in this process is to decide which board resources will be mapped to the ports of the design. The second step is to map the ports to the IOBs of the FPGA. The pin configuration of the FPGA is described in the manufacturer's data sheets for the Xilinx Spartan-XL10 part. The board's I/O resources are described in the documentation for the Digilab Spartan XL prototyping board. Although a tool (e.g., Xilinx \texttt{Foundation Express}) will map ports to pads (pins) automatically, the result might not be compatible with the fixed pad locations on the prototyping board, and may even vary from run to run. It is advisable to constrain the pad mapping. In this application, the pads of the FPGA have been hard wired to certain pins of the prototyping board. It is critical that the correct signals be mapped to the pins that are being used by the application.

The datapath of \texttt{ALU\_machine\_4\_bit} will be controlled by the finite-state machine \texttt{Toggle\_Button} (previously designed and verified). The signal \texttt{Go} initiates the activity of loading \texttt{Data} into \texttt{Register}. \texttt{Data} and \texttt{Opcode} are to be controlled by the outputs of the manual slide switches. While the state of \texttt{Toggle\_Button} is \texttt{S\_3} the machine asserts \texttt{Led\_rdy}, and the slide switches for \texttt{Data} and \texttt{Opcode} can be exercised to test the machine by presenting different words to the datapaths of the ALU. A value can be loaded into \texttt{Register}, then a different value can be arranged for \texttt{Data} by changing the slide switches after \texttt{Led\_rdy} is asserted.

The input port signals of \texttt{ALU\_machine\_4\_bit} must be mapped to the slide switches and push buttons of the Digilab board; the output ports are to be mapped to the LEDs. Figure P8-7f shows the (a) slide switch configuration, (b) push-button configuration, and (c) the LED configuration that are to be used. These resources are connected to pins at the J2 connector on the board. Note: The side of the slide switch that is closest to the nearest edge of the board is logical 1.

Figure P8-7f shows the LED, button and switch pin assignments that have been specified for \texttt{ALU\_machine\_4\_bit}, which are listed in Table P8-7c.

Write a test plan that specifies how the FPGA prototype circuit will be tested, making specific reference to the board resources (e.g., slide switch configurations, LED readouts, special instrumentation, etc.). Develop test cases demonstrating that the ALU works correctly. Verify that the \texttt{Go} button, the reset button, and the LEDs function correctly. After successfully completing all of the above steps, create the bit-stream file and download it to the prototyping board. Verify the functionality by executing (1) ALU and Opcode tests, (2) \texttt{Led\_idle}, \texttt{Led\_wait}, and \texttt{Led\_rdy} assertion tests, (3) reset assertion test, and (4) any other test that will impress management. Using some of the test cases developed for the test plan, execute the test plan and demonstrate that the prototype functions correctly.
FIGURE P8-7f  Digilab/XLA Pin assignments: (a) slide switches, (b) push buttons, and (c) LEDs.

TABLE P8-7C  Pin assignments for the prototype board.

<table>
<thead>
<tr>
<th>Port</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alu_out [4]</td>
<td>P69</td>
</tr>
<tr>
<td>Alu_out [3]</td>
<td>P68</td>
</tr>
<tr>
<td>Alu_out [1]</td>
<td>P66</td>
</tr>
<tr>
<td>Alu_out [0]</td>
<td>P65</td>
</tr>
<tr>
<td>Led_idle</td>
<td>P62</td>
</tr>
<tr>
<td>Led_wait</td>
<td>P61</td>
</tr>
<tr>
<td>Led_rdy</td>
<td>P60</td>
</tr>
<tr>
<td>Data [3]</td>
<td>P28</td>
</tr>
<tr>
<td>Data [2]</td>
<td>P27</td>
</tr>
<tr>
<td>Data [1]</td>
<td>P26</td>
</tr>
<tr>
<td>Data [0]</td>
<td>P25</td>
</tr>
<tr>
<td>Opcode [2]</td>
<td>P23</td>
</tr>
<tr>
<td>Opcode [1]</td>
<td>P20</td>
</tr>
<tr>
<td>Opcode [0]</td>
<td>P19</td>
</tr>
<tr>
<td>clk</td>
<td>P13</td>
</tr>
<tr>
<td>Go</td>
<td>P56</td>
</tr>
<tr>
<td>reset</td>
<td>P57</td>
</tr>
</tbody>
</table>
8. FPGA-Based Design Exercise: Some Ring Counters

Using the shell below write and verify a Verilog model of Counter8_Prog, a parameterized and programmable 8-bit counter that implements various display patterns to exercise the Digilab prototyping board. The implementation is to use a separate Verilog function for each pattern (e.g., ring1_count). Develop a test plan clearly listing each functional feature that is to be tested. Develop a carefully documented testbench and execute the testplan to debug and verify the model and generate final graphical results. (Note: Organize the display to have the format shown Figure P8.8a.)

module Counter8_Prog (count, mode, direction, enable, clk, reset);
output [7: 0] count; // Hardwired for demo board
input [1: 0] mode; // Determine pattern sequence displayed by count
input direction; // Determines movement (left/up, right/down)
input enable;
input clk, reset;
reg count;
parameter start_count = 1; // Sets initial pattern of the display to LSB of count

// Mode of count
parameter binary = 0;
parameter ring1 = 1;
parameter ring2 = 2;
parameter jump2 = 3;

// Direction of count
parameter left = 0;
parameter right = 1;
parameter up = 0;
parameter down = 1;

always @(posedge clk or posedge reset)
if (reset == 1) count <= start_count;
else if (enable == 1)
case (mode)
  ring1: count <= ring1_count (count, direction);
  ring2: count <= ring2_count (count, direction);
  jump2: count <= jump2_count (count, direction);
  default: count <= binary_count (count, direction);
endcase

function [7:0] binary_count;
input [7:0] count;
input direction;
begin
  if (direction == up) binary_count = count + 1; else binary_count = count - 1;
end
endfunction

// Other functions are declared here.
endmodule

At the active edge of the clock, an 8-bit count will be updated under the control of mode and direction, which selects one of four different functions to form the next value of count.

**binary:** A binary count pattern controlled by *direction* to count up or down.

**ring1:** A ring counter controlled by *direction* to move left (up) or right (down).

**ring2:** A ring counter like *ring1*, but that moves two adjacent cells at a time.
**jump2**  A ring counter that jumps by two cells.

The patterns that are to be implemented for *ring2* and *jump2* are illustrated in Figure P8-8b.

![Figure P8-8b](image)

**FIGURE P8-8b** Patterns to be generated by *ring2* and by *jump2.*

---

**Design—Prototype Synthesis and Implementation**

The frequency of the clock signal at Pin 13 on the prototyping board is 25 MHz. Model and verify a clock divider that will produce an internal clock signal whose frequency will be low enough to allow changes in the LEDs to be visible.

Your design is to be encapsulated in a module, *TOP*, having the following structure:

```verilog
module TOP (count, mode, direction, enable, clk, reset);
  input ... 
  output ... 

Clock_Divider M0 (clk_internal, clk);
Counter8 Prog UUT (count, mode, direction, enable, clk_internal, reset);
endmodule
```
Synthesize and implement your design on the Digilab prototyping board. Develop a hardware test plan and use it to test the operation of the prototype.

Using the pin assignments shown in Figure P8-8c, connect the pins of the design to the pins of the FPGA on the Digilab-XLA board. Connect the clock port, clk, to Pin 13, which is hard wired to the 25 MHz clock signal from the clock chip.

Develop and verify a Verilog model of Jumper, a module that generates the pattern in Figure P8-8d. Verify the hardware prototype.
9. FPGA-Based Design Exercise: SRAM with Controller

The static RAM modeled by $SRAM_{2048\times8}$ (see Example 8.5) is asynchronous. Many applications require a synchronous interface to an SRAM. One such interface (controller) is illustrated in Figure P8-9a(a) below, where a processor provides an address strobe, $ADS$, a read/write signal $R\_W$, a clock signal, and a reset to $SRAM\_Con$, which forms the signals $OE\_b$, $CS\_b$, and

![SRAM with Controller Diagram](image)

**FIGURE P8-9a**  Synchronous controller for a SRAM: (a) hierarchical block diagram with interface signals, and (b) ASM chart for the controller.
WE_b, which control the SRAM, and a signal, Rdy, which asserts for one clock cycle at the end of a read or write sequence. The ASM chart in Figure P8.9a(b) describes the controller (Note: the assertions of the active-low signals of the implementation can be inferred from these on the ASM chart.)

Form a generic module, SRAM, by renaming SRAM_2048B and re-using its code. Using the headers and testbench below, develop a generic (parameterized) controller module, SRAM_Con that implements the behavior of the ASM chart in Figure P8.9a(b), then instantiate SRAM and SRAM_Con within SRAM_with_Con. Use the results shown in Figure P8.9b to organize the displayed information (including the bus activity) produced by the simulator.

Note that the testbench contains commented statements that would cause the model to fail, because the bus activity presents a high-impedance condition before the SRAM can latch the data. The remedy is shown in the code for the testbench, and consists of conditioning the bus to enter a high-impedance condition after the rising latching edge of CS_b or WE_b in the SRAM.

**FIGURE P8.9b** Simulation results for SRAM_Con.
Design—Prototype Synthesis and Implementation

After verifying the functionality of the model for the parameters of SRAM_2048_8, choose parameters that will size the model to fit in the Xilinx Spartan-XL device S10XLC084 for prototype implementation of the integrated unit on the Digilab-XLA board. Consider the structure shown in Figure P8-9c, where the inbound data and the address are mapped to the pins of the board's slide switches, and ADS, R_W, and reset are mapped to push buttons. Rdy is to be mapped to a LED. The output data is to be mapped to the board’s seven segment displays (a decoder will be needed) or to the LEDs. Consider the following issues: (1) bus contention, (2) display of the machine state (consider the LEDs), and (3) clock speed. The module Toggle_Synch is to synchronize the asynchronous input ADS and to have logic that generates a single pulse on assertion of ADS regardless of how long the push button is pressed.

```verbatim
timescale 1ns / 10ps
module SRAM_with_Con (data, addr, Rdy, ADS, R_W, clk, reset);
    parameter     word_size = 8;
    parameter     addr_size = 11;
    inout [word_size - 1: 0] data;
    input [addr_size - 1: 0] addr;
    output Rdy;
    input ADS, R_W;
    input clk, reset;
    SRAM_Con M0 (Rdy, CS_b, OE_b, WE_b, ADS, R_W, clk, reset);
    SRAM M1 (data, addr, CS_b, OE_b, WE_b);
endmodule

module SRAM_Con (Rdy, CS_b, OE_b, WE_b, ADS, R_W, clk, reset);
    output Rdy;
```
output CS_b;
output OE_b;
output WE_b;
input ADS;
input R_W;
input clk, reset;
reg [2:0] state, next_state;
reg CS_b, OE_b, WE_b;
reg Rdy;

endmodule

module SRAM (data, addr, CS_b, OE_b, WE_b);

endmodule

module test_SRAM_with_Con ();

parameter word_size = 8;
parameter addr_size = 11;
parameter mem_depth = 128;
parameter col_addr_size = 4;
parameter row_addr_size = 7;
parameter num_col = 16;
parameter initial_pattern = 8'b000_0001;
parameter Hi_Z_pattern = 8'bzzzz_zzzz;
parameter stop_time = 290000;
parameter latency = 248000;
reg [word_size -1: 0] data_to_memory;
reg ADS, R_W, clk, reset;
reg send, recv;
integer col, row;
wire [col_addr_size -1:0] col_address = col;
wire [row_addr_size -1:0] row_address = row;
wire [addr_size -1:0] addr = (row_address, col_address);

// Three-state, bi-directional bus
wire [word_size -1:0] data_bus = send? data_to_memory: Hi_Z_pattern;
wire [word_size -1:0] data_from_memory = recv? data_bus: Hi_Z_pattern;

SRAM_with_Con M1 (data_bus, addr, Rdy, ADS, R_W, clk, reset); // UUT

initial #stop_time $finish;
initial begin reset = 1; #1 reset = 0; end
// Non-Zero delay test: Write walking ones to memory
initial begin
    ADS = 0;
    R_W = 0;
    send = 0;
    recv = 0;
    for (col = 0; col <= num_col - 1; col = col + 1) begin
        data_to_memory = initial_pattern;
        for (row = 0; row <= mem_depth - 1; row = row + 1) begin
            @ (negedge clk);
            @ (negedge clk);
            @ (negedge clk) ADS = 1; R_W = 0; // writing
            @ (negedge clk) ADS = 0;
            @ (posedge clk) send = 1;
            // @ (posedge clk) send = 0;  // Does not work
            @ (posedge M1.M1.WE_b or posedge M1.M1.CS_b) send = 0;
            // @ (posedge clk) #1 send = 0; //Replacing above line with this works too.
            @ (posedge clk) data_to_memory =
                (data_to_memory[word_size-2:0], data_to_memory[word_size-1:1]);
        end
    end
end

// Non-Zero delay test: Read back walking ones from memory
initial begin
    #latency;
    ADS = 0;
    R_W = 1;
    send = 0;
    recv = 1;
    ADS = 1;
    for (col = 0; col <= num_col - 1; col = col + 1) begin
        for (row = 0; row <= mem_depth - 1; row = row + 1) begin
            #50;
        end
    end
end

// Testbench probe to monitor write activity
reg [word_size -1:0] write_probe;
always @ (posedge M1.M1.WE_b or posedge M1.M1.CS_b)
case (M1.M1.col_addr)

0: write_probe = M1.M1.RAM_col0[M1.M1.row_addr];
endcase
endmodule

10. FPGA-Based Design Exercise: Programmable Lock

The objective of this exercise is to design and implement a hardware prototype of a programmable digital combination lock, using the Digilab-XLA prototyping board and the Grayhill 072 hexadecimal keypad. The top-level block diagram of the programmable lock is shown in Figure P8-10a.

The programmable lock has two modes: normal and programming. The action of reset is to place the machine into the normal mode. Pressing mode once will put the machine in the program mode. In the program mode the Prog_Mode LED is asserted and the machine accepts a correct sequence of eight entries from a hex keypad. On receipt of a correct entry code the machine asserts the Enter_code LED and awaits entry of six hex values from the keypad. If the sequence of eight values is not correct the machine returns to the reset state and de-asserts the Prog_Mode LED. The six values entered after the Enter_code LED is asserted will be the key for the lock. After they are entered the machine returns to the reset state and the normal mode automatically. The master code required to enter the program mode is hard-wired to an assigned value of your choice.

In the normal mode, a user must enter a sequence of six hex characters. If the sequence of characters matches the key, the Unlock LED is asserted and blinks at a low frequency that is visible to the human eye. If the sequence does
not match the key, the False_Code LED is asserted and blinks at a high rate until reset is asserted.

Develop an ASM chart for a programmable lock meeting the specifications given above. Using the ASM chart and the module...endmodule encapsulations and ports given below, develop and verify a Verilog model of Prog_Lock.

module Prog_Lock (Col, valid, Enter_mode, Prog_mode, Unlock, Row, mode, clk, reset);
endmodule

module Top_Prog_Lock (Col, Enter_mode, Prog_mode, Unlock, Row, mode, clk, reset);
endmodule

The module Top_Prog_Lock is to encapsulate the programmable lock and interface to the LEDs, push buttons, and the keypad.

Design—Issues

Consider debouncing the keypad and/or reducing the clock frequency. Contact bounce is 4 ms at make and 10 ms at break. Specifications for the keypad are available at www.grayhill.com. The action of reset should not erase the stored
key. As a worst case, a clear button might have to be added to your design, but the design can be implemented cleverly without it. For security, consider having the displays blank until a key is read.

Design—Prototype Synthesis and Implementation

The Grayhill keypad is to be connected to the Digilab-XLA board as shown below in Figure P8-10b. It is essential that the orientation of the ribbon cable and the connector be exactly as shown. This configuration uses the same pins that connect to the slide switches on the board. Therefore, the slide switches must be placed in the 0 position (away from the nearest edge of the board), and may not be used for any other function.

![FIGURE P8-10b Digilab board pin mapping for the Grayhill 072 Hexadecimal keypad.](image)

11. FPGA-Based Design Exercise: Keypad Scanner with FIFO Storage

The objective of this multistage exercise is to systematically design and implement an FPGA-based keypad scanner integrated with a FIFO for data storage and retrieval, a display mux, and the seven-segment displays, slide switches, and LEDs of the Digilab-XLA prototyping board. The top-level block diagram of the system is shown in Figure P8-11a, with the partitioned system. The hardware prototype will be verified to operate with the Grayhill 072 hex Keypad.

The user interface consists of the following inputs: a mode toggle button, a read button, a reset button, and a hexadecimal keypad. The outputs are two seven-segment displays and eight LEDs. The button mode_toggle will be used to toggle between display states so that more than eight signals can be presented for view. When a button of the hex keypad is pressed, the system must decode the button and store the data in an internal FIFO. The read button will be
FIGURE P8-11a FIFO Keypad Scanner: (a) top-level block diagram, and (b) system partition and architecture.
used to read data from the FIFO, and to display the data on the seven-segment displays. The LEDs will display the status of the FIFO and other information.

The system architecture is subject to future engineering change orders (ECOs) as the customer's specifications evolve to accommodate a rapidly changing marketplace. Note that the specification has not considered the need for a switch debounce circuit and that it does not address the constraints that will be imposed by the Digilab's boards circuitry for the seven-segment displays.

**Design: FIFO**

The design will use the keypad decoder and synchronizer that were presented in Chapter 5. This part of the exercise will integrate a FIFO with the keypad scanner. A FIFO (first-in, first-out) buffer is a dedicated memory stack consisting of a fixed array of registers. The FIFO that is to be used in this exercise is shown in Figure P8-11b. The registers of the stack operate synchronously (rising edge) with a common clock, subject to reset. The stack has two pointers (addresses), one pointing to the next word to which data will be written and another pointing to the next word that will be read, subject to write and read inputs, respectively. The FIFO has input and output datapaths, and two bit-lines serving as flags to denote the status of the stack (full or empty).

Using the FIFO model and testbench provided below, verify the operation of the FIFO in the Silos-III environment. Configure the FIFO to have a stack of eight registers of 4 bits width.

![FIGURE P8-11b FIFO Buffer: Signal Interface.](image-url)
module FIFO (
    Data_out, // Data path from FIFO
    stack_empty, // Flag asserted high for empty stack
    stack_full, // Flag asserted high for full stack
    Data_in, // Data path into FIFO
    write_to_stack, // Flag controlling a write to the stack
    read_from_stack, // Flag controlling a read from the stack
    clk, rst // External clock and reset
);

parameter stack_width = 4; // Width of stack and data paths
parameter stack_height = 8; // Height of stack (in # of words)
parameter stack_ptr_width = 3; // Width of pointer to address stack
output [stack_width-1:0] Data_out;
output stack_empty, stack_full;
input [stack_width-1:0] Data_in;
input clk, rst;
input write_to_stack, read_from_stack;

// Pointers (addresses) for reading and writing
reg [stack_ptr_width-1:0] read_ptr, write_ptr;
reg [stack_ptr_width : 0] ptr_diff; // Gap between
    // ptrs
reg [stack_width-1:0] Data_out;
reg [stack_width-1:0] stack [stack_height-1 : 0]; // memory
    // array

assign stack_empty = (ptr_diff == 0) ? 1'b1 : 1'b0;
assign stack_full = (ptr_diff == stack_height) ? 1'b1 : 1'b0;

always @ (posedge clk or posedge rst) begin: data_transfer
    if (rst) begin
        Data_out <= 0;
        read_ptr <= 0;
        write_ptr <= 0;
        ptr_diff <= 0;
    end
    else begin
        if ((read_from_stack) && (!stack_empty)) begin
            Data_out <= stack [read_ptr];
            read_ptr <= read_ptr + 1;
            ptr_diff <= ptr_diff -1;
        end
        else if ((write_to_stack) && (!stack_full)) begin
            stack [write_ptr] <= Data_in;
        end
    end
end
write_ptr <= write_ptr + 1;  // Address for next clock edge
ptr_diff <= ptr_diff + 1;
end
end
end // data_transfer
demodule

module t_FIFO ();
parameter stack_width = 4;
parameter stack_height = 8;
parameter stack_ptr_width = 3;
wire [stack_width-1 : 0] Data_out;
wire stack_empty, stack_full;
reg [stack_width-1 : 0] Data_in;
reg clk, rst, write_to_stack, read_from_stack;
wire [11:0] stack0, stack1, stack2, stack3, stack4, stack5, stack6, stack7;

assign stack0 = M1.stack[0];  // Probes of the stack
assign stack1 = M1.stack[1];
assign stack2 = M1.stack[2];
assign stack3 = M1.stack[3];
assign stack4 = M1.stack[4];
assign stack5 = M1.stack[5];
assign stack6 = M1.stack[6];
assign stack7 = M1.stack[7];

FIFO M1 (Data_out, stack_empty, stack_full, Data_in, write_to_stack, read_from_stack, clk, rst);

always begin clk = 0; forever #5 clk = ~clk; end
initial #1500 $stop;
initial begin
#10 rst = 1;
#40 rst = 0;
#420 rst = 1;
#460 rst = 0;
end
initial ford
#80 Data_in = 1;
forever #10 Data_in = Data_in + 1;
join
initial ford
#80 write_to_stack = 1;
#180 write_to_stack = 0;
#250 read_from_stack = 1;
#350 read_from_stack = 0;
Design: Decoder

The decoder must be designed to decode a word read from the FIFO and create driving signals for the active-low, seven-segment displays on the Digilab board. The decoder must generate display signals for the 16 codes of the Grayhill 072 keypad. Using the module ... endmodule encapsulation and port given below, write a Verilog module, Decoder_L, of a functional unit that forms the left and right (active-low) codes of two seven-segment displays. The MSB of Left_out and Right_out must be mapped to the “a” segment of the display, and the LSB must be mapped to the “g” segment of the string “abcdefg.”

module Decoder_L (Left_out, Right_out, Code_in);  // active low displays
output [6:0] Left_out, Right_out;
input [3:0] Data_in;
endmodule

Write a test plan specifying how Decoder_L is to be tested. Using the test plan, write a testbench, t_Decoder_L, that verifies the functionality of Decoder_L, and execute the test plan.

Design—Display Units

The next objective is to develop functional units supporting the reading and displaying of the contents of the FIFO on the seven-segment displays of the Digilab prototyping board. The seven-segment displays on the Digilab prototyping board have a common anode. Each unit has seven cathode pins, corresponding to the segments string “abcdefg.” We wish to implement the structure shown in Figure P8-11c below. The output of the FIFO will be decoded to form the active-low code of the left and right displays. A mux will select between the two codes and route them to the appropriate segment simultaneously with an assertion of the appropriate anode. A clock divider must be used to strobe the displays at a frequency that is high enough to eliminate the flicker effect (i.e., at a frequency above the bandwidth of the human eye) and low enough to be displayed by the LED.

The FIFO module will read its data on the active edge of the clock while the read input signal is asserted. The prototyping boards have clocks that are running at 25 MHz or 50 MHz, depending on the model. In either case, a single push of the button would cause the entire content of the FIFO to be dumped before the button could be released. Therefore, a machine must be designed to accept the button signal and assert a read signal at the FIFO for only one clock signal. The button must be de-asserted before another read can occur.
Design—Clock Divider

Now we will design a parameterized clock divider that can be used to strobe the
mux controlling the seven-segment displays and to operate the system at a suit-
able frequency. Using the module . . . endmodule encapsulation below, develop a
Verilog module of Clock_Divider, a parameterized clock divider having a de-
fault division by $2^3$.

module Clock_Divider (clk_out, clk_in, reset);

endmodule

Design—Asynchronous User Interface

Two additional units must be designed: (1) a synchronizer for the “read” signal
controlling the FIFO, and (2) a toggle unit that allows only one cell of the FIFO
to be read at a time. Using the module . . . endmodule encapsulation below, de-
velop a Verilog module of Synchro_2, a two-stage synchronizer for the signal
that reads the FIFO. The output of Synchro_2 should be synchronized to the
negative edge of clk, because the state machine is active on the positive edge.

module Synchro_2 (synchro_out, synchro_in, clk, reset);

endmodule

Using the module . . . endmodule encapsulation below, develop a Verilog
module of Toggle a state machine that accepts the synchronized signal directing
that the FIFO be read, and asserts a signal that reads only one cell of the FIFO,
regardless of whether the user holds the “read” button for more than one clock
cycle or not. The machine is to ensure that only one cell of the FIFO is read each time the button is pushed.

module Toggle (read_fifo, read_sync, clk, reset);
...
endmodule

Write a test plan specifying how Toggle is to be tested. Using the test plan, write a testbench, i_Toggle, that verifies the functionality of Toggle, and that it operates correctly with the FIFO. Execute the test plan.

We will now design a multiplexer to control the common-anode, seven-segment displays of the Digilab prototyping board. Using the module ...

endmodule encapsulation below, develop a Verilog module of Display_Mux_3_4, a functional unit that selects between two cathode codes and asserts the selected code at its output, and also asserts the appropriate anode of the rightmost pair of seven-segment displays on the board.

module Display_Mux_3_4 (Cathode, Left_anode, Right_anode, Display_3, Display_4, sel);
...
endmodule

Write and execute a test plan for verifying Display_Mux_3_4.

The following objectives remain: (1) integrate the functional units of the FIFO keypad system that were designed above, (2) synthesize the integrated system and implement it in a Xilinx FPGA, and (3) conduct a hardware verification of the working system.

For simplicity, the implementation will omit the mode_toggle button controlling the LED display. Instead, the LEDs will be hard wired to display [state[5:0], empty, full], the state of the controller and the status of the FIFO. An optional version would use the mode_toggle button to toggle between the LED data groups.

Design—System Integration

We will integrate the previously designed and verified functional units, and verify that the integrated system functions correctly. Using the top-level encapsulating module TOP_Keypad_FIFO given below, form the integrated system. Using the SILOS-III verification environment, eliminate any syntax errors from the integration. Pay careful attention to the mapping of formal and actual port names.

module TOP_Keypad_FIFO (Cathode, Col, Left_anode, Right_anode, valid, empty, full, Row, read, clk, reset);
output [6: 0] Cathode;
output [3: 0] Col;
output Left_anode, Right_anode;
output valid;
output empty;
output full;
input [3: 0] Row;
input read;
input clk, reset;
wire [3: 0] Code, Code_out;
wire S_Row;
wire valid;
wire [6: 0] Left_out, Right_out;
wire clk_slow, clk_display;
wire read_fifo, read_sync;

Synchronizer M0 (  
.S_Row(S_Row),  
.Row(Row),  
.clock(clk_slow),  
.reset(reset));

Hex_Keypad_Grayhill_072 M1(  
.Code(Code),  
.Col(Col),  
.Valid(valid),  
.Row(Row),  
.S_Row(S_Row),  
.clock(clk_slow""),  
.reset(reset));

FIFO M2 (  
.Data_out(Code_out),  
.stack_empty(empty),  
.stack_full(full),  
.Data_in(Code),  
.write_to_stack(valid),  
.read_from_stack(read_fifo);  
.clk(~clk_slow),  
.rst(reset),  
);

Decoder_L M3 (  
.Left_out(Left_out),  
.Right_out(Right_out),  
.Code_in(Code_out));

Display_Mux_3_4 M5 (  
.Cathode(Cathode),  
.Left_anode(Left_anode),  
.Right_anode(Right_anode),  
.Display_3(Left_out),  
.Display_4(Right_out),  
.sel(clk_display""));
Clock_Divider #(7) M6
  .clk_out(clk_slow),
  .clk_in(clk),
  .reset(reset));

Clock_Divider #(20) M7
  .clk_out(clk_display),
  .clk_in(clk),
  .reset(reset));

Toggle M8
  .toggle_out (read_fifo),
  .toggle_in (read_synch),
  .clk(clk_slow),
  .reset(reset));

Synchro_2 M9
  .synchro_out(read_synch),
  .synchro_in(read),
  .clk(clk_slow),
  .reset(reset));
endmodule

module Row_Signal (Row, Key, Col);
  output [3: 0] Row;
  input [15: 0] Key;
  input [3: 0] Col;
  reg Row;

// Scan for row of the asserted key
always @ (Key or Col) begin  // Asynchronous behavior for key assertion
  || Key[3] & Col[3];
  || Key[7] & Col[3];
  || Key[15] & Col[3];
end
endmodule

// Using the testbench modules given below, verify the functionality of the integrated system. Pay careful attention to the formation of the graphic user interface (GUI) displaying waveforms to display results in a user-friendly format.
```verilog
// Test Bench
module TOP_keypad_FIFO();
wire [5: 0] state;
wire [6: 0] Cathode;
wire [3: 0] Col;
wire Left_anode, Right_anode;
wire valid;
wire empty;
wire full;
wire [3: 0] Row;
reg read;
reg clock, reset;
reg [15: 0] Key;
integer j, k;
reg [39: 0] Pressed;
parameter stack_width = 4;
parameter [39: 0] Key_0 = "Key_0";
parameter [39: 0] Key_1 = "Key_1";
parameter [39: 0] Key_2 = "Key_2";
parameter [39: 0] Key_3 = "Key_3";
parameter [39: 0] Key_4 = "Key_4";
parameter [39: 0] Key_5 = "Key_5";
parameter [39: 0] Key_6 = "Key_6";
parameter [39: 0] Key_7 = "Key_7";
parameter [39: 0] Key_8 = "Key_8";
parameter [39: 0] Key_9 = "Key_9";
parameter [39: 0] Key_A = "Key_A";
parameter [39: 0] Key_B = "Key_B";
parameter [39: 0] Key_C = "Key_C";
parameter [39: 0] Key_D = "Key_D";
parameter [39: 0] Key_E = "Key_E";
parameter [39: 0] Key_F = "Key_F";
parameter [39: 0] None = "None";
wire [stack_width-1:0] stack0 = UUT_M2.stack[0]; // Probes of the stack
wire [stack_width-1:0] stack1 = UUT_M2.stack[1];
wire [stack_width-1:0] stack2 = UUT_M2.stack[2];
wire [stack_width-1:0] stack3 = UUT_M2.stack[3];
wire [stack_width-1:0] stack4 = UUT_M2.stack[4];
wire [stack_width-1:0] stack5 = UUT_M2.stack[5];
wire [stack_width-1:0] stack6 = UUT_M2.stack[6];
wire [stack_width-1:0] stack7 = UUT_M2.stack[7];

always @ (Key) begin
  case (Key)
    j:
      Pressed = stack0;
    k:
      Pressed = stack1;
    l:
      Pressed = stack2;
    m:
      Pressed = stack3;
    n:
      Pressed = stack4;
    o:
      Pressed = stack5;
    p:
      Pressed = stack6;
    q:
      Pressed = stack7;
  endcase
end
```
16'h0000: Pressed = None;
16'h0001: Pressed = Key_0;
16'h0002: Pressed = Key_1;
16'h0004: Pressed = Key_2;
16'h0008: Pressed = Key_3;
16'h0010: Pressed = Key_4;
16'h0020: Pressed = Key_5;
16'h0040: Pressed = Key_6;
16'h0080: Pressed = Key_7;
16'h1000: Pressed = Key_8;
16'h2000: Pressed = Key_9;
16'h4000: Pressed = Key_A;
16'h8000: Pressed = Key_B;
16'h1000: Pressed = Key_C;
16'h2000: Pressed = Key_D;
16'h4000: Pressed = Key_E;
16'h8000: Pressed = Key_F;

default: Pressed = None;
endcase
end

TOP_Keypad_FIFO UUT
(Cathode, Col, Left_anode, Right_anode, valid, empty, full, Row, read, clock, reset);

Row_Signal M2(Row, Key, Col);

initial #42000 $finish;
initial begin clock = 0; forever #5 clock = ~clock; end
initial begin reset = 1; #10 reset = 0; end
initial begin for (k = 0; k <= 16; k = k+1) begin
  Key[] = 1; #180 Key = 0; end end

initial begin forever begin
  #307 read = 1;
  #20 read = 0;
end
end
endmodule
Design—Prototype Synthesis and Implementation

Synthesize the integrated system and target the design into a Xilinx Spartan-10XL FPGA. For the Digilab-XLA prototyping board. Download the bitmap file into the Spartan-10XL FPGA and conduct a demonstration of the functionality of the system.

12. Modify the keypad scanner circuit from the previous exercises to incorporate protection against switch bounce. Consider requiring a switched input to hold its value for a sufficiently long time before the machine accepts the input (e.g., 20 ms). Write and verify a Verilog model of the modified circuit. Synthesize the modified circuit and verify that the debounce circuitry works on the prototyping board.

13. FPGA-Based Design Exercise: Serial Communications Link with Error Correction

The objective of this exercise is to implement a serial communication link between a pair of FPGA prototyping boards and to demonstrate the functionality of an error correction unit. The UARTS that were presented in Chapter 7 are to be used, and include an extended Hamming encoder and an extended Hamming decoder. The block diagram in Figure P6-13a shows the configuration of the transmitter board and the receiver board. At the transmitter board, a sender interacts with the FPGA by pressing a key of the keypad. The keypad decoder, together with a two-stage synchronizer, produces a 4-bit code corresponding to the pressed key. The Hamming encoder accepts a 4-bit input and generates an 8-bit encoded output word; a pair of push buttons can be used to deliberately inject errors into the code, for subsequent decoding and error correction at the receiver board. The UART transmitter will send the output of the Error Injection Unit.

The Error Injection Unit is to be hard wired to corrupt bits 1 and 5, depending on whether the push-button switches are pressed. The data stream bits can be XOR-ed with the logic value presented by the condition of the pushbutton switch. When not pressed, the switch presents a logical 0. The selected bits allow a data bit and a parity bit to be corrupted.

The Hamming decoder at the receiver board accepts an 8-bit word and forms a 4-bit output word. The unit is to be implemented with combinational logic and operate fast enough to form its output in a single cycle of the clock. The decoder is to detect and correct a single-bit error and display the corrected data word on the seven-segment displays. An LED will also be illuminated to indicate the occurrence of such an error. The extended Hamming code includes an additional bit to allow the decoder to detect, but not correct, the occurrence of a double-bit error. Such a condition will be indicated by illumination of another LED.

The Hex keypad interface is to form a unique code for each pressed key. A clock divider forms additional clock signals from the board’s nominal 50 MHz clock signal. The information presented to the set of seven-segment displays

\[\text{See Sections 5.16 and 5.17.}\]
FIGURE P8-13 Serial communication link between prototyping boards: (a) transmitter unit, and (b) receiver unit.
will have to be time-multiplexed. The clock signals to be used in the design are shown in Table P8-13a.

Develop, verify, and synthesize the functional units supporting the serial communications link. Synthesize the top modules that are to reside on the transmitter and receiver boards. Discuss the resources required to support the design.

14. Using an FPGA synthesis tool, synthesize RISC_SPM (see Section 7.3). Discuss the machine's performance and its use of FPGA resources.

<table>
<thead>
<tr>
<th>Clock</th>
<th>Frequency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>50 MHz</td>
<td>Input Clock</td>
</tr>
<tr>
<td>clk40</td>
<td>3.125 MHz</td>
<td>Keypad Interface clock</td>
</tr>
<tr>
<td>clk200</td>
<td>3.125 MHz</td>
<td>Seven-segment Display clock</td>
</tr>
<tr>
<td>clk_rx</td>
<td>25 MHz</td>
<td>UART Receiver clock</td>
</tr>
<tr>
<td>clk_tx</td>
<td>3.125 MHz</td>
<td>UART Transmitter clock</td>
</tr>
</tbody>
</table>