Chapter 7 Memory and Programmable Logic

CHAPTER OBJECTIVES

1. Know the organizational structure and functionality of programmable logic devices (PLDs).
2. Know how array logic diagrams differ from conventional logic diagrams.
3. Know the letters that are used to refer to the number of words in a memory.
4. Know how to write an HDL description of a memory.
5. Know how to interpret memory cycle timing waveforms.
6. Given the capacity and word size of a memory, know how to specify the number of its address and data lines.
7. Know how to use a Hamming code to detect and correct a single error, and to detect a double error.
8. Be able to write a truth table for a ROM.
9. Be able to write a programming table for a PLA.
10. Be able to write a programming table for a PAL.
11. Know the basic architecture of a field-programmable gate array (FPGA).
12. Know the circuit for a programmable interconnect point in a FPGA.
13. Know the difference between block RAM and distributed RAM in a FPGA.
14. Be able to write an HDL model of a RAM.

7.1 INTRODUCTION

A memory unit is a device to which binary information is transferred for storage and from which information is retrieved when needed for processing. When data processing takes place, information from memory is transferred to selected registers in the processing unit. Intermediate and final results obtained in the processing unit are transferred back to be stored in memory. Binary information received from an input device is stored in memory, and information transferred to an output device is taken from memory. A memory unit is a collection of cells capable of storing a large quantity of binary information.

There are two types of memories that are used in digital systems: *random-access memory* (RAM) and *read-only memory* (ROM). RAM stores new information for later use. The process of storing new information into memory is referred to as a memory *write* operation. The process of transferring the stored information out of memory is referred to as a memory *read* operation. RAM can perform both write and read operations. ROM can perform only the read operation. This means that suitable binary information is already stored inside memory and can be retrieved or read at any time. However, that information cannot be altered by writing.

ROM is a *programmable logic device* (PLD). The binary information that is stored within such a device is specified in some fashion and then embedded within the hardware in a process referred to as *programming* the device. The word “programming” here refers to a hardware procedure, which specifies the bits that are inserted into the hardware configuration of the device.

ROM is one example of a PLD. Other such units are the programmable logic array (PLA), programmable array logic (PAL), and the field-programmable gate array (FPGA). A PLD is an integrated circuit with internal logic gates connected through electronic paths that behave similarly to fuses. In the original state of the device, all the fuses are intact. Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function. In this chapter, we introduce the configuration of PLDs and indicate procedures for their use in the design of digital systems.

We also present CMOS FPGAs, which are configured by downloading a stream of bits into the device to configure transmission gates to establish the internal connectivity required by a specified logic function (combinational or sequential).

A typical PLD may have hundreds to millions of gates interconnected through hundreds to thousands of internal paths. In order to show the internal logic diagram of such a device in a concise form, it is necessary to employ a special gate symbology applicable to array logic. Figure 7.1 shows the conventional and array logic symbols for a multiple-input OR gate. Instead of having multiple input lines into the gate, we draw a single line entering the gate. The input lines are drawn perpendicular to this single line and are connected to the gate through internal fuses. In a similar fashion, we can draw the array logic for an AND gate. This type of graphical representation for the inputs of gates will be used throughout the chapter in array logic diagrams.

FIGURE 7.1

Conventional and array logic diagrams for OR gate

7.2 RANDOM-ACCESS MEMORY

A memory unit is a collection of storage cells, together with associated circuits needed to transfer information into and out of a device. The architecture of memory is such that information can be selectively retrieved from any of its internal locations. The time it takes to transfer information to or from any desired random location is always the same—hence the name *random-access memory,* abbreviated RAM. In contrast, the time required to retrieve information that is stored on magnetic tape depends on the location of the data.

A memory unit stores binary information in groups of bits called *words*. A word in memory is a set of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction, one or more alphanumeric characters, or any other binary-coded information. A group of 8 bits is called a *byte*. Most computer memories use words that are multiples of 8 bits in length. Thus, a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of a memory unit is usually stated as the total number of bytes that the unit can store.

Communication between memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer. A block diagram of a memory unit is shown in Fig. 7.2. The *n* data input lines provide the information to be stored in memory, and the *n* data output lines supply the information coming out of memory. The *k* address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: The *Write* input causes binary data to be transferred into the memory, and the *Read* input causes binary data to be transferred out of memory.

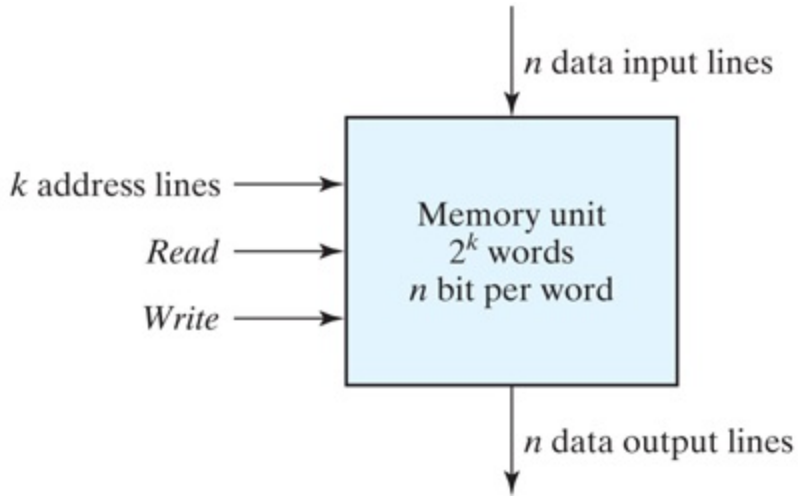


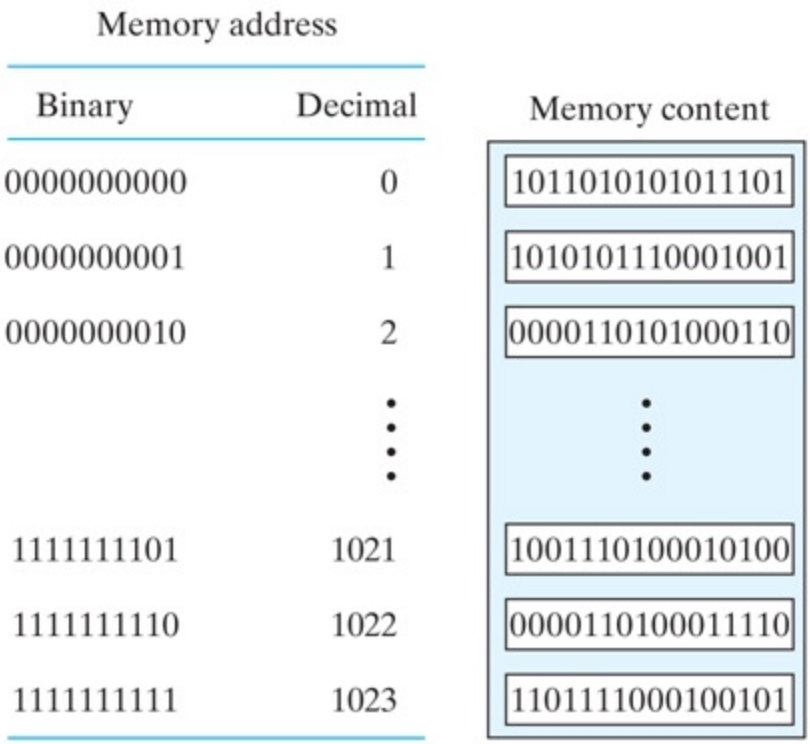
FIGURE 7.2

Block diagram of a memory unit

The memory unit is specified by the number of words it contains and the number of bits in each word. The address lines select one particular word. Each word in memory is assigned an identification number, called an *address*, starting from 0 up to 2k−1, where *k* is the number of address lines. The selection of a specific word inside memory is done by applying the *k*-bit address to the address lines. An internal decoder accepts this address and opens the paths needed to select the word specified. Memories vary greatly in size and may range from 1,024 words, requiring an address of 10 bits, to 232 words, requiring 32 address bits. It is customary to refer to the number of words (or bytes) in memory with one of the letters K (kilo), M (mega), and G (giga). K is equal to 210, M is equal to 220, and G is equal to 230. Thus, 64K=216, 2M=221, and 4G=232.

Consider, for example, a memory unit with a capacity of 1K words of 16 bits each. Since 1K=1,024=210 and 16 bits constitute two bytes, we can say that the memory can accommodate 2,048=2K bytes. Figure 7.3 shows possible contents of the first three and the last three words of this memory. Each word contains 16 bits that can be divided into two bytes. The words are recognized by their decimal address from 0 to 1,023. The equivalent binary address consists of 10 bits. The first address is specified with ten 0’s; the last address is specified with ten 1’s, because 1,023 in binary is equal to 1111111111. A word in memory is selected by its binary address. When a word is read or written, the memory operates on all 16 bits as a

single unit.



Memory address		Memory content
Binary	Decimal	
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
⋮	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

FIGURE 7.3

Contents of a 1024×16 memory

The 1K×16 memory of Fig. 7.3 has 10 bits in the address and 16 bits in each word. As another example, a 64K×10 memory will have 16 bits in the address (since 64K=216) and each word will consist of 10 bits. The number of address bits needed in a memory is dependent on the total number of words that can be stored in the memory and is independent of the number of bits in each word. The number of bits in the address is determined from the relationship $2k \geq m$, where *m* is the total number of words and *k* is the number of address bits needed to satisfy the relationship.

Write and Read Operations

The two operations that RAM can perform are the write and read operations. As alluded to earlier, the write signal specifies a transfer-in

operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired operation.

The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Apply the binary address of the desired word to the address lines.
2. Apply the data bits that must be stored in memory to the data input lines.
3. Activate the *write* input.

The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Apply the binary address of the desired word to the address lines.
2. Activate the *read* input.

The memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The contents of the selected word do not change after the read operation, that is, the read operation is nondestructive.

Commercial memory components available in integrated-circuit chips sometimes provide the two control inputs for reading and writing in a somewhat different configuration. Instead of having separate read and write inputs to control the two operations, most integrated circuits provide two other control inputs: One input selects the unit and the other determines the operation. The memory operations that result from these control inputs are specified in Table 7.1.

Table 7.1 *Control Inputs to Memory Chip*

Memory Enable	Read/Write	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

The memory enable (sometimes called the chip select) is used to enable the particular memory chip in a multichip implementation of a large memory. When the memory enable is inactive, the memory chip is not selected and no operation is performed. When the memory enable input is active, the read/write input determines the operation to be performed.

Memory Description in HDL

HDLs model memory by an array of words.

Verilog

A memory in Verilog is declared with a **reg** keyword, using a two-dimensional array. The first number specified in the array determines the number of bits in a word (the *word length*), and the second gives the number of words in memory (memory *depth*). For example, a memory of 1,024 words with 16 bits per word is declared as

```
reg [ 15: 0 ] memword [ 0: 1023 ];
```

This statement describes a two-dimensional array of 1,024 registers, each containing 16 bits. The second array range in the declaration of *memword* specifies the address range of the total number of words in memory; a specific value addresses a word of memory. For example, *memword[512]* refers to the 16-bit memory word at address 512. The individual bits in a

memory cannot be addressed directly. Instead, a word must be read from memory and assigned to a one-dimensional array; then a bit or a part-select[1] can be read from the word.

[1] A part-select is a contiguous range of bits.

VHDL

A memory in VHDL is modeled as an array of bit vectors or std_logic vectors. For example, a memory of 512 16-bit words can be declared as:

type RAM_512×16 is array (0 to 511) of bit (0 to 15)

The operation of a simple memory unit is illustrated in HDL Example 7.1. The memory has 64 words of four bits each. There are two control inputs: *Enable* and *ReadWrite*. The *DataIn* and *DataOut* lines have four bits each. The input *Address* must have six bits (since 26=64). The memory is declared with *Mem* used as an identifier that can be referenced with an index to access any of the 64 words. A memory operation requires that the *Enable* input be active. The *ReadWrite* input determines the type of operation. If *ReadWrite* is 1, the memory performs a read operation symbolized by the statement

DataOut ← Mem [Address];

Execution of this statement causes a transfer of four bits from the selected memory word specified by *Address* onto the *DataOut* lines. If *ReadWrite* is 0, the memory performs a write operation symbolized by the statement

Mem [Address] ← DataIn;

Execution of this statement causes a transfer from the four-bit *DataIn* lines into the memory word selected by *Address*. When *Enable* is equal to 0, the memory is disabled and the outputs are assumed to be in a high-impedance state, indicated by the symbol **z**. Thus, the memory has three-state outputs.

HDL Example 7.1

Verilog

```
// Read and write operations of memory
```

```
// Memory size is 64 words of four bits each.

module memory (Enable, ReadWrite, Address, DataIn, DataOut);
 input  Enable, ReadWrite;
 input  [3: 0] DataIn;
 input  [5: 0] Address;
 output [3: 0] DataOut;
 reg    [3: 0] DataOut;
 reg    [3: 0] Mem [0: 63];                        // 64 x 4 memor

always @ (Enable or ReadWrite or DataIn)
  if (Enable) begin
   if (ReadWrite) DataOut = Mem [Address];        // Read
   else Mem [Address] = DataIn;                   // Write
  else DataOut = 4'bz;
  end                                             // High impedan
endmodule
```

VHDL

```
// Read and write operations of memory
// Memory size is 64 words of four bits each

entity memory is
port (Enable, Readwrite: in, Std_Logic DataIn: in Std_Logic_Vec
  Address: in Std_Logic_Vector (5 downto 0);
  DataOut: out Std_Logic_Vector (3 downto 0));
end memory;

architecture Behavioral of memory is
begin
process (Enable, ReadWrite, DataIn) begin
 if Enable = 1 then
  if ReadWrite = 1 then DataOut <= Mem(address);
  else memory(address) <= DataIn; end if;
 else DataOut <= "zzzz"; end if;
endprocess;
end Behavioral;
```

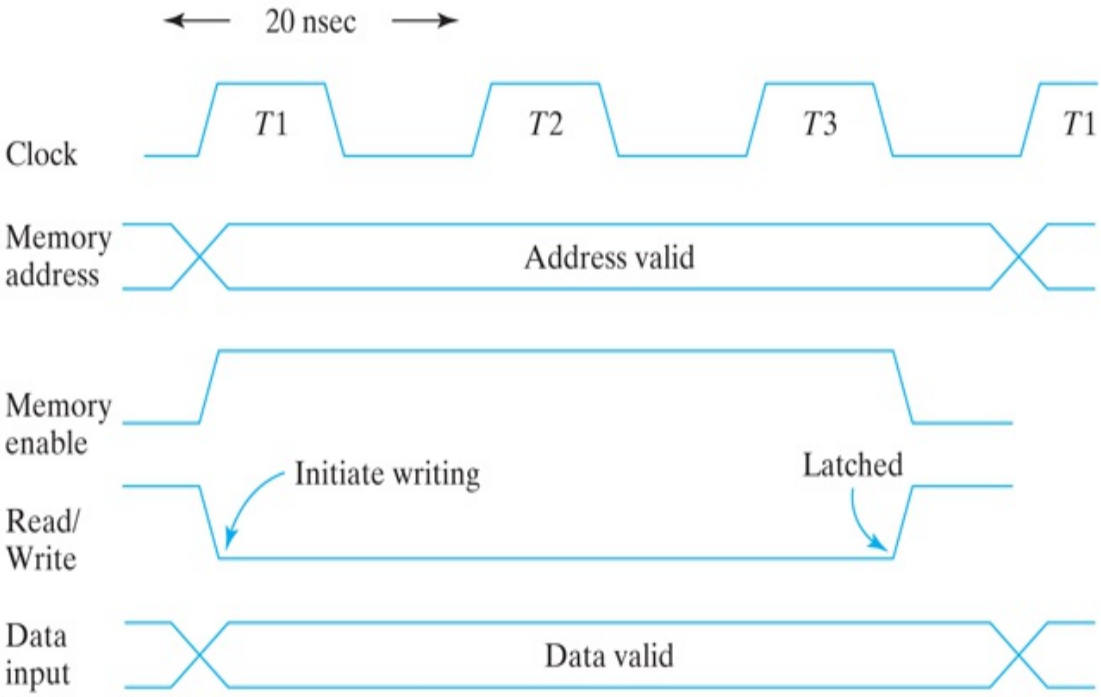
Timing Waveforms

The operation of the memory unit is controlled by an external device such as a central processing unit (CPU). The CPU is usually synchronized by its own clock. The memory, however, does not employ an internal clock. Instead, its read and write operations are specified by control inputs. The *access time* of memory is the time required to select a word and read it. The *cycle time* of memory is the time required to complete a write operation. The CPU must provide the memory control signals in such a

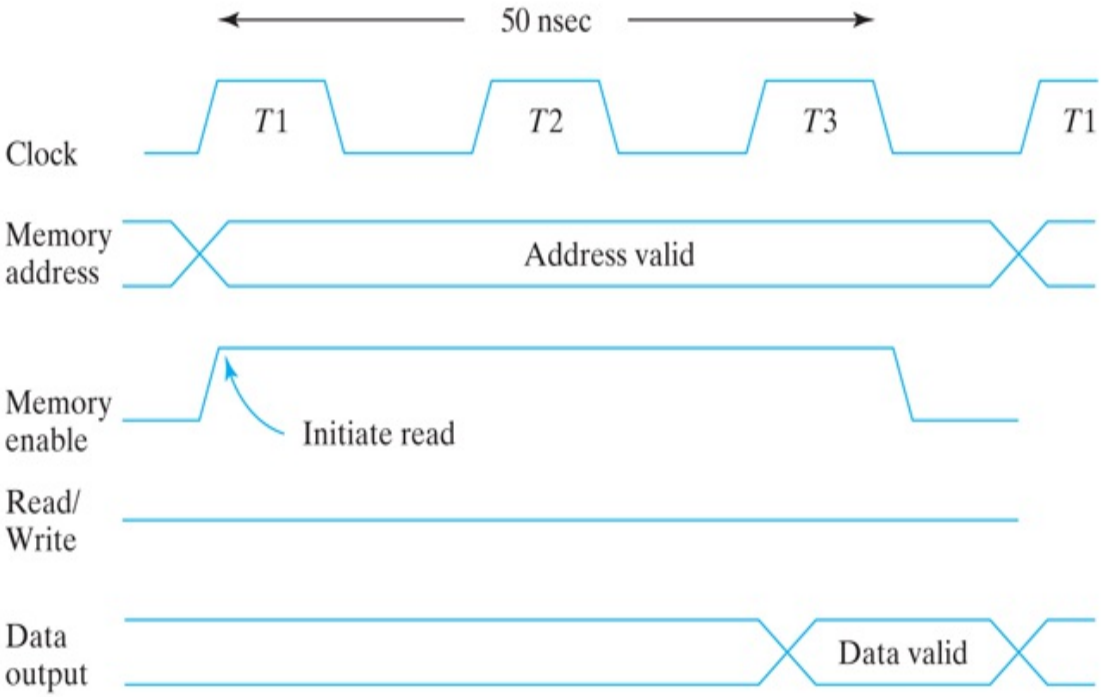
way as to synchronize its internal clocked operations with the read and write operations of memory. This means that the access time and cycle time of the memory must be within a time equal to a fixed number of CPU clock cycles.

Suppose as an example that a CPU operates with a clock frequency of 50 MHz, giving a period of 20 ns for one clock cycle. Suppose also that the CPU communicates with a memory whose access time and cycle time do not exceed 50 ns. This means that the write cycle terminates the storage of the selected word within a 50 ns interval and that the read cycle provides the output data of the selected word within 50 ns or less. (The two numbers are not always the same.) Since the period of the CPU cycle is 20 ns, it will be necessary to devote at least two-and-a-half, and possibly three, clock cycles for each memory request.

The memory timing shown in Fig. 7.4 is for a CPU with a 50 MHz clock and a memory with 50 ns maximum cycle time. The write cycle in part (a) shows three 20 ns cycles: *T*1, *T*2, and *T*3. For a write operation, the CPU must provide the address and input data to the memory. This is done at the beginning of *T*1. (The two lines that cross each other in the address and data waveforms designate a possible change in value of the multiple lines.) The memory enable and the read/write signals must be activated after the signals in the address lines are stable in order to avoid destroying data in other memory words. The memory enable signal switches to the high level and the read/write signal switches to the low level to indicate a write operation. The two control signals must stay active for at least 50 ns. The address and data signals must remain stable for a short time after the control signals are deactivated. At the completion of the third clock cycle, the memory write operation is completed and the CPU can access the memory again with the next *T*1 cycle.



(a) Write cycle



(b) Read cycle

FIGURE 7.4

Memory cycle timing waveforms

Description

The read cycle shown in Fig. 7.4(b) has an address for the memory provided by the CPU. The memory enable and read/write signals must be in their high level for a read operation. The memory places the data of the word selected by the address into the output data lines within a 50 ns interval (or less) from the time that the memory enable is activated. The CPU can transfer the data into one of its internal registers during the negative transition of *T*3. The next *T*1 cycle is available for another memory request.

Types of Memories

The mode of access of a memory system is determined by the type of components used. In a random-access memory, the word locations may be thought of as being separated in space, each word occupying one particular location. In a sequential-access memory, the information stored in some medium is not immediately accessible, but is available only at certain intervals of time. A magnetic disk or tape unit is of this type. Each memory location passes the read and write heads in turn, but information is read out only when the requested word has been reached. In a random-access memory, the access time is always the same regardless of the particular location of the word. In a sequential-access memory, the time it takes to access a word depends on the position of the word with respect to the position of the read head; therefore, the access time is variable.

Integrated circuit RAM units are available in two operating modes: *static* and *dynamic*. Static RAM (SRAM) consists essentially of internal latches that store the binary information. The stored information remains valid as long as power is applied to the unit. Dynamic RAM (DRAM) stores the binary information in the form of electric charges on capacitors provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time, and the capacitors must be periodically recharged by *refreshing* the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. DRAM offers reduced power consumption and larger storage capacity in a single memory chip. SRAM is easier to use and has shorter

read and write cycles.

Memory units that lose stored information when power is turned off are said to be *volatile*. CMOS integrated circuit RAMs, both static and dynamic, are of this category, since the binary cells need external power to maintain the stored information. In contrast, a nonvolatile memory, such as magnetic disk, retains its stored information after the removal of power. This type of memory is able to retain information because the data stored on magnetic components are represented by the direction of magnetization, which is retained after power is turned off. ROM is another nonvolatile memory. A nonvolatile memory enables digital computers to store programs that will be needed again after the computer is turned on. Programs and data that cannot be altered are stored in ROM, while other large programs are maintained on magnetic disks. The latter programs are transferred into the computer RAM as needed. Before the power is turned off, the binary information from the computer RAM is transferred to the disk so that the information will be retained. Ferroelectric RAM technology (FeRAM) is relatively new, and it also provides designers with a viable option for including nonvolatile memory in a design.

7.3 MEMORY DECODING

In addition to requiring storage components in a memory unit, there is a need for decoding circuits to select the memory word specified by the input address. In this section, we present the internal construction of a RAM and demonstrate the operation of the decoder. To be able to include the entire memory in one diagram, the memory unit presented here has a small capacity of 16 bits, arranged in four words of 4 bits each. An example of a two-dimensional coincident decoding arrangement is presented to show a more efficient decoding scheme that is used in large memories. We then give an example of address multiplexing commonly used in DRAM integrated circuits.

Internal Construction

The internal construction of a RAM of *m* words and *n* bits per word consists of m×n binary storage cells and associated decoding circuits for selecting individual words. The binary storage cell is the basic building block of a memory unit. The equivalent logic of a binary cell that stores one bit of information is shown in Fig. 7.5. The storage part of the cell is modeled by an *SR* latch with associated gates to form a *D* latch. Actually, the cell is an electronic circuit with four to six transistors. Nevertheless, it is possible and convenient to model it in terms of logic symbols. A binary storage cell must be very small in order to be able to pack as many cells as possible in the small area available in the integrated circuit chip. The binary cell stores one bit in its internal latch. The select input enables the cell for reading or writing, and the read/write input determines the operation of the cell when it is selected. A 1 in the read/write input provides the read operation by forming a path from the latch to the output terminal. A 0 in the read/write input provides the write operation by forming a path from the input terminal to the latch.

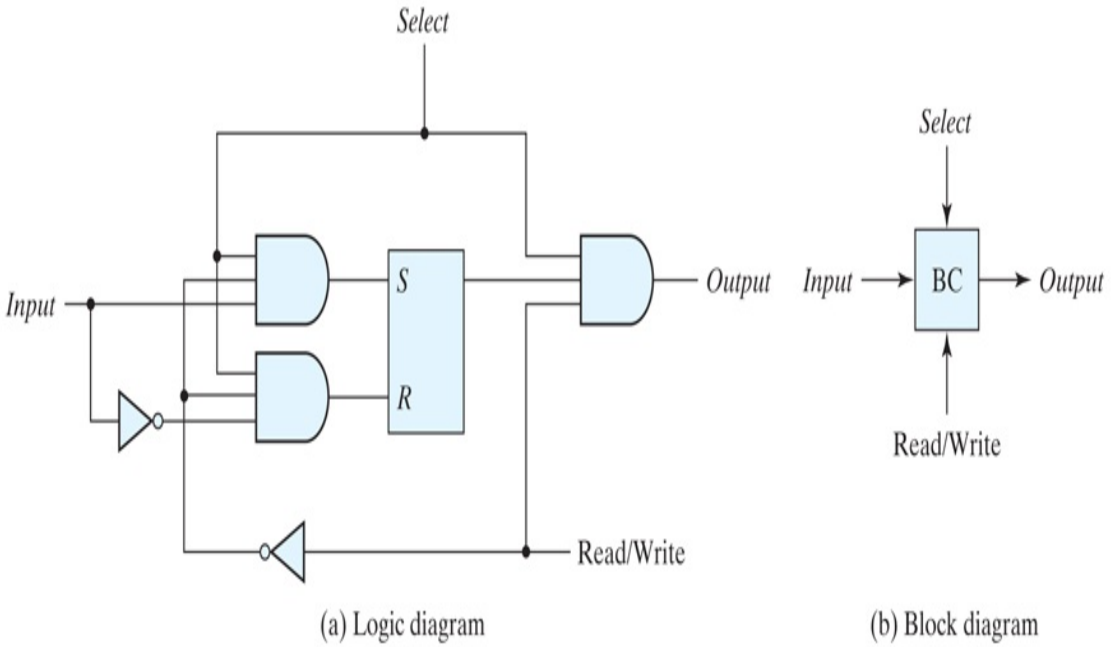


FIGURE 7.5

Memory cell

Description

The logical construction of a small RAM is shown in Fig. 7.6. This RAM consists of four words of four bits each and has a total of 16 binary cells. The small blocks labeled BC represent the binary cell with its three inputs and one output, as specified in Fig. 7.5(b). A memory with four words needs two address lines. The two address inputs go through a 2×4 decoder to select one of the four words. The decoder is enabled with the memory enable input. When the memory enable is 0, all outputs of the decoder are 0 and none of the memory words are selected. With the memory select at 1, one of the four words is selected, dictated by the value in the two address lines. Once a word has been selected, the read/write input determines the operation. During the read operation, the four bits of the selected word go through OR gates to the output terminals. (Note that the OR gates are drawn according to the array logic established in Fig. 7.1.) During the write operation, the data available in the input lines are transferred into the four binary cells of the selected word. The binary cells that are not selected are disabled, and their previous binary values remain unchanged. When the memory select input that goes into the decoder is equal to 0, none of the words are selected and the contents of all cells

remain unchanged regardless of the value of the read/write input.

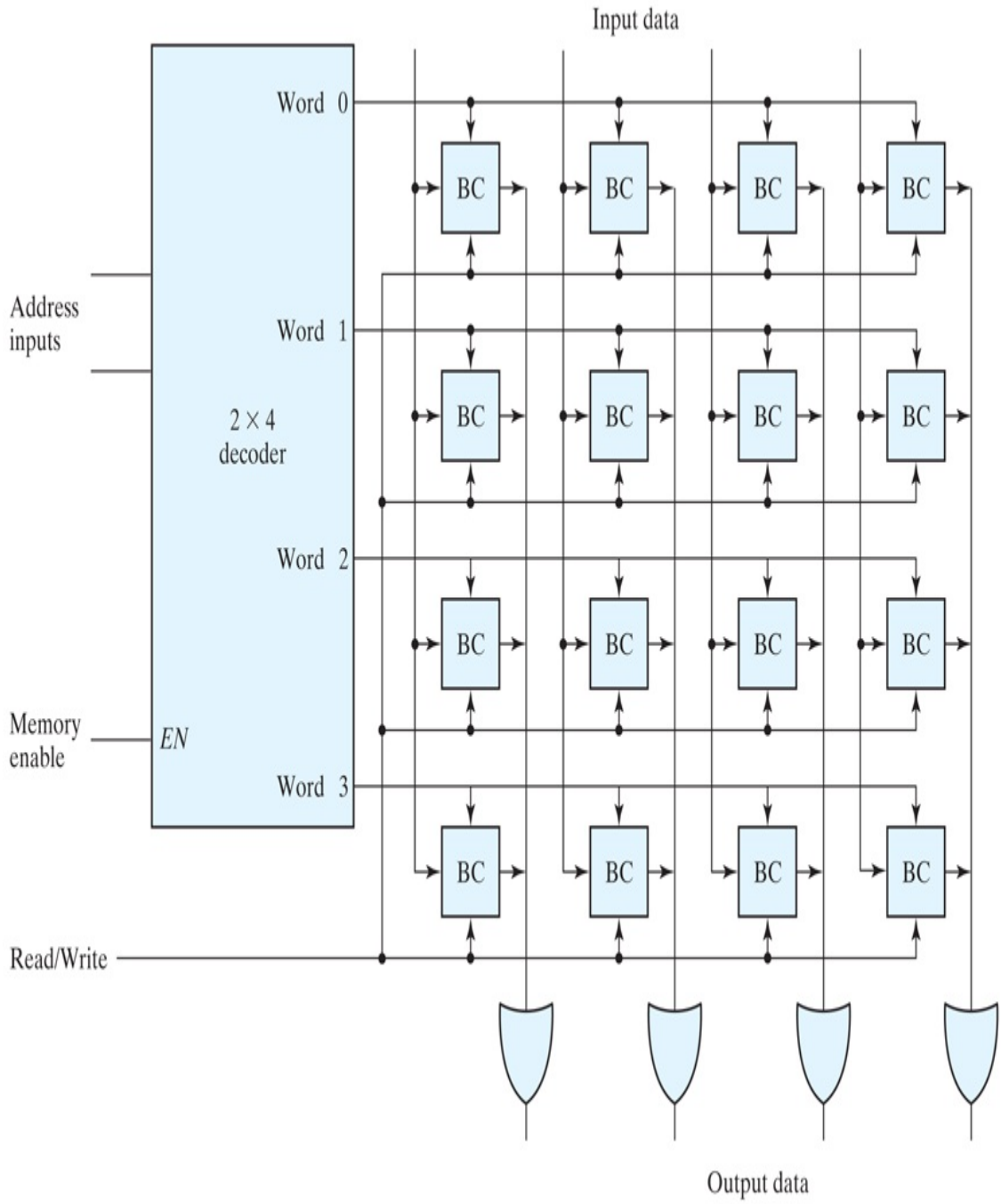


FIGURE 7.6

Diagram of a 4×4 RAM

Description

Commercial RAMs may have a capacity of thousands of words, and each word may range from 1 to 64 bits. The logical construction of a large-capacity memory would be a direct extension of the configuration shown here. A memory with 2k words of *n* bits per word requires *k* address lines that go into a k×2k decoder. Each one of the decoder outputs selects one word of *n* bits for reading or writing.

Coincident Decoding

A decoder with *k* inputs and 2k outputs requires 2k AND gates with *k* inputs per gate. The total number of gates and the number of inputs per gate can be reduced by employing two decoders in a two-dimensional selection scheme. The basic idea in two-dimensional decoding is to arrange the memory cells in an array that is close as possible to square. In this configuration, two *k*/2-input decoders are used instead of one *k*-input decoder. One decoder performs the row selection and the other the column selection in a two-dimensional matrix configuration.

The two-dimensional selection pattern is demonstrated in Fig. 7.7 for a 1K-word memory. Instead of using a single 10×1,024 decoder, we use two 5×32 decoders. With the single decoder, we would need 1,024 AND gates with 10 inputs in each. In the two-decoder case, we need 64 AND gates with 5 inputs in each. The five most significant bits of the address go to input *X* and the five least significant bits go to input *Y*. Each word within the memory array is selected by the coincidence of one *X* line and one *Y* line. Thus, each word in memory is selected by the coincidence between 1 of 32 rows and 1 of 32 columns, for a total of 1,024 words. Note that each intersection represents a word that may have any number of bits.

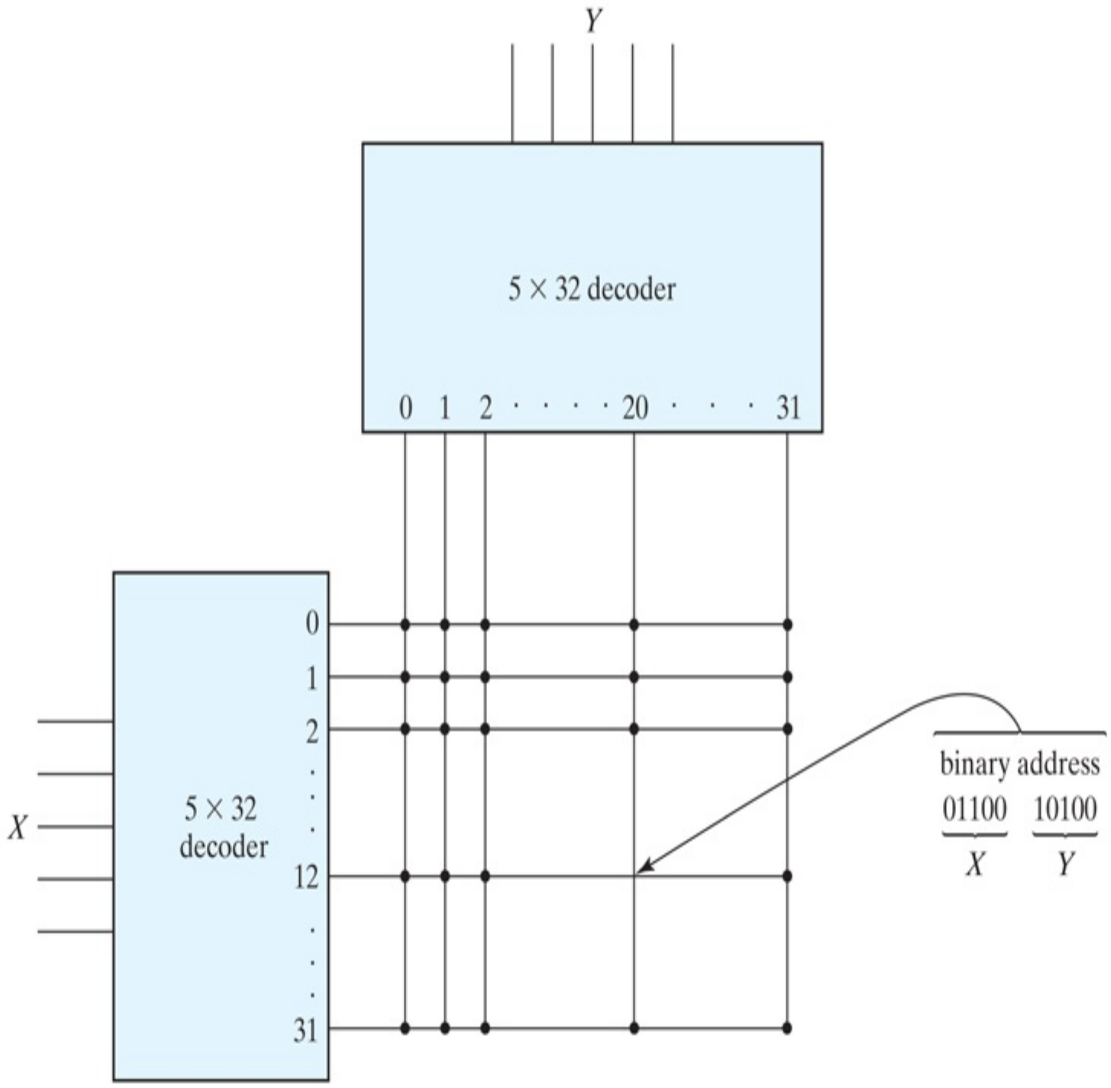


FIGURE 7.7

Two-dimensional decoding structure for a 1K-word memory

Description

As an example, consider the word whose address is 404. The 10-bit binary equivalent of 404 is 01100 10100. This makes X=01100 (binary 12) and Y=10100 (binary 20). The *n*-bit word that is selected lies in the *X* decoder output number 12 and the *Y* decoder output number 20. All the bits of the word are selected for reading or writing.

Address Multiplexing

The SRAM memory cell modeled in Fig. 7.5 typically contains six transistors. In order to build memories with higher density, it is necessary to reduce the number of transistors in a cell. The DRAM cell contains a single MOS transistor and a capacitor. The charge stored on the capacitor discharges with time, and the memory cells must be periodically recharged by refreshing the memory. Because of their simple cell structure, DRAMs typically have four times the density of SRAMs. This allows four times as much memory capacity to be placed on a given size of chip. The cost per bit of DRAM storage is three to four times less than that of SRAM storage. A further (operational) cost savings is realized because of the lower power requirement of DRAM cells. These advantages make DRAM the preferred technology for large memories in personal digital computers. DRAM chips are available in capacities from 64K to 512M bits. Most DRAMs have a 1-bit word size, so several chips have to be combined to produce a larger word size.

Because of their large capacity, the address decoding of DRAMs is arranged in a two-dimensional array, and larger memories often have multiple arrays. To reduce the number of pins in the IC package, designers utilize address multiplexing whereby one set of address input pins accommodates the address components. In a two-dimensional array, the address is applied in two parts at different times, with the row address first and the column address second. Since the same set of pins is used for both parts of the address, the size of the package is decreased significantly.

We will use a 64K-word memory to illustrate the address-multiplexing idea. A diagram of the decoding configuration is shown in Fig. 7.8. The memory consists of a two-dimensional array of cells arranged into 256 rows by 256 columns, for a total of $28 \times 28 = 216 = 64K$ words. There is a single data input line, a single data output line, and a read/write control, as well as an eight-bit address input and two address *strobes,* the latter included for enabling the row and column address into their respective registers. The row address strobe (RAS) enables the eight-bit row register, and the column address strobe (CAS) enables the eight-bit column register. The bar on top of the name of the strobe symbol indicates that the registers are enabled on the zero level of the signal.

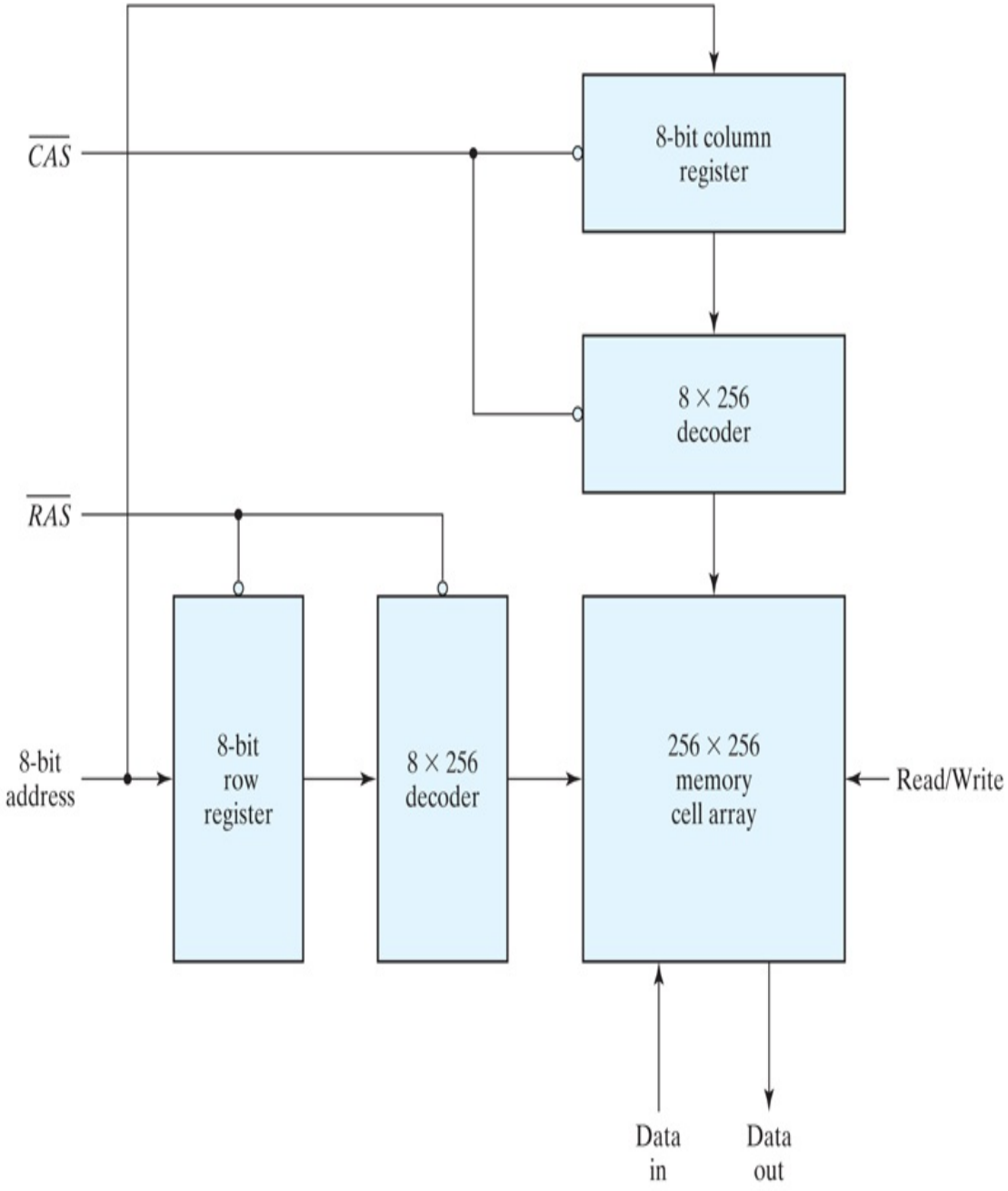


FIGURE 7.8

Address multiplexing for a 64K DRAM

Description

The 16-bit address is applied to the DRAM in two steps using RAS and CAS. Initially, both strobes are in the 1 state. The 8-bit row address is applied to the address inputs and RAS is changed to 0. This loads the row

address into the row address register. RAS also enables the row decoder so that it can decode the row address and select one row of the array. After a time equivalent to the settling time of the row selection, RAS goes back to the 1 level. The 8-bit column address is then applied to the address inputs, and CAS is driven to the 0 state. This transfers the column address into the column register and enables the column decoder. Now the two parts of the address are in their respective registers, the decoders have decoded them to select the one cell corresponding to the row and column address, and a read or write operation can be performed on that cell. CAS must go back to the 1 level before initiating another memory operation.

7.4 ERROR DETECTION AND CORRECTION

The dynamic physical interaction of the electrical signals affecting the data path of a memory unit may cause occasional errors in storing and retrieving the binary information. The reliability of a memory unit may be improved by employing error-detecting and error-correcting codes. The most common error detection scheme is the parity bit. (See Section 3.8.) A parity bit is generated and stored along with the data word in memory. The parity of the word is checked after reading it from memory. The data word is accepted if the parity of the bits read out is correct. If the parity checked results in an inversion, an error is detected, but it cannot be corrected.

An error-correcting code generates multiple parity check bits that are stored with the data word in memory. Each check bit is a parity over a group of bits in the data word. When the word is read back from memory, the associated parity bits are also read from memory and compared with a new set of check bits generated from the data that have been read. If the check bits are correct, no error has occurred. If the check bits do not match the stored parity, they generate a unique pattern, called a *syndrome*, that can be used to identify the bit that is in error. A single error occurs when a bit changes in value from 1 to 0 or from 0 to 1 during the write or read operation. If the specific bit in error is identified, then the error can be corrected by complementing the erroneous bit.

Hamming Code

One of the most common error-correcting codes used in RAMs was devised by R. W. Hamming. In the Hamming code, *k* parity bits are added to an *n*-bit data word, forming a new word of n+k bits. The bit positions are numbered in sequence from 1 to n+k. Those positions numbered as a power of 2 are reserved for the parity bits. The remaining bits are the data bits. The code can be used with words of any length. Before giving the general characteristics of the code, we will illustrate its operation with a data word of eight bits.

Consider, for example, the 8-bit data word 11000100. We include 4 parity bits with the 8-bit word and arrange the 12 bits as follows:

Bit position: 1 2 3 4 5 6 7 8 9 10 11 12

P1 P2 1 P4 1 0 0 P8 0 1 0 0

The 4 parity bits, P1, P2, P4, and P8, are in positions 1, 2, 4, and 8, respectively. The 8 bits of the data word are in the remaining positions. Each parity bit is calculated as follows:

P1 = XOR of bits (3, 5, 7, 9, 11)=1⊕1⊕0⊕0⊕0=0 P2 =
XOR of bits (3, 6, 7, 10, 11)=1⊕0⊕0⊕1⊕0=0 P4 =
XOR of bits (5, 6, 7, 12)=1⊕0⊕0⊕0=1 P8 =
XOR of bits (9, 10, 11, 12)=0⊕1⊕0⊕0=1

Remember that the exclusive-OR operation performs the odd function: It is equal to 1 for an odd number of 1's in the variables and to 0 for an even number of 1's. Thus, each parity bit is set so that the total number of 1's in the checked positions, including the parity bit, is always even.

The 8-bit data word is stored in memory together with the 4 parity bits as a 12-bit composite word. Substituting the 4 *P* bits in their proper positions, we obtain the 12-bit composite word stored in memory:

0 0 1 1 1 0 0 1 0 1 0 0

Bit position: 1 2 3 4 5 6 7 8 9 10 11 12

When the 12 bits are read from memory, they are checked again for errors. The parity is checked over the same combination of bits, including the parity bit. The 4 check bits are evaluated as follows:

C1 = XOR of bits (1, 3, 5, 7, 9, 11) C2 = XOR of bits (2, 3, 6, 7, 10, 11)
C4 = XOR of bits (4, 5, 6, 7, 12) C8 = XOR of bits (8, 9, 10, 11, 12)

A 0 check bit designates even parity over the checked bits and a 1 designates odd parity. Since the bits were stored with even parity, the result, C=C8C4C2C1=0000, indicates that no error has occurred. However, if C≠0, then the 4-bit binary number formed by the check bits gives the position of the erroneous bit. For example, consider the following three cases:

Bit position: 1 2 3 4 5 6 7 8 9 10 11 12

0 0 1 1 1 0 0 1 0 1 0 0 No error

1 0 1 1 1 0 0 1 0 1 0 0 Error in bit 1

0 0 1 1 0 0 0 1 0 1 0 0 Error in bit 5

In the first case, there is no error in the 12-bit word. In the second case, there is an error in bit position number 1 because it changed from 0 to 1. The third case shows an error in bit position 5, with a change from 1 to 0. Evaluating the XOR of the corresponding bits, we determine the 4 check bits to be as follows:

C8 C4 C2 C1

For no error: 0 0 0 0

With error in bit 1: 0 0 0 1

With error in bit 5: 0 1 0 1

Thus, for no error, we have C=0000; with an error in bit 1, we obtain C=0001; and with an error in bit 5, we get C=0101. When the binary

number *C* is not equal to 0000, it gives the position of the bit in error. The error can be corrected by complementing the corresponding bit. Note that an error can occur in the data word or in one of the parity bits.

The Hamming code can be used for data words of any length. In general, the Hamming code consists of *k* check bits and *n* data bits, for a total of n+k bits. The syndrome value *C* consists of *k* bits and has a range of 2k values between 0 and 2k−1. One of these values, usually zero, is used to indicate that no error was detected, leaving 2k−1 values to indicate which of the n+k bits was in error. Each of these 2k−1 values can be used to uniquely describe a bit in error. Therefore, the range of *k* must be equal to or greater than n+k, giving the relationship

$2k-1 \geq n+k$

Solving for *n* in terms of *k*, we obtain:

$2k-1-k \geq n$

This relationship gives a formula for establishing the number of data bits that can be used in conjunction with *k* check bits. For example, when k=3, the number of data bits that can be used is $n \leq (2^3-1-3)=4$. For k=4, we have $2^4-1-4=11$, giving $n \leq 11$. The data word may be less than 11 bits, but must have at least 5 bits; otherwise, only 3 check bits will be needed. This justifies the use of 4 check bits for the 8 data bits in the previous example. Ranges of *n* for various values of *k* are listed in Table 7.2.

Table 7.2 *Range of Data Bits for k Check Bits*

Number of Check Bits, *k*	Range of Data Bits, *n*
3	2–4
4	5–11

5	12–26
6	27–57
7	58–120

The grouping of bits for parity generation and checking can be determined from a list of the binary numbers from 0 through 2k−1. The least significant bit is a 1 in the binary numbers 1, 3, 5, 7, and so on. The second significant bit is a 1 in the binary numbers 2, 3, 6, 7, and so on. Comparing these numbers with the bit positions used in generating and checking parity bits in the Hamming code, we note the relationship between the bit groupings in the code and the position of the 1-bits in the binary count sequence. Note that each group of bits starts with a number that is a power of 2: 1, 2, 4, 8, 16, etc. These numbers are also the position numbers for the parity bits.

Single-Error Correction, Double-Error Detection

The Hamming code can detect and correct only a single error. By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors. If we include this additional parity bit, then the previous 12-bit coded word becomes 001110010100P13, where P13 is evaluated from the exclusive-OR of the other 12 bits. This produces the 13-bit word 0011100101001 (even parity). When the 13-bit word is read from memory, the check bits are evaluated, as is the parity *P* over the entire 13 bits. If P=0, the parity is correct (even parity), but if P=1, then the parity over the 13 bits is incorrect (odd parity). The following four cases can arise:

- If C=0 and P=0, no error occurred.
- If C≠0 and P=1, a single error occurred that can be corrected.

- If C≠0 and P=0, a double error occurred that is detected, but that cannot be corrected.

- If C=0 and P=1, an error occurred in the P13 bit.

This scheme may detect more than two errors, but is not guaranteed to detect all such errors.

Integrated circuits use a modified Hamming code to generate and check parity bits for single-error correction and double-error detection. The modified Hamming code uses a more efficient parity configuration that balances the number of bits used to calculate the XOR operation. A typical integrated circuit that uses an 8-bit data word and a 5-bit check word is IC type 74637. Other integrated circuits are available for data words of 16 and 32 bits. These circuits can be used in conjunction with a memory unit to correct a single error or detect double errors during write and read operations.

7.5 READ-ONLY MEMORY

A read-only memory (ROM) is essentially a memory device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. Once the pattern is established, it stays within the unit even when power is turned off and on again.

A block diagram of a ROM consisting of *k* inputs and *n* outputs is shown in Fig. 7.9. The inputs provide the address for memory, and the outputs give the data bits of the stored word that is selected by the address. The number of words in a ROM is determined from the fact that *k* address input lines are needed to specify 2k words. Note that a ROM does not have data inputs, because it does not have a write operation. Integrated circuit ROM chips have one or more enable inputs and sometimes come with three-state outputs to facilitate the construction of large arrays of ROM.

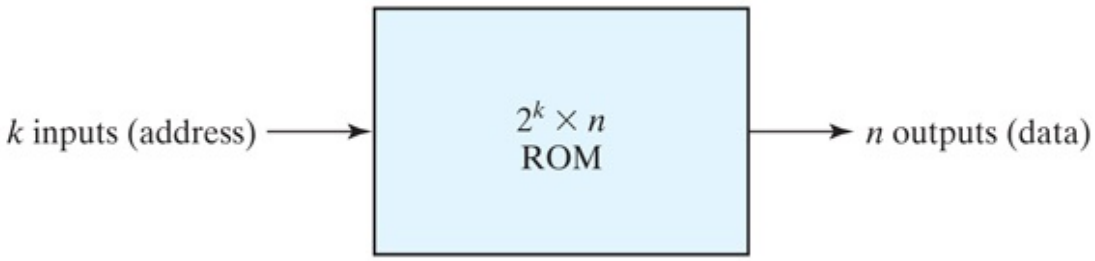


FIGURE 7.9

ROM block diagram

Consider, for example, a 32×8 ROM. The unit consists of 32 words of 8 bits each. There are five input lines that form the binary numbers from 0 through 31 for the address. Figure 7.10 shows the internal logic construction of this ROM. The five inputs are decoded into 32 distinct outputs by means of a 5×32 decoder. Each output of the decoder represents a memory address. The 32 outputs of the decoder are connected to each of the eight OR gates. The diagram shows the array logic convention used in complex circuits. (See Fig. 6.1.) Each OR gate must be considered as having 32 inputs. Each output of the decoder is connected to one of the

inputs of each OR gate. Since each OR gate has 32 input connections and there are 8 OR gates, the ROM contains 32×8=256 internal connections. In general, a 2k×n ROM will have an internal k×2k decoder and *n* OR gates. Each OR gate has 2k inputs, which are connected to each of the outputs of the decoder.

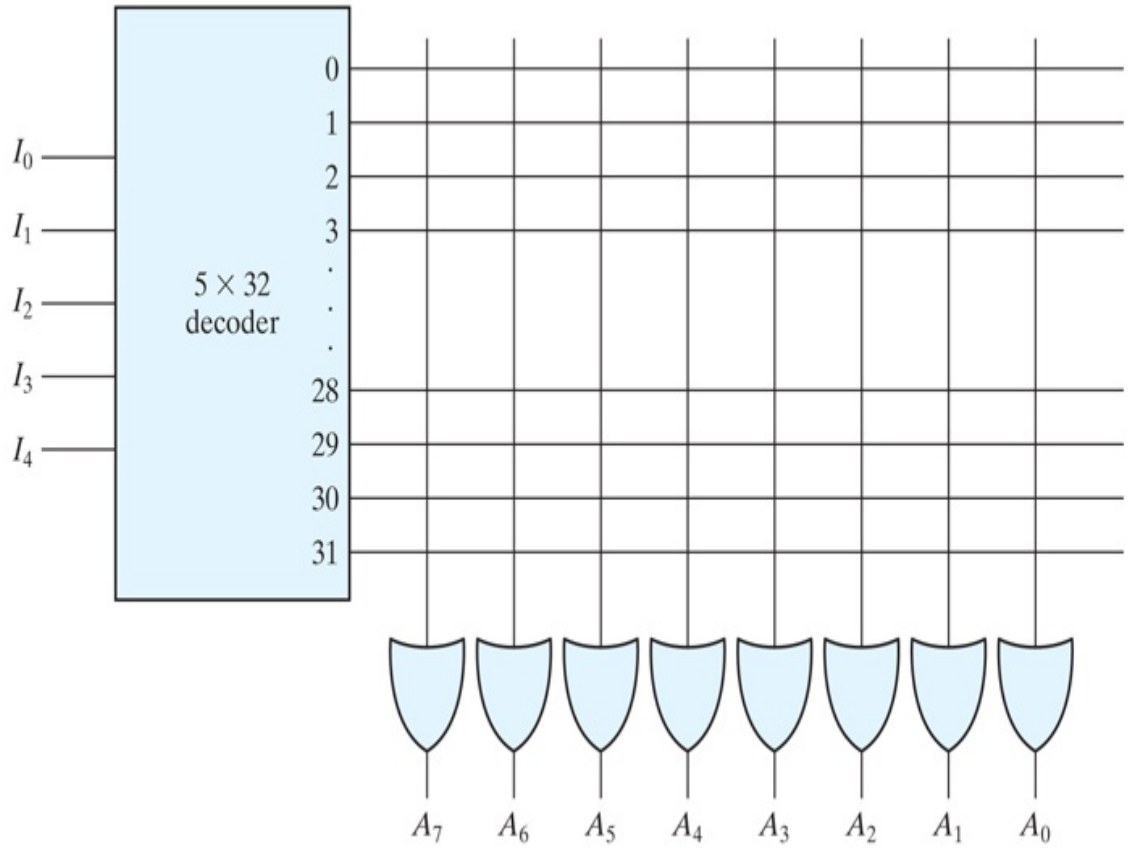


FIGURE 7.10

Internal logic of a 32×8 ROM

The 256 intersections in Fig. 7.10 are programmable. A programmable connection between two lines is logically equivalent to a switch that can be altered to be either closed (meaning that the two lines are connected) or open (meaning that the two lines are disconnected). The programmable intersection between two lines is sometimes called a *crosspoint*. Various physical devices are used to implement crosspoint switches. One of the simplest technologies employs a fuse that normally connects the two points, but is opened or “blown” by the application of a high-voltage pulse into the fuse.

The internal binary storage of a ROM is specified by a truth table that shows the word content in each address. For example, the content of a 32×8 ROM may be specified with a truth table similar to the one shown in Table 7.3. The truth table shows the five inputs under which are listed all 32 addresses. Each address stores a word of 8 bits, which is listed in the outputs columns. The table shows only the first four and the last four words in the ROM. The complete table must include the list of all 32 words.

Table 7.3 *ROM Truth Table (Partial)*

Inputs					**Outputs**							
I4	**I3**	**I2**	**I1**	**I0**	**A7**	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮						⋮				
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0

1 1 1 1 0 0 1 0 0 1 0 1 0

1 1 1 1 1 0 0 1 1 0 0 1 1

The hardware procedure that programs the ROM blows fuse links in accordance with a given truth table. For example, programming the ROM according to the truth table given by Table 7.3 results in the configuration shown in Fig. 7.11. Every 0 listed in the truth table specifies the absence of a connection, and every 1 listed specifies a path that is obtained by a connection. For example, the table specifies the eight-bit word 10110010 for permanent storage at address 3. The four 0's in the word are programmed by blowing the fuse links between output 3 of the decoder and the inputs of the OR gates associated with outputs A6, A3, A2, and A0. The four 1's in the word are marked with a×to denote a temporary connection, in place of a dot used for a permanent connection in logic diagrams. When the input of the ROM is 00011, all the outputs of the decoder are 0 except for output 3, which is at logic 1. The signal equivalent to logic 1 at decoder output 3 propagates through the connections to the OR gate outputs of A7, A5, A4, and A1. The other four outputs remain at 0. The result is that the stored word 10110010 is applied to the eight data outputs.

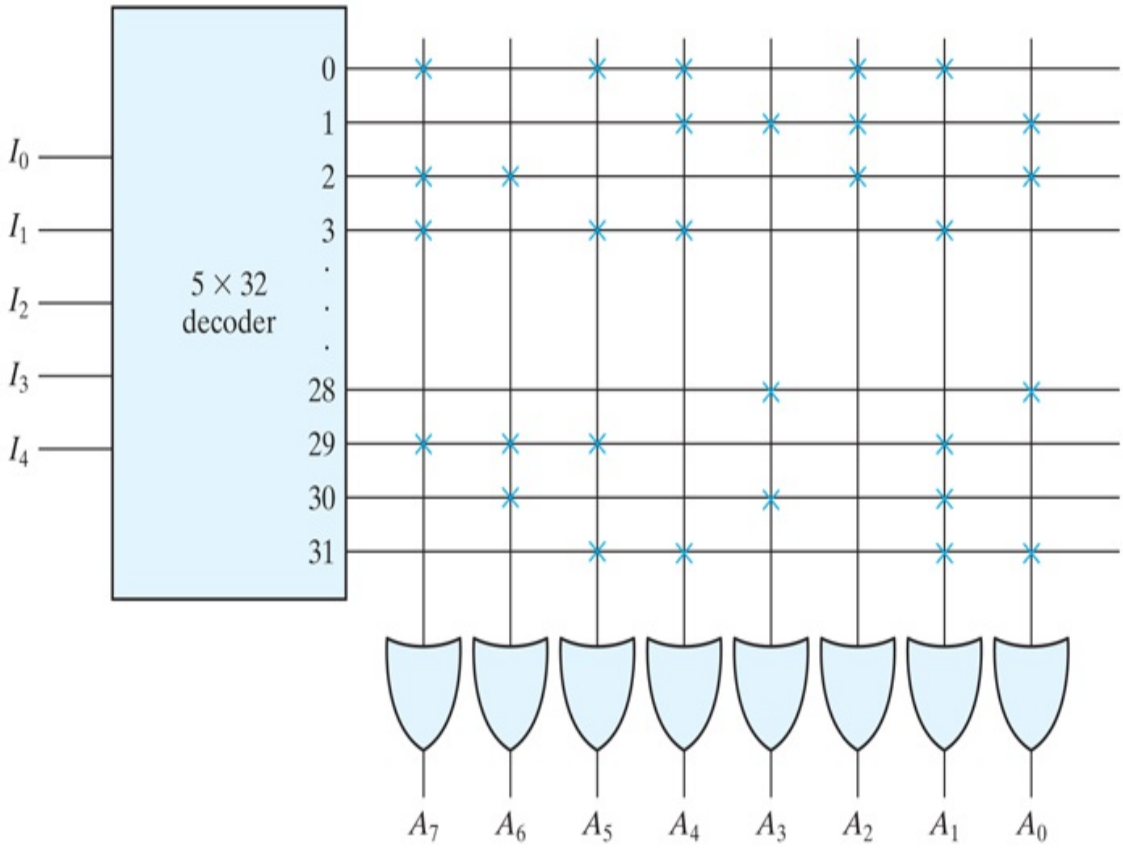


FIGURE 7.11

Programming the ROM according to Table 7.3

Combinational Circuit Implementation

In Section 4.9, it was shown that a decoder generates the 2k minterms of the *k* input variables. By inserting OR gates to sum the minterms of Boolean functions, we were able to generate any desired combinational circuit. The ROM is essentially a device that includes both the decoder and the OR gates within a single device to form a minterm generator. By choosing connections for those minterms which are included in the function, *the ROM outputs can be programmed to represent the Boolean functions of the output variables in a combinational circuit.*

The internal operation of a ROM can be interpreted in two ways. The first

interpretation is that of a memory unit that contains a fixed pattern of stored words. The second interpretation is that of a unit which implements a combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed as a sum of minterms. For example, the ROM of Fig. 7.11 may be considered to be a combinational circuit with eight outputs, each a function of the five input variables. Output A7 can be expressed in sum of minterms as

$$A7(I4, I3, I2, I1, I0)=\Sigma(0, 2, 3, \ldots, 29)$$

(The three dots represent minterms 4 through 27, which are not specified in the figure.) A connection marked with × in the figure produces a minterm for the sum. All other crosspoints are not connected and are not included in the sum.

In practice, when a combinational circuit is designed by means of a ROM, it is not necessary to design the logic or to show the internal gate connections inside the unit. All that the designer has to do is specify the particular ROM by its IC number and provide the applicable truth table. The truth table gives all the information for programming the ROM. No internal logic diagram is needed to accompany the truth table.

Example 7.1

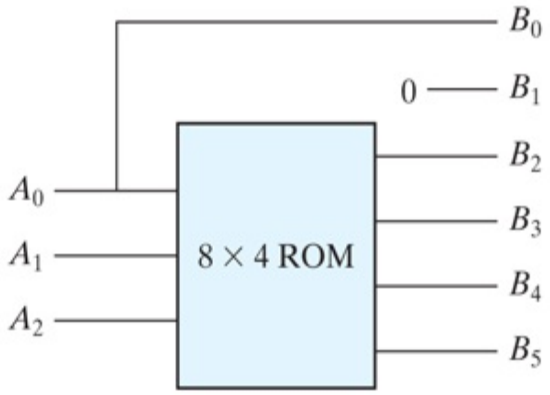
Design a combinational circuit using a ROM. The circuit accepts a three-bit number and outputs a binary number equal to the square of the input number.

The first step is to derive the truth table of the combinational circuit. In most cases, this is all that is needed. In other cases, we can use a partial truth table for the ROM by utilizing certain properties in the output variables. Table 7.4 is the truth table for the combinational circuit. Three inputs and six outputs are needed to accommodate all possible binary numbers. We note that output B0 is always equal to input A0, so there is no need to generate B0 with a ROM, since it is equal to an input variable. Moreover, output B1 is always 0, so this output is a known constant. We actually need to generate only four outputs with the ROM; the other two are readily obtained. The minimum size of ROM needed must have three inputs and four outputs. Three inputs specify eight words, so the ROM

must be of size 8×4. The ROM implementation is shown in Fig. 7.12. The three inputs specify eight words of four bits each. The truth table in Fig. 7.12(b) specifies the information needed for programming the ROM. The block diagram of Fig. 7.12(a) shows the required connections of the combinational circuit.

Table 7.4 *Truth Table for Circuit of Example 7.1*

Inputs			Outputs						
A2	**A1**	**A0**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**	**Decimal**
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49



(a) Block diagram

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

FIGURE 7.12

ROM implementation of Example 7.1

Description

Types of ROMs

The required paths in a ROM may be programmed in four different ways. The first is called *mask programming* and is done by the semiconductor company during the last fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table he or she wishes the ROM to satisfy. The truth table may be submitted in a special form provided by the manufacturer or in a specified format on a computer output medium. The manufacturer makes the corresponding mask for the paths to produce the 1's and 0's according to the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking the particular ROM. For this reason, mask programming is economical only if a large quantity of the same ROM configuration is to be ordered.

For small quantities, it is more economical to use a second type of ROM called *programmable read-only memory*, or PROM. When ordered, PROM units contain all the fuses intact, giving all 1's in the bits of the stored words. The fuses in the PROM are blown by the application of a

high-voltage pulse to the device through a special pin. A blown fuse defines a binary 0 state and an intact fuse gives a binary 1 state. This procedure allows the user to program the PROM in the laboratory to achieve the desired relationship between input addresses and stored words. Special instruments called PROM programmers are available commercially to facilitate the procedure. In any case, all procedures for programming ROMs are hardware procedures, even though the word *programming* is used.

The hardware procedure for programming ROMs or PROMs is irreversible, and once programmed, the fixed pattern is permanent and cannot be altered. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed. A third type of ROM is the *erasable PROM*, or EPROM, which can be restructured to the initial state even though it has been programmed previously. When the EPROM is placed under a special ultraviolet light for a given length of time, the shortwave radiation discharges the internal floating gates that serve as the programmed connections. After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of values.

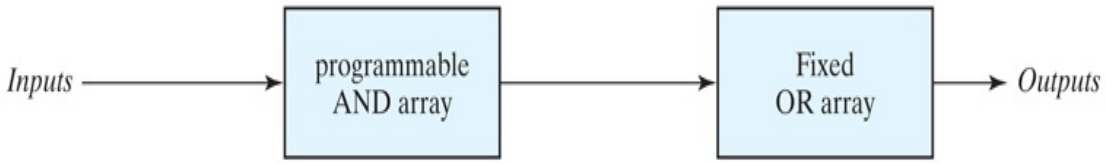
The fourth type of ROM is the electrically erasable PROM (EEPROM or E2PROM). This device is like the EPROM, except that the previously programmed connections can be erased with an electrical signal instead of ultraviolet light. The advantage is that the device can be erased without removing it from its socket.

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 in cell phones, digital cameras, set-top boxes, digital TV, telecommunications, nonvolatile data storage, and microcontrollers. Their low consumption of power makes them an attractive storage medium for laptop and notebook computers. Flash memories incorporate additional circuitry, too, allowing simultaneous erasing of blocks of memory, for example, of size 16–64 K bytes. Like EEPROMs, flash memories are subject to fatigue, typically having about 105 block erase cycles.

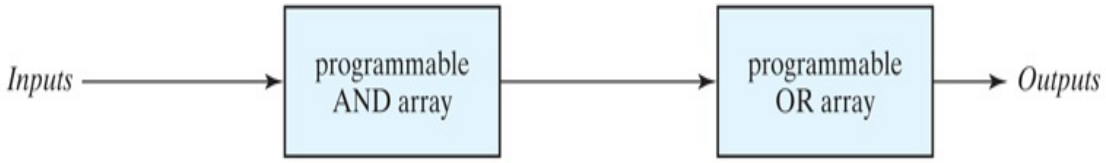
Combinational PLDs

The PROM is a combinational programmable logic device (PLD)—an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND–OR sum-of-product implementation. There are three major types of combinational PLDs, differing in the placement of the programmable connections in the AND–OR array. Figure 7.13 shows the configuration of the three PLDs. The PROM has a fixed AND array constructed as a decoder and a programmable OR array. The programmable OR gates implement the Boolean functions in sum-of-minterms form. The PAL has a programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate. The most flexible PLD is the PLA, in which both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum-of-products implementation. Historically, the names PAL and PLA emerged from different vendors during the development of PLDs. The implementation of combinational circuits with PROM was demonstrated in this section. The design of combinational circuits with PLA and PAL is presented in the next two sections.

(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL)



(c) Programmable logic array (PLA)

FIGURE 7.13

Basic configuration of three PLDs

Description

7.6 PROGRAMMABLE LOGIC ARRAY

The PLA is similar in concept to the PROM, except that the PLA does not provide full decoding of the variables and does not generate all the minterms. The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions.

The internal logic of a PLA with three inputs and two outputs is shown in Fig. 7.14. Such a circuit is too small to be useful commercially, but is presented here to demonstrate the typical logic configuration of a PLA. The diagram uses the array logic graphic symbols for complex circuits. Each input goes through a buffer–inverter combination, shown in the diagram with a composite graphic symbol, that has both the true and complement outputs. Each input and its complement are connected to the inputs of each AND gate, as indicated by the intersections between the vertical and horizontal lines. The outputs of the AND gates are connected to the inputs of each OR gate. The output of the OR gate goes to an XOR gate, where the other input can be programmed to receive a signal equal to either logic 1 or logic 0. The output is inverted when the XOR input is connected to 1 (since $x \oplus 1 = x'$). The output does not change when the XOR input is connected to 0 (since $x \oplus 0 = x$). The particular Boolean functions implemented in the PLA of Fig. 7.14 are:

$F1 = AB' + AC + A'BC'$ $F2 = (AC + BC)'$

The product terms generated in each AND gate are listed along the output of the gate in the diagram. The product term is determined from the inputs whose crosspoints are connected and marked with a ×. The output of an OR gate gives the logical sum of the selected product terms. The output may be complemented or left in its true form, depending on the logic being realized.

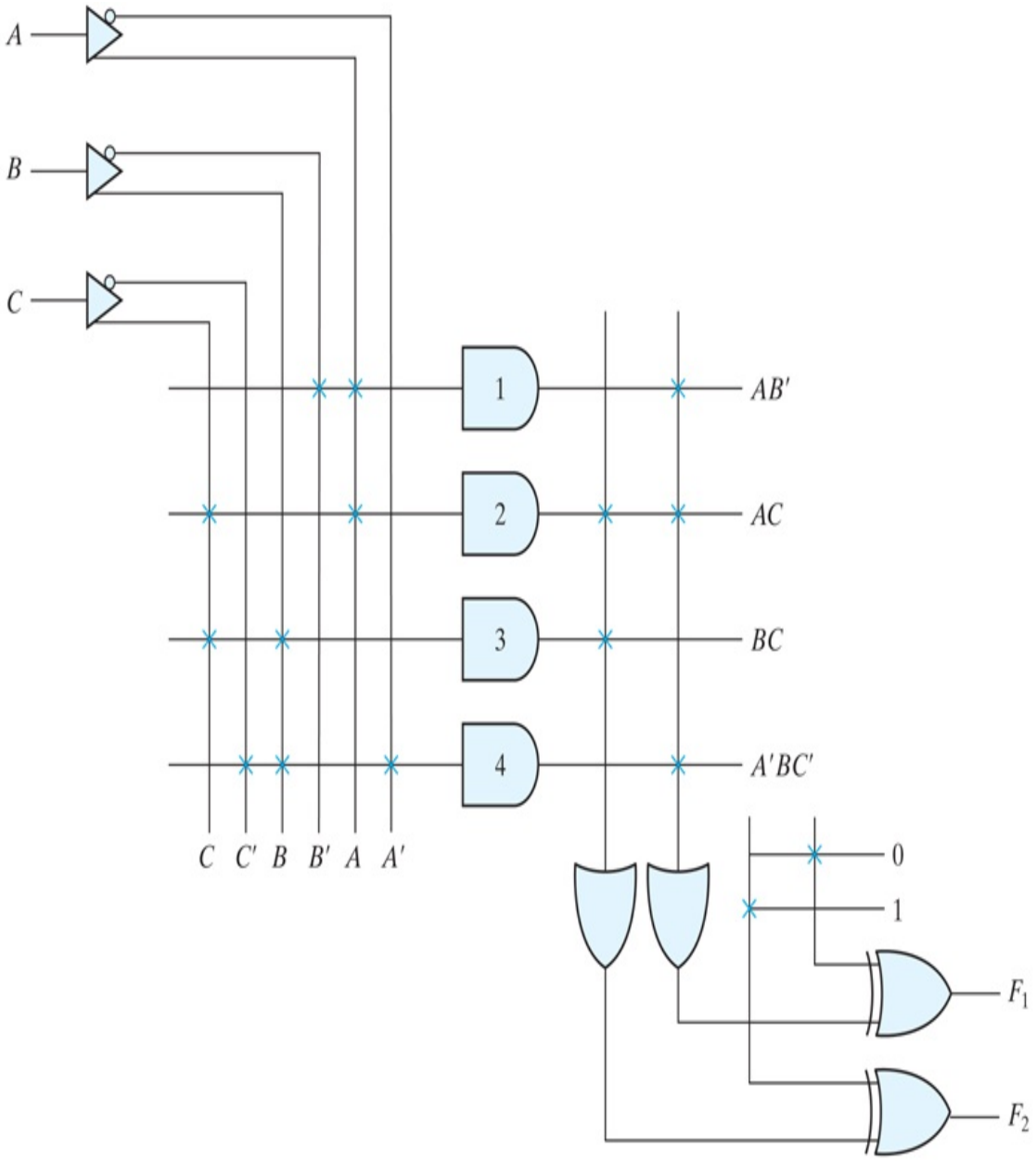


FIGURE 7.14

PLA with three inputs, four product terms, and two outputs

Description

The fuse map of a PLA can be specified in a tabular form. For example, the programming table that specifies the PLA of Fig. 7.14 is listed in Table 7.5. The PLA programming table consists of three sections. The first section lists the product terms numerically. The second section specifies the required paths between inputs and AND gates. The third section

specifies the paths between the AND and OR gates. For each output variable, we may have a T (for true) or C (for complement) for programming the XOR gate. The product terms listed on the left are not part of the table; they are included for reference only. For each product term, the inputs are marked with 1, 0, or — (dash). If a variable in the product term appears in the form in which it is true, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. If the variable is absent from the product term, it is marked with a dash.

Table 7.5 *PLA Programming Table*

		Inputs			Outputs (T)	Outputs (C)
Product Term		***A***	***B***	***C***	**F1**	**F2**
AB′	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
A′BC′	4	0	1	0	1	—

Note: See text for meanings of dashes.

The paths between the inputs and the AND gates are specified under the column head “Inputs” in the programming table. A 1 in the input column specifies a connection from the input variable to the AND gate. A 0 in the input column specifies a connection from the complement of the variable to the input of the AND gate. A dash specifies a blown fuse in both the input variable and its complement. It is assumed that an open terminal in the input of an AND gate behaves like a 1.

The paths between the AND and OR gates are specified under the column head “Outputs.” The output variables are marked with 1’s for those product terms which are included in the function. Each product term that has a 1 in the output column requires a path from the output of the AND gate to the input of the OR gate. Those marked with a dash specify a blown fuse. It is assumed that an open terminal in the input of an OR gate behaves like a 0. Finally, a T (true) output dictates that the other input of the corresponding XOR gate be connected to 0, and a C (complement) specifies a connection to 1.

The size of a PLA is specified by the number of inputs, the number of product terms, and the number of outputs. A typical integrated circuit PLA may have 16 inputs, 48 product terms, and eight outputs. For *n* inputs, *k* product terms, and *m* outputs, the internal logic of the PLA consists of *n* buffer–inverter gates, *k* AND gates, *m* OR gates, and *m* XOR gates. There are 2n×k connections between the inputs and the AND array, k×m connections between the AND and OR arrays, and *m* connections associated with the XOR gates.

In designing a digital system with a PLA, there is no need to show the internal connections of the unit as was done in Fig. 7.14. All that is needed is a PLA programming table from which the PLA can be programmed to supply the required logic. As with a ROM, the PLA may be mask programmable or field programmable. With mask programming, the customer submits a PLA program table to the manufacturer. This table is used by the vendor to produce a custom-made PLA that has the required internal logic specified by the customer. A second type of PLA that is available is the field-programmable logic array, or FPLA, which can be programmed by the user by means of a commercial hardware programmer unit.

In implementing a combinational circuit with a PLA, careful investigation must be undertaken in order to reduce the number of distinct product

terms, since a PLA has a finite number of AND gates. This can be done by simplifying each Boolean function to a minimum number of terms. The number of literals in a term is not important, since all the input variables are available anyway. Both the true value and the complement of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

Example 7.2

Implement the following two Boolean functions with a PLA:

F1(A, B, C)=Σ(0, 1, 2, 4) F2(A, B, C)=Σ(0, 5, 6, 7)

The two functions are simplified in the maps of Fig. 7.15. Both the true value and the complement of the functions are simplified into sum-of-products form. The combination that gives the minimum number of product terms is:

PLA programming table

	Product term	Inputs A B C	Outputs (C) F_1	(T) F_2
AB	1	1 1 –	1	1
AC	2	1 – 1	1	1
BC	3	– 1 1	1	–
$A'B'C'$	4	0 0 0	–	1

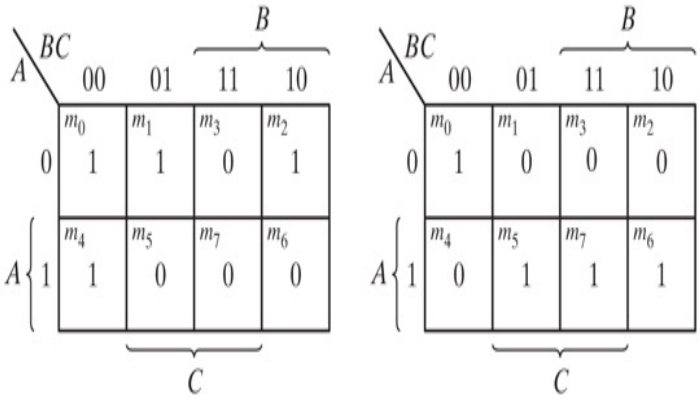


FIGURE 7.15

Solution to Example 7.2

Description
F1=(AB+AC+BC)′

and

F2=AB+AC+A′B′C′

This combination gives four distinct product terms: *AB*, *AC*, *BC*, and A′B′C′. The PLA programming table for the combination is shown in the figure. Note that output F1 is the true output, even though a *C* is marked over it in the table. This is because F1′ is generated with an AND–OR circuit and is available at the output of the OR gate. The XOR gate complements the function to produce the true F1 output.

The combinational circuit used in Example 7.2 is too simple for implementing with a PLA. It was presented merely for purposes of illustration. A typical PLA has a large number of inputs and product terms. The simplification of Boolean functions with so many variables should be carried out by means of computer-assisted simplification procedures. The computer-aided design (CAD) program simplifies each function and its complement to a minimum number of terms. The program then selects a minimum number of product terms that cover all functions in the form in which they are true or in their complemented form. The PLA programming table is then generated and the required fuse map obtained. The fuse map is applied to an FPLA programmer that goes through the hardware procedure of blowing the internal fuses in the integrated circuit.

7.7 PROGRAMMABLE ARRAY LOGIC

The PAL is a programmable logic device with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL is easier to program than, but is not as flexible as, the PLA. Figure 7.16 shows the logic configuration of a typical PAL with four inputs and four outputs. Each input has a buffer–inverter gate, and each output is generated by a fixed OR gate. There are four sections in the unit, each composed of an AND–OR array that is *three wide*, the term used to indicate that there are three programmable AND gates in each section and one fixed OR gate. Each AND gate has 10 programmable input connections, shown in the diagram by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple-input configuration of the AND gate. One of the outputs is connected to a buffer–inverter gate and then fed back into two inputs of the AND gates.

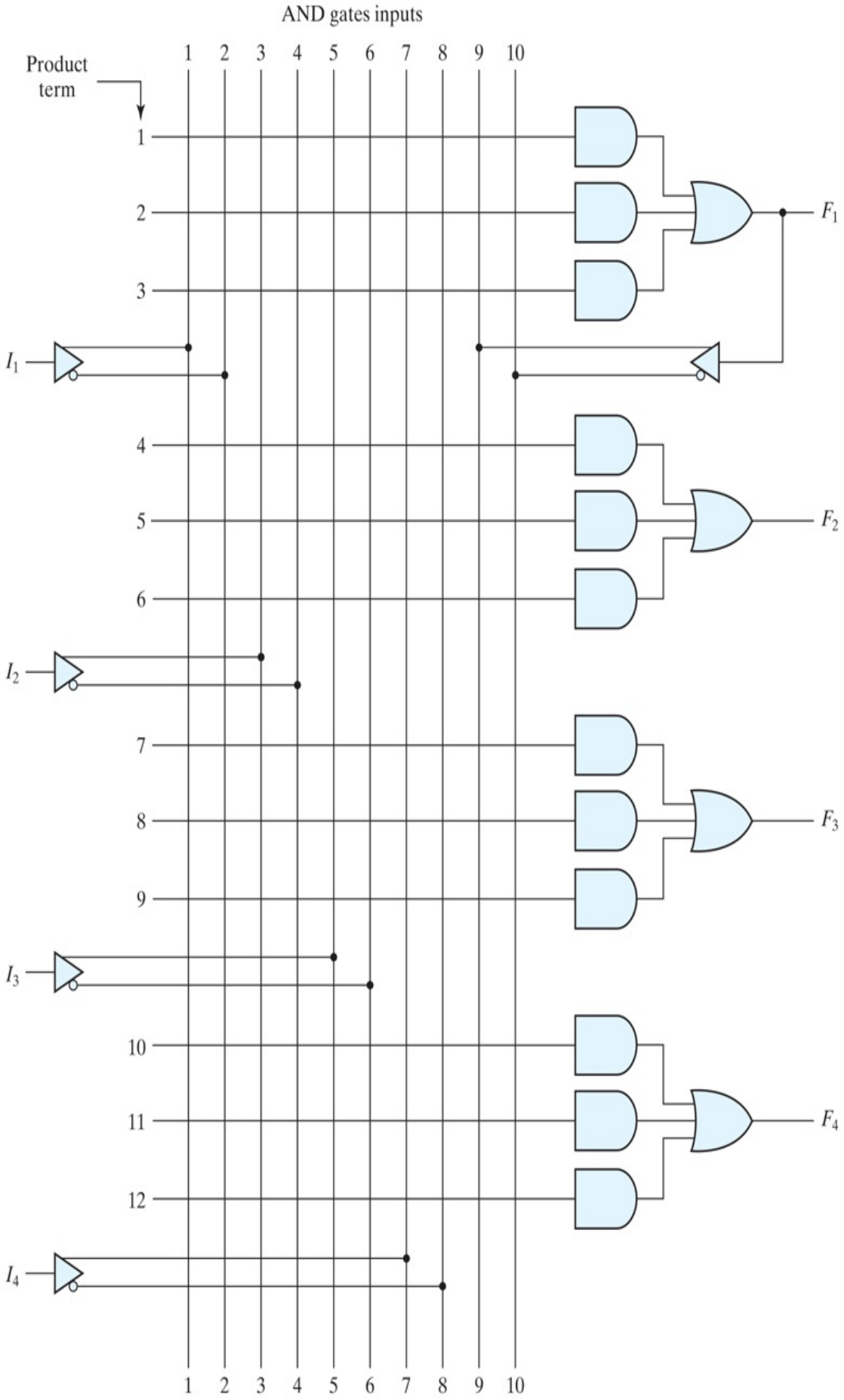

AND gates inputs
Product term
1 2 3 4 5 6 7 8 9 10
1
2
3
4
5
6
7
8
9
10
11
12
I_1
I_2
I_3
I_4
F_1
F_2
F_3
F_4

FIGURE 7.16

PAL with four inputs, four outputs, and a three-wide AND–OR structure

Description

Commercial PAL devices contain more gates than the one shown in Fig. 7.16. A typical PAL integrated circuit may have eight inputs, eight outputs, and eight sections, each consisting of an eight-wide AND–OR array. The output terminals are sometimes driven by three-state buffers or inverters.

In designing with a PAL, the Boolean functions must be simplified to fit into each section. Unlike the situation with a PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself, without regard to common product terms. The number of product terms in each section is fixed, and if the number of terms in the function is too large, it may be necessary to use two sections to implement one Boolean function.

As an example of using a PAL in the design of a combinational circuit, consider the following Boolean functions, given in sum-of-minterms form:

w(A, B, C, D) = Σ(2, 12, 13) x(A, B, C, D) =
Σ(7, 8, 9, 10, 11, 12, 13, 14, 15) y(A, B, C, D) =
Σ(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15) z(A, B, C, D) = Σ(1, 2, 8, 12, 13)

Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

w = ABC′+A′B′CD′ x = A+BCD y = A′B+CD+B′D′ z = ABC′+A′B′CD
′+AC′D′+A′B′C′D = w+AC′D′+A′B′C′D

Note that the function for *z* has four product terms. The logical sum of two of these terms is equal to *w*. By using *w*, it is possible to reduce the number of terms for *z* from four to three.

The PAL programming table is similar to the one used for the PLA, except

that only the inputs of the AND gates need to be programmed. Table 7.6 lists the PAL programming table for the four Boolean functions. The table is divided into four sections with three product terms in each, to conform to the PAL of Fig. 7.16. The first two sections need only two product terms to implement the Boolean function. The last section, for output *z*, needs four product terms. Using the output from *w*, we can reduce the function to three terms.

Table 7.6 *PAL Programming Table*

	AND Inputs					
Product Term	***A***	***B***	***C***	***D***	***w***	**Outputs**
1	1	1	0	—	—	w=ABC′+A′B′CD′
2	0	0	1	0	—	
3	—	—	—	—	—	
4	1	—	—	—	—	x=A+BCD
5	—	1	1	1	—	
6	—	—	—	—	—	
7	0	1	—	—	—	y=A′B+CD+B′D′

8	—	—	1	1	—	
9	—	0	—	0	—	
10	—	—	—	—	1	z=w+AC′D′+A′B′C′D
11	1	—	0	0	—	
12	0	0	0	1	—	

The fuse map for the PAL as specified in the programming table is shown in Fig. 7.17. For each 1 or 0 in the table, we mark the corresponding intersection in the diagram with the symbol for an intact fuse. For each dash, we mark the diagram with blown fuses in both the true and complement inputs. If the AND gate is not used, we leave all its input fuses intact. Since the corresponding input receives both the true value and the complement of each input variable, we have AA′=0 and the output of the AND gate is always 0.

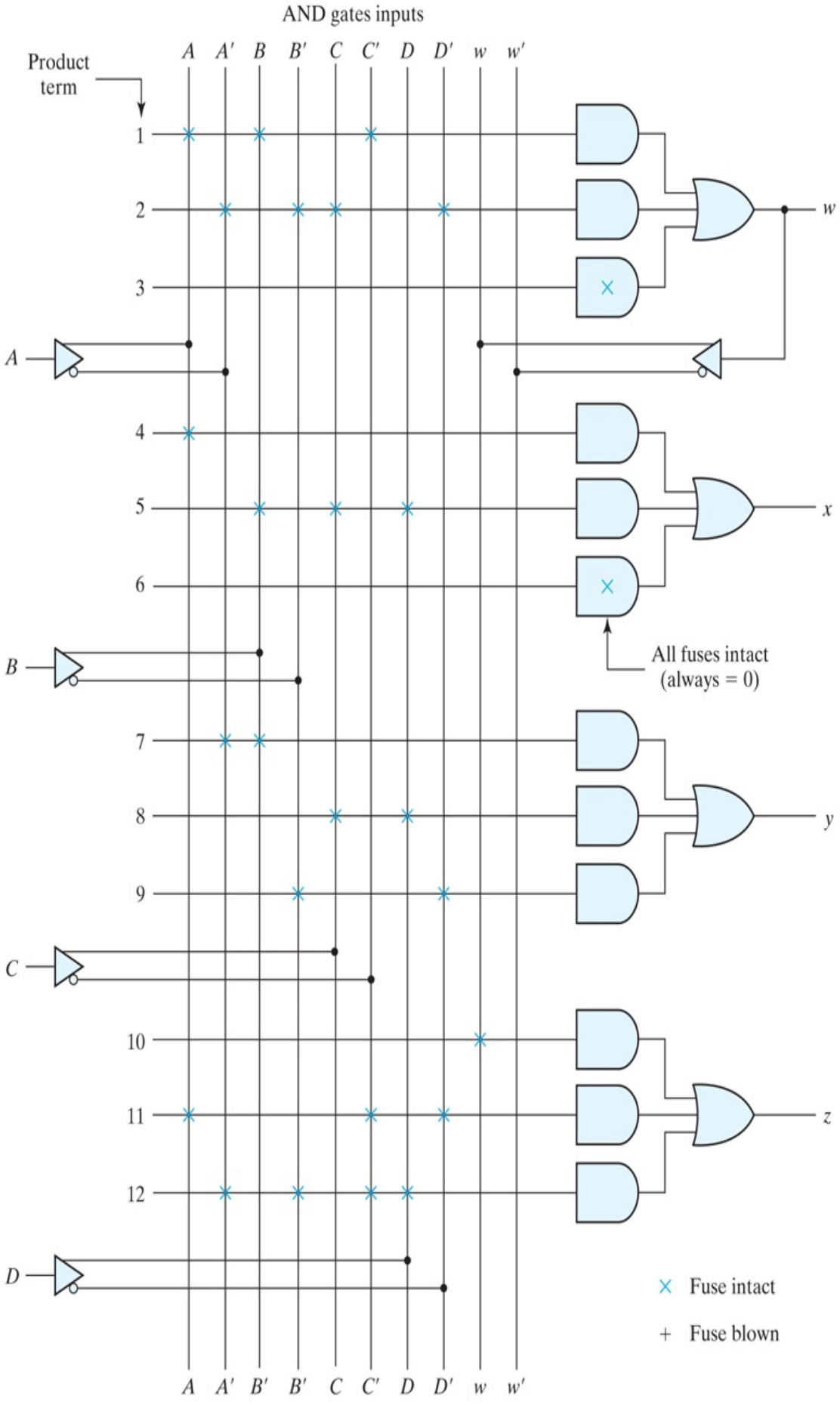

AND gates inputs
A A′ B B′ C C′ D D′ w w′
Product term
1
2
3
4
5
6
7
8
9
10
11
12
w
x
y
z
A
B
C
D
All fuses intact (always = 0)
× Fuse intact
+ Fuse blown

FIGURE 7.17

Fuse map for PAL as specified in Table 7.6

Description

As with all PLDs, the design with PALs is facilitated by using CAD techniques. The blowing of internal fuses is a hardware procedure done with the help of special electronic instruments.

7.8 SEQUENTIAL PROGRAMMABLE DEVICES

Digital systems are designed with flip-flops and gates. Since the combinational PLD consists of only gates, it is necessary to include external flip-flops when they are used in the design. Sequential programmable devices include both gates and flip-flops. In this way, the device can be programmed to perform a variety of sequential-circuit functions. There are several types of sequential programmable devices available commercially, and each device has vendor-specific variants within each type. The internal logic of these devices is too complex to be shown here. Therefore, we will describe three major types without going into their detailed construction:

1. Sequential (or simple) programmable logic device (SPLD).
2. Complex programmable logic device (CPLD).
3. Field-programmable gate array (FPGA).

The sequential PLD is sometimes referred to as a simple PLD to differentiate it from the complex PLD. The SPLD includes flip-flops, in addition to the AND–OR array, within the integrated circuit chip. The result is a sequential circuit as shown in Fig. 7.18. A PAL or PLA is modified by including a number of flip-flops connected to form a register. The circuit outputs can be taken from the OR gates or from the outputs of the flip-flops. Additional programmable connections are available to include the flip-flop outputs in the product terms formed with the AND array. The flip-flops may be of the *D* or the *JK* type.

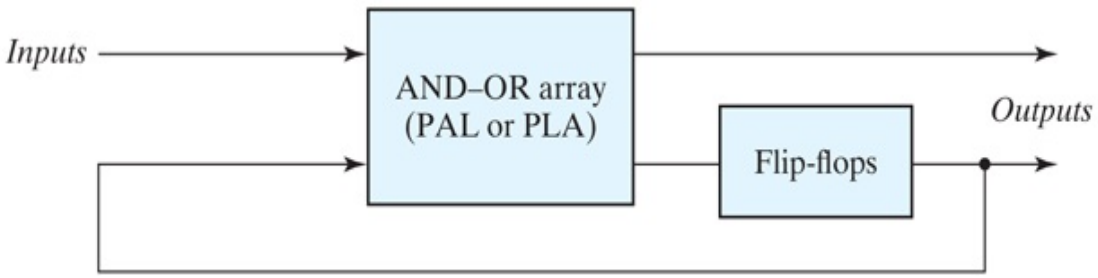


FIGURE 7.18

Sequential programmable logic device

The first programmable device developed to support sequential circuit implementation is the field-programmable logic sequencer (FPLS). A typical FPLS is organized around a PLA with several outputs driving flip-flops. The flip-flops are flexible in that they can be programmed to operate as either the *JK* or the *D* type. The FPLS did not succeed commercially, because it has too many programmable connections. The configuration mostly used in an SPLD is the combinational PAL together with *D* flip-flops. A PAL that includes flip-flops is referred to as a *registered* PAL, to signify that the device contains flip-flops in addition to the AND–OR array. Each section of an SPLD is called a *macrocell*, which is a circuit that contains a sum-of-products combinational logic function and an optional flip-flop. We will assume an AND–OR sum-of-products function, but in practice, it can be any one of the two-level implementations presented in Section 3.7.

Figure 7.19 shows the logic of a basic macrocell. The AND–OR array is the same as in the combinational PAL shown in Fig. 7.16. The output is driven by an edge-triggered *D* flip-flop connected to a common clock input and changes state on a clock edge. The output of the flip-flop is connected to a three-state buffer (or inverter) controlled by an output-enable signal marked in the diagram as *OE*. The output of the flip-flop is fed back into one of the inputs of the programmable AND gates to provide the present-state condition for the sequential circuit. A typical SPLD has from 8 to 10 macrocells within one IC package. All the flip-flops are connected to the common *CLK* input, and all three-state buffers are controlled by the *OE* input.

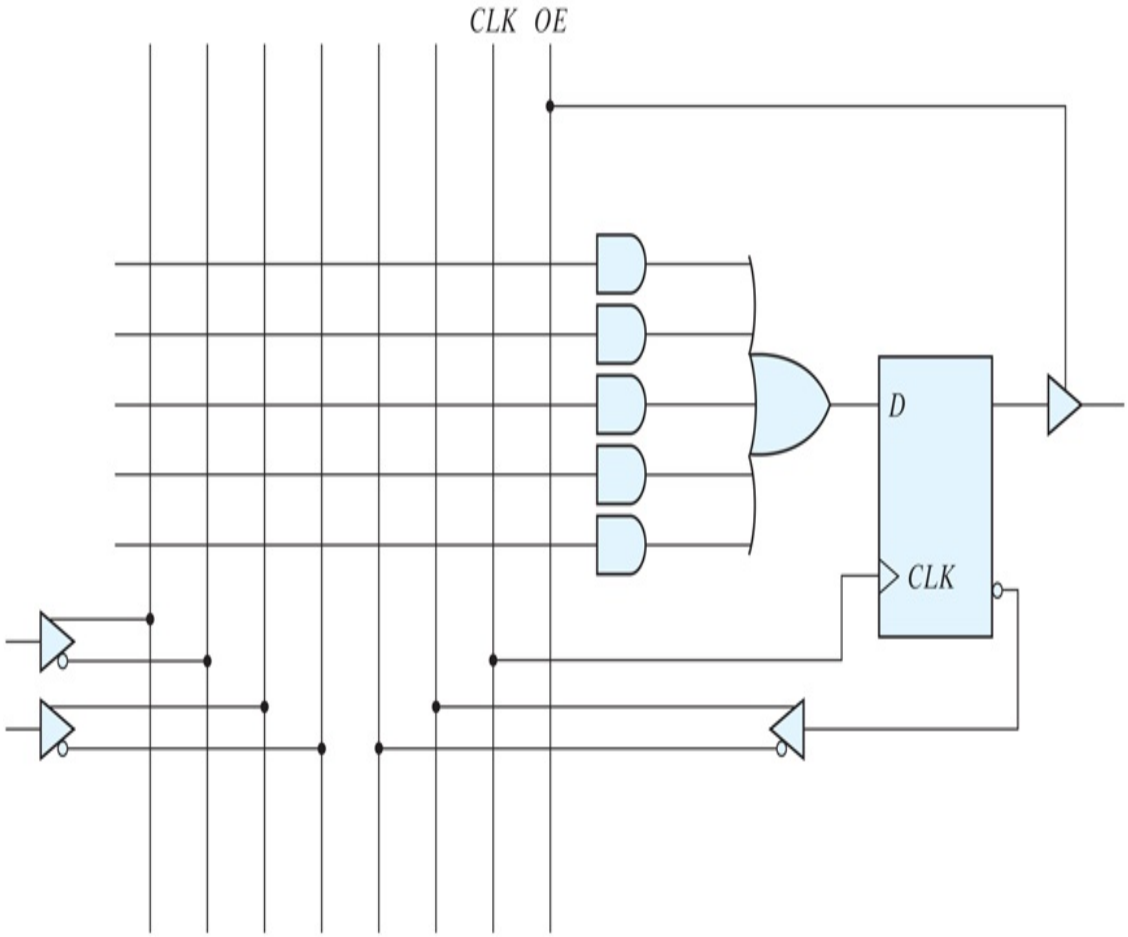


FIGURE 7.19

Basic macrocell logic

Description

In addition to programming the AND array, a macrocell may have other programming features. Typical programming options include the ability to either use or bypass the flip-flop, the selection of clock edge polarity, the selection of preset and clear for the register, and the selection of the true value or complement of an output. An XOR gate is used to program a true/complement condition. Multiplexers select between two or four distinct paths by programming the selection inputs.

The design of a digital system using PLDs often requires the connection of several devices to produce the complete specification. For this type of application, it is more economical to use a complex programmable logic

device (CPLD), which is a collection of individual PLDs on a single integrated circuit. A programmable interconnection structure allows the PLDs to be connected to each other in the same way that can be done with individual PLDs.

Figure 7.20 shows the general configuration of a CPLD. The device consists of multiple PLDs interconnected through a programmable switch matrix. The input–output (I/O) blocks provide the connections to the IC pins. Each I/O pin is driven by a three-state buffer and can be programmed to act as input or output. The switch matrix receives inputs from the I/O block and directs them to the individual macrocells. Similarly, selected outputs from macrocells are sent to the outputs as needed. Each PLD typically contains from 8 to 16 macrocells, usually fully connected. If a macrocell has unused product terms, they can be used by other nearby macrocells. In some cases the macrocell flip-flop is programmed to act as a *D*, *JK*, or *T* flip-flop.

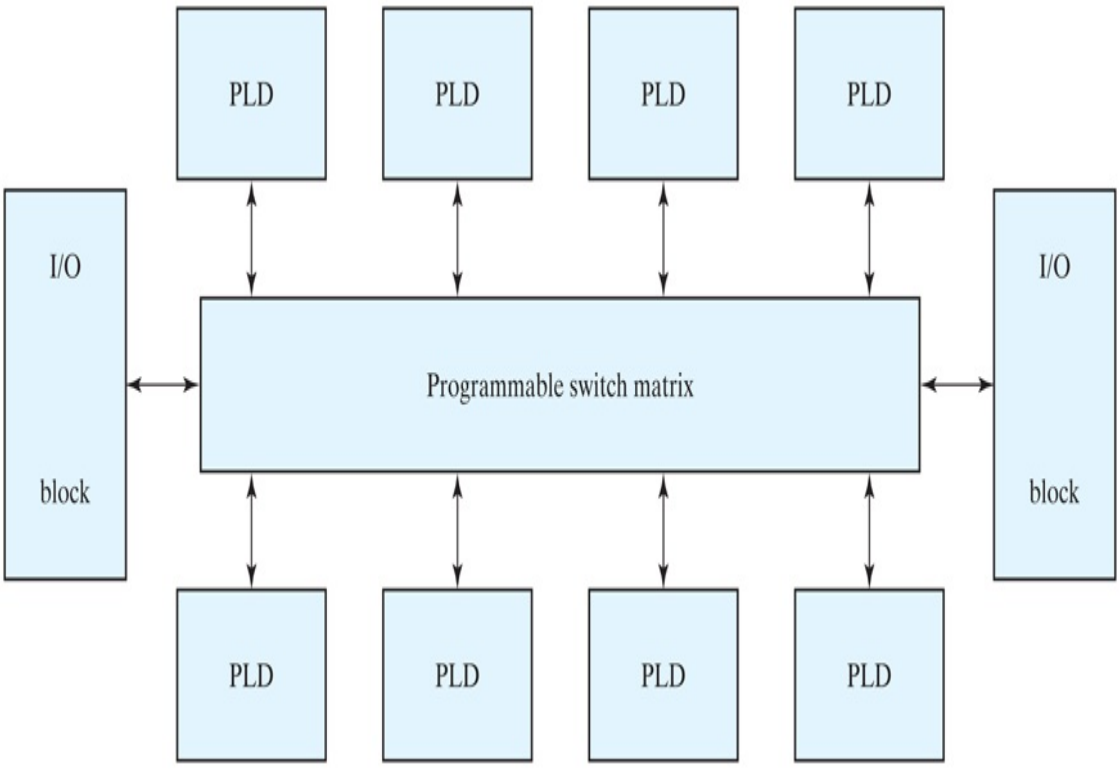


FIGURE 7.20

General CPLD configuration

Different manufacturers have taken different approaches to the general architecture of CPLDs. Areas in which they differ include the individual PLDs (sometimes called *function blocks*), the type of macrocells, the I/O blocks, and the programmable interconnection structure. The best way to investigate a vendor-specific device is to look at the manufacturer's literature.

The basic component used in VLSI design is the *gate array*, which consists of a pattern of gates, fabricated in an area of silicon, that is repeated thousands of times until the entire chip is covered with gates. Arrays of one thousand to several hundred thousand gates are fabricated within a single IC chip, depending on the technology used. The design with gate arrays requires that the customer provide the manufacturer the desired interconnection pattern. The first few levels of the fabrication process are common and independent of the final logic function. Additional fabrication steps are required to interconnect the gates according to the specifications given by the designer.

A field-programmable gate array (FPGA) is a VLSI circuit that can be programmed at the user's location. A typical FPGA consists of an array of millions of logic blocks, surrounded by programmable input and output blocks and connected together via programmable interconnections. There is a wide variety of internal configurations within this group of devices. The performance of each type of device depends on the circuit contained in its logic blocks and the efficiency of its programmed interconnections.

A typical FPGA logic block consists of lookup tables, multiplexers, gates, and flip-flops. A lookup table is a truth table stored in an SRAM and provides the combinational circuit functions for the logic block. These functions are realized from the lookup table, in the same way that combinational circuit functions are implemented with ROM, as described in Section 7.5. For example, a 16×2 SRAM can store the truth table of a combinational circuit that has four inputs and two outputs. The combinational logic section, along with a number of programmable multiplexers, is used to configure the input equations for the flip-flop and the output of the logic block.

The advantage of using RAM instead of ROM to store the truth table is that the table can be programmed by writing into memory. The disadvantage is that the memory is volatile and presents the need for the lookup table's content to be reloaded in the event that power is disrupted.

The program can be downloaded either from a host computer or from an onboard PROM. The program remains in SRAM until the FPGA is reprogrammed or the power is turned off. The device must be reprogrammed every time power is turned on. The ability to reprogram the FPGA can serve a variety of applications by using different logic implementations in the program.

The design with PLD, CPLD, or FPGA requires extensive computer-aided design (CAD) tools to facilitate the synthesis procedure. Among the tools that are available are schematic entry packages and hardware description languages (HDLs), such as ABEL, Verilog, VHDL, and SystemVerilog. Synthesis tools are available that allocate, configure, and connect logic blocks to match a high-level design description written in HDL. As an example of CMOS FPGA technology, we will discuss the evolution of Xilinx FPGAs.[2]

[2] See www.Altera.com for an alternative CMOS FPGA architecture.

Xilinx FPGAs

Xilinx launched the world's first commercial FPGA in 1985, with the vintage XC2000 device family.[3] The XC3000 and XC4000 families soon followed, setting the stage for today's Spartan™, Artix™, Kintex™, and Virtex™ device families. Each evolution of devices has brought improvements in density, performance, power consumption, voltage levels, pin counts, I/O support, and functionality. For example, the inaugural Spartan family of devices initially offered a maximum of 40K system gates, but today's Spartan-6 family now offers 150,000 logic cells plus up to 4.8 Mb block RAM. Higher densities are available in the Artix™, Kintex™, and Virtex™ device families.

[3] See www.Xilinx.com for detailed, up-to-date information about Xilinx products.

The remainder of this chapter will provide an introduction to the architecture of Xilinx devices. Its objective is to create awareness of important characteristics of FPGAs, as illustrated by the evolution of Xilinx devices, and is not intended to be comprehensive. It presumes some knowledge of CMOS transmission gates, which may not be covered until

later in a curriculum.

Basic Xilinx Architecture

The basic architecture of Spartan and earlier device families consists of an array of configurable logic blocks (CLBs), a variety of local and global routing resources, and input–output (I/O) blocks (IOBs), programmable I/O buffers, and an SRAM-based configuration memory, as shown in Fig. 7.21.

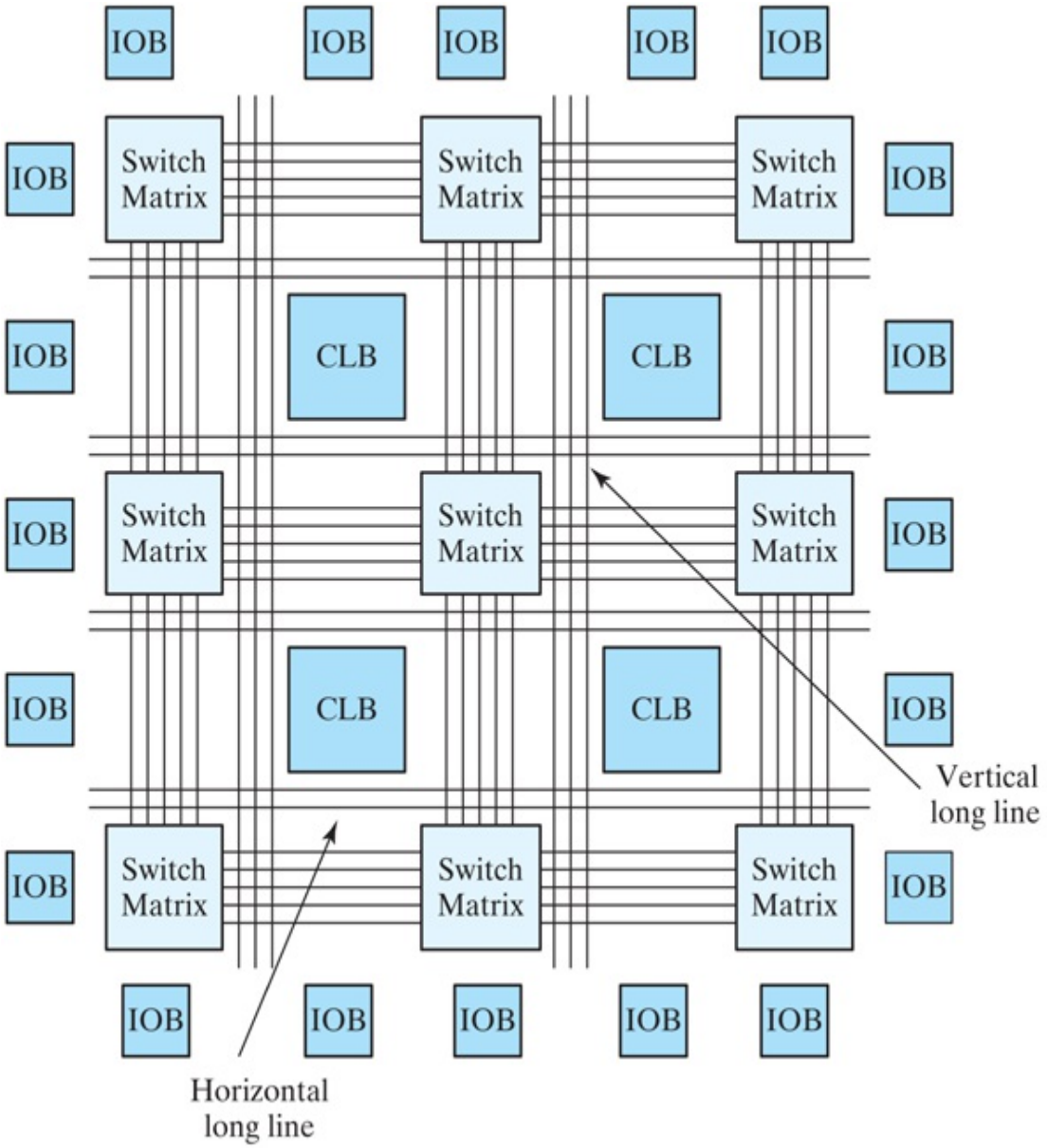


FIGURE 7.21

Basic architecture of Xilinx Spartan and predecessor devices

Description

Configurable Logic Block (CLB)

Each CLB consists of a programmable lookup table, multiplexers, registers, and paths for control signals, as shown in Fig. 7.22. Two of the function generators (F and G) of the lookup table 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 lookup tables or from external inputs. The three function generators can be programmed to generate (1) three different functions of three independent sets of variables (two 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.

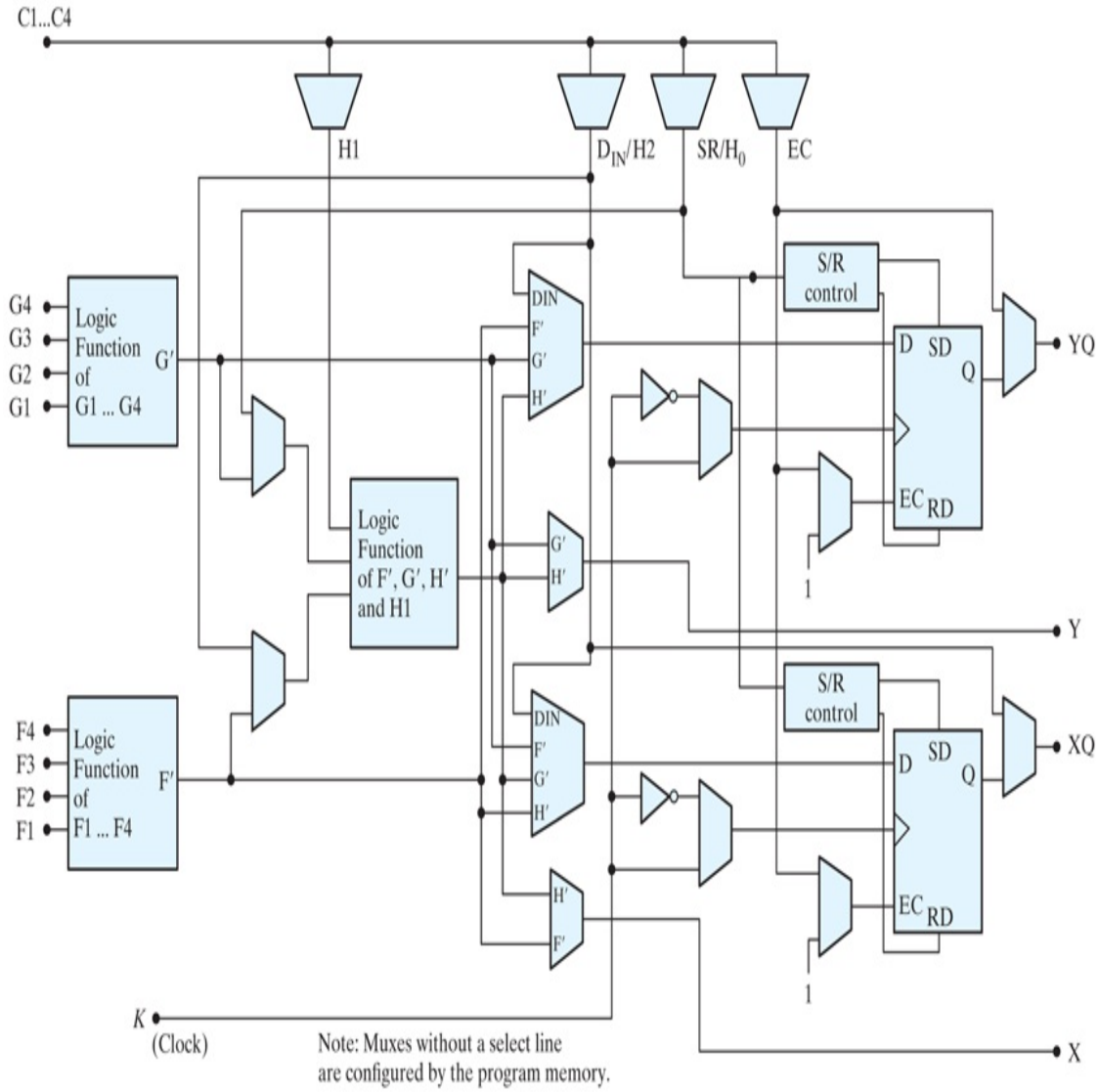


FIGURE 7.22

CLB architecture

Description

Each CLB 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 an enable. The storage elements can get their inputs from the function generators or from the Din input. The function generators can also drive two outputs (*X* and *Y*) directly 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 set/reset during power-up; the global set/reset is programmed to match the programming of the local S/R control for a given storage element.

Distributed RAM

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

Interconnect Resources

A grid of switch matrices overlays the architecture of CLBs to provide general-purpose interconnect for branching and routing signals throughout the device. The interconnect has three types of general-purpose interconnects: single-length lines, double-length lines, and long lines. A grid of horizontal and vertical single-length lines connects an array of switch boxes that provide a reduced number of connections between signal paths within each box, not a full crossbar switch. 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 hardwired 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 matrices, thus eliminating the delay incurred on paths going through a switch matrix.[4]

[4] See Xilinx documentation for the pin-out conventions to establish local interconnects between CLBs.

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-fan-out control signals. Long vertical lines have a programmable splitter that segments the lines and allows two independent routing channels spanning one-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.

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 buses within the architecture and implement wired-AND logic.[5] 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.

[5] A wired-AND net is pulled to 0 if any of its drivers is 0.

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 (PIPs) 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 Fig. 7.23. The connection is established when the transmission gate is on (i.e., when 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). Thus, the device can be reprogrammed simply by changing the contents of the controlling memory cell.

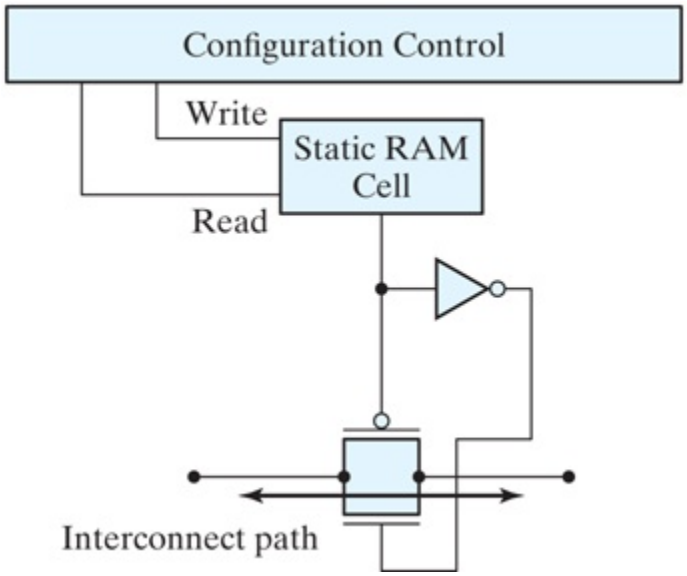


FIGURE 7.23

RAM cell controlling a PIP transmission gate

The architecture of a PIP-based interconnection in a switch box is shown in Fig. 7.24, which shows possible signal paths through a PIP. The configuration of CMOS transmission gates determines the connection between a horizontal line and the opposite horizontal line and between the vertical lines at the connection. Each switch matrix PIP requires six pass transistors to support full horizontal and vertical connectivity.

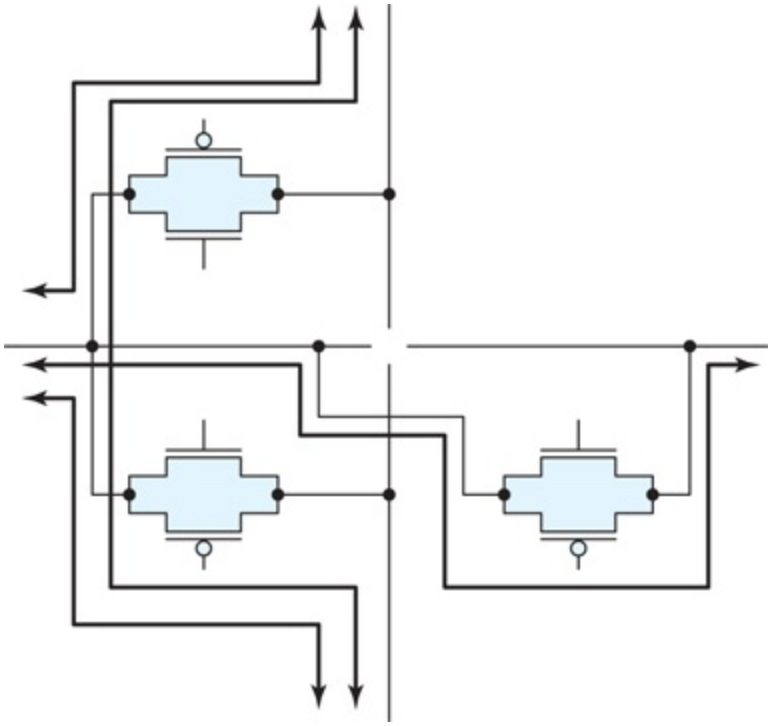


FIGURE 7.24

Circuit for a programmable PIP

Description

I/O Block (IOB)

Each programmable I/O pin has a programmable IOB having buffers for compatibility with TTL and CMOS signal levels. Figure 7.25 shows a simplified schematic for a programmable IOB. It can be used as an input, an output, or a 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/reset.

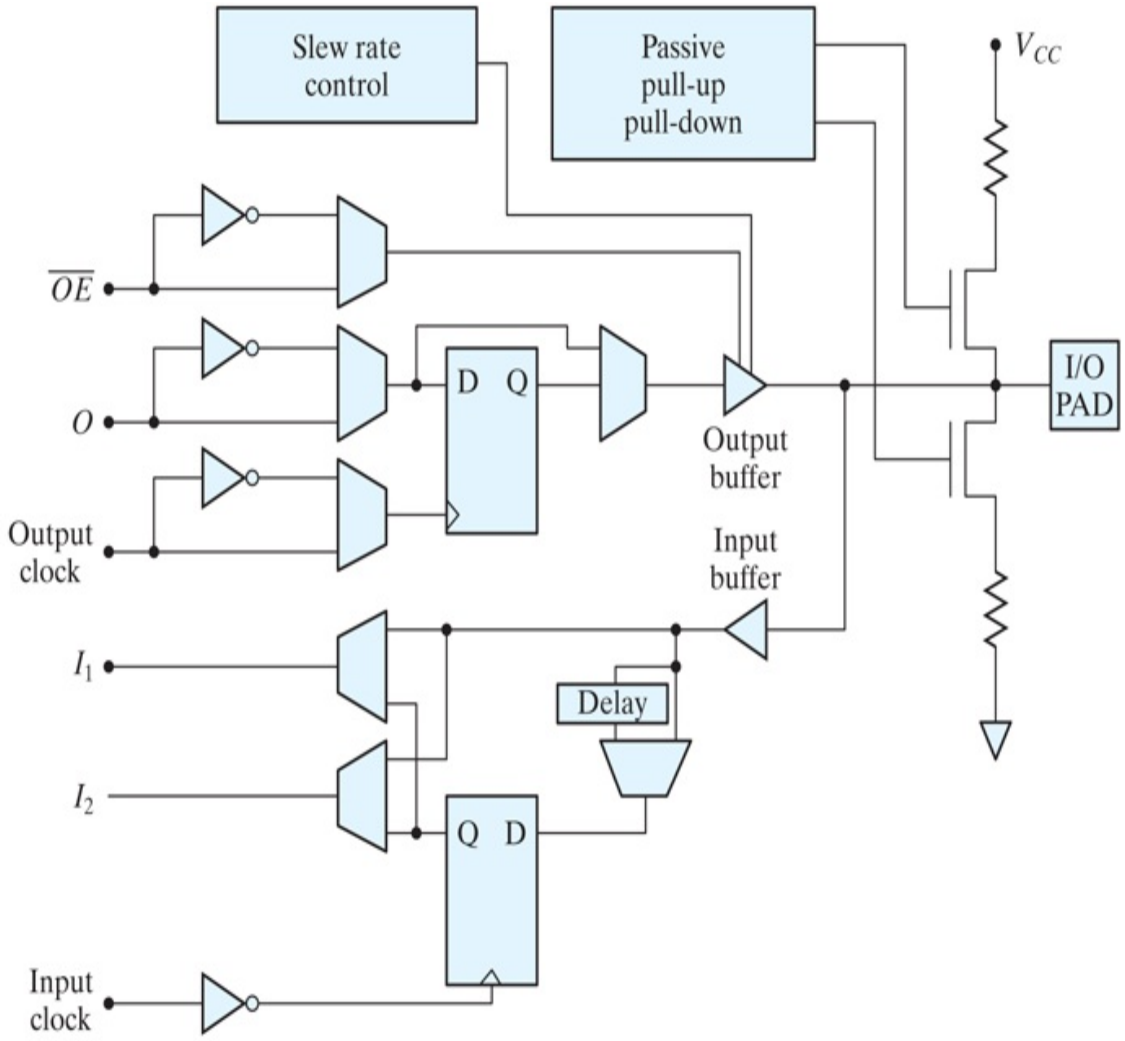


FIGURE 7.25

XC4000 series IOB

Description

Internal delay elements compensate for the delay induced when a clock signal passes through a global buffer before reaching an IOB. This strategy 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 devices have embedded 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 testing, the device can be checked to verify that all the pins on a PC board are connected and operate properly 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.

Enhancements

Spartan chips can accommodate embedded soft cores, and their on-chip distributed, dual-port, synchronous RAM (SelectRAM™) can be used to implement first-in, first-out register files (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 7.26 displays the structure of the on-chip RAM that is formed by programming a lookup table to implement a single-port RAM with synchronous write and asynchronous read. Each CLB can be programmed as a 16×2 or 32×1 memory.

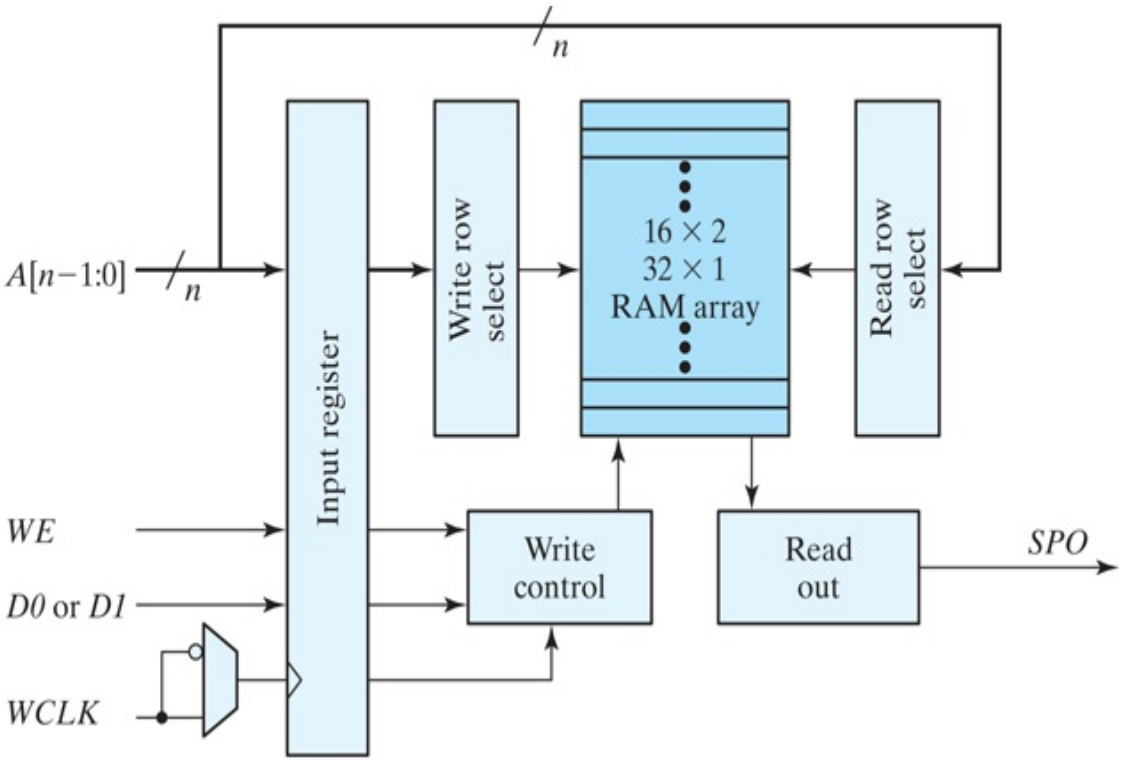


FIGURE 7.26

Distributed RAM cell formed from a lookup table

Description

Dual-port RAMs are emulated in a Spartan device by the structure shown in Fig. 7.27, which has a single (common) write port and two asynchronous read ports. A CLB can form a memory having a maximum size of 16×1.

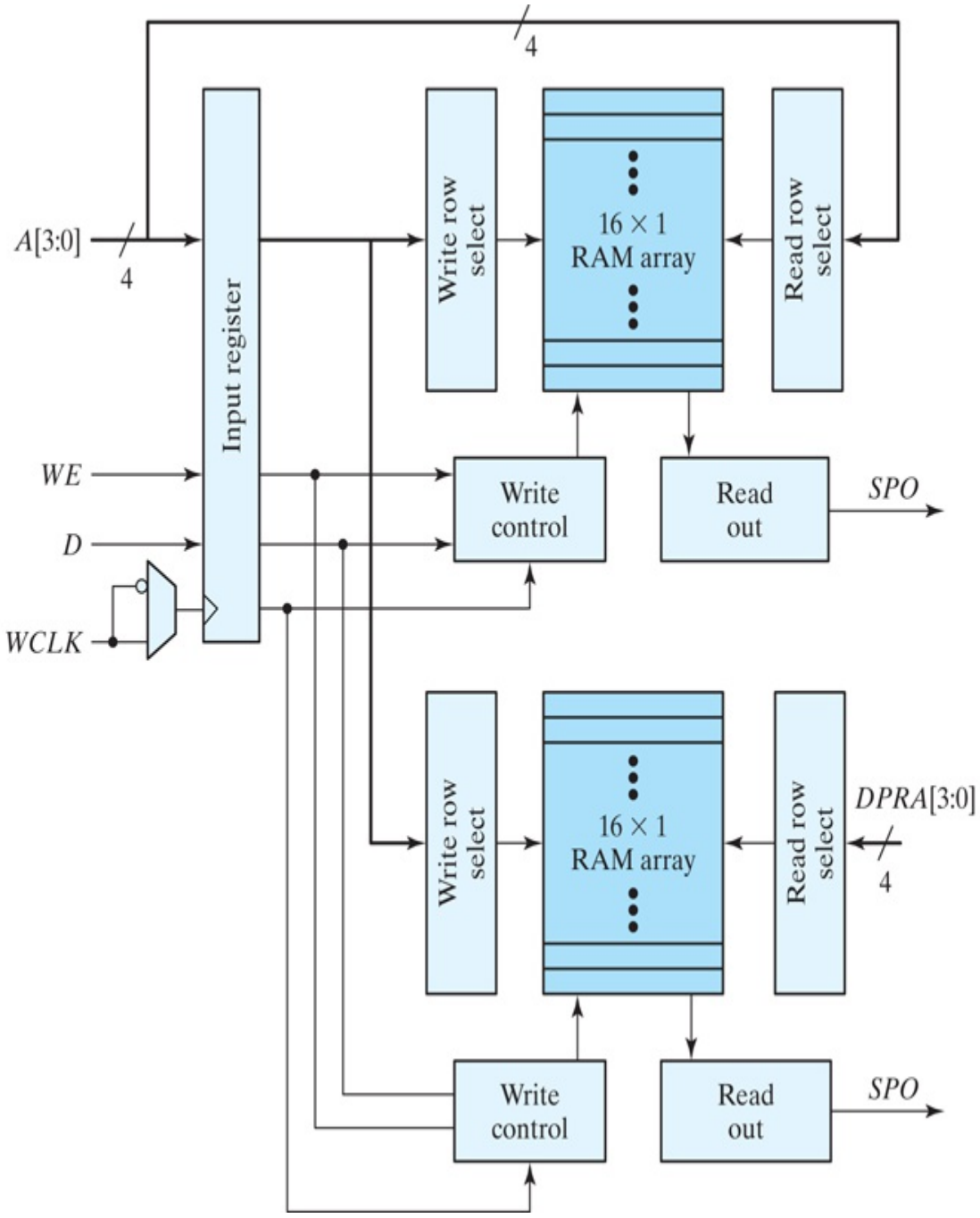


FIGURE 7.27

Spartan dual-port RAM

Description

The architecture of the Spartan and earlier devices consists of an array of CLB tiles mingled within an array of switch matrices, surrounded by a

perimeter of IOBs. Those devices supported only distributed memory, whose use reduces the number of CLBs that could be used for logic. Their relatively small amount of on-chip memory limited the devices to applications in which operations with off-chip memory devices do not compromise performance objectives. The Spartan family evolved to support configurable embedded block memory, as well as distributed memory in a new architecture.

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 predecessor Spartan devices: (1) on-chip block memory, (2) novel architecture, (3) support for multiple I/O standards, and (4) delay locked loops (DLLs).[6] With six layers of metal for interconnect, devices in this family incorporate 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 or distributed memory that is available for an application. A large on-chip memory can improve system performance by eliminating or reducing the need to access off-chip storage.

[6] Spartan II devices do not support low-voltage differential signaling (LVDS) or low-voltage positive emitter-coupled logic (LVPECL) I/O standards.

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 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 (shorter clock periods), 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 or management in high-speed circuits. DLLs eliminate the clock distribution delay and provide frequency multipliers, frequency dividers, and clock mirrors.The top-level tiled architecture introduced by

the Spartan II device, shown in Fig. 7.28, marked a new organization structure of the Xilinx parts. Each of four quadrants of CLBs is supported by a DLL and is flanked by a 4,096-bit block[7] of RAM, and the periphery of the chip is lined with IOBs.

[7] Parts are available with up to 14 blocks (56 K bits).

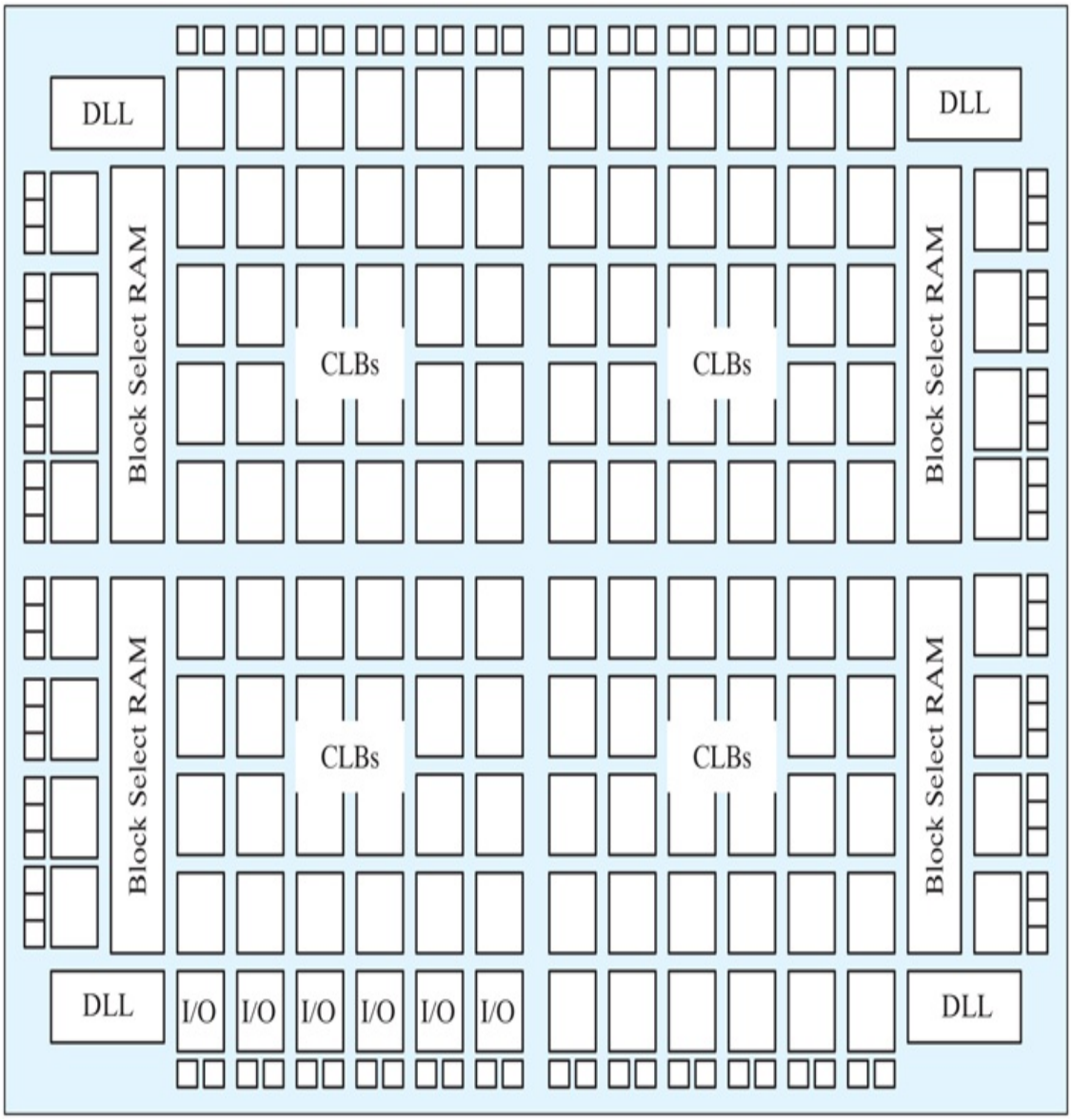


FIGURE 7.28

Spartan II architecture

Each CLB contains four logic cells, organized as a pair of slices. Each

logic cell, shown in Fig. 7.29, has a four-input lookup table, logic for carry and control, and a *D*-type flip-flop. The CLB contains additional logic for configuring functions of five or six inputs.

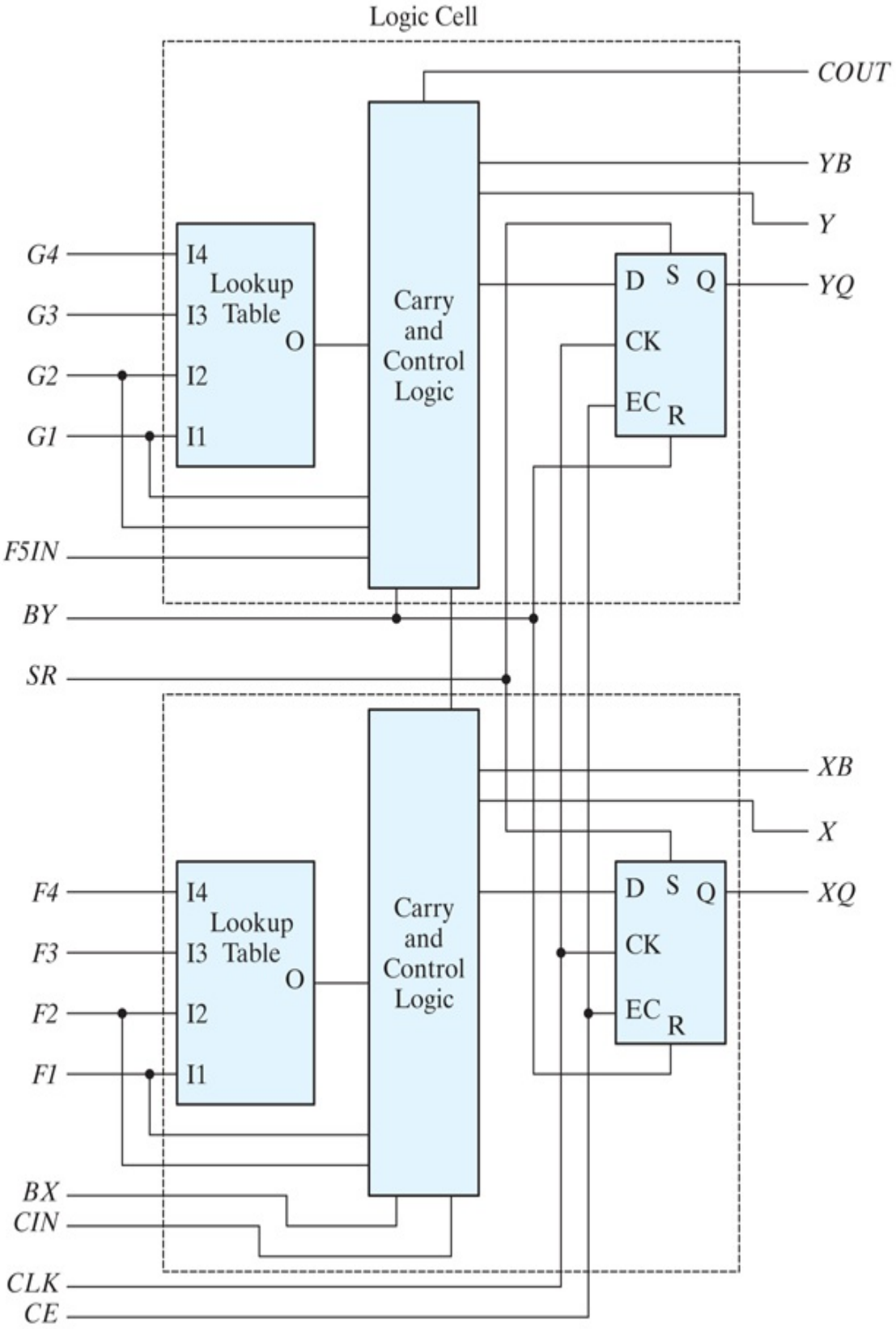


FIGURE 7.29

Spartan II CLB slice

Description

The Spartan II part family provided the flexibility and capacity of an on-chip block RAM; in addition, each lookup table could be configured as a 16×1 RAM (distributed), and the pair of lookup tables in a logic cell could be configured as a 16×2 bit RAM or a 32×1 bit RAM.

The IOBs of the Spartan II family were individually programmable to support the reference, output voltage, and termination voltages of a variety of high-speed memory and bus standards. (See Fig. 7.30.) Each IOB had three registers that could function as *D*-type flip-flops or as level-sensitive latches. One register (*TFF*) could be used to register the signal that (synchronously) controls the programmable output buffer. A second register (*OFF*) could be programmed to register a signal from the internal logic. (Alternatively, a signal from the internal logic could pass directly to the output buffer.) The third device could register the signal coming from the I/O pad. (Alternatively, this signal could 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 could be used to eliminate the pad-to-pad hold time.

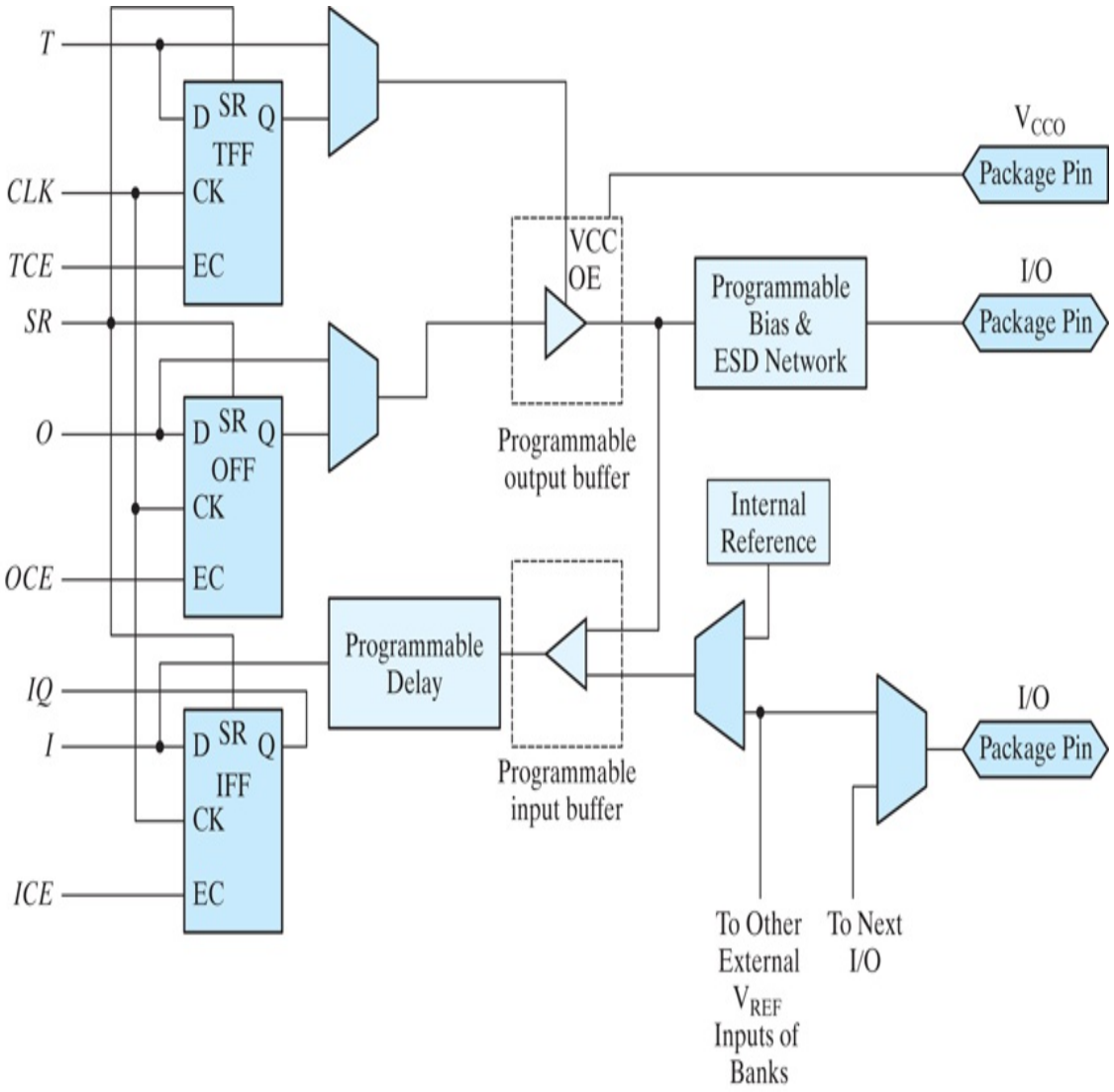


FIGURE 7.30

Spartan II IOB

Description

SPARTAN-6 FPGA Family

A recent addition to the Spartan line of devices is the Spartan-6 FPGA. According to Xilinx product specifications, the Spartan-6 device family is targeted at low-cost, high-volume applications. Its power consumption is half that of previous Spartan families. A suite of thirteen members spans

the family, providing from 3,840 to 147,443 logic cells. Table 7.7 presents significant attributes of devices in the Spartan-6 family. The look-up table (LUT) in this family has six inputs (supporting more complex logic), compared to four in previous generations of Xilinx parts. Each slice contains four LUTs and eight flip-flops. The clock management tile (CMT) contains two digital clock managers (DCMs) and one phase-locked loop (PLL).[8] They have application in clock synchronization, demodulation, and frequency synthesis. Other attributes of the Spartan-6 family are available in the manufacturer's literature.

[8] Phase-lock loops reduce clock jitter to increase clock stability.

Table 7.7 *Features of the Spartan-6 device family*

	Configurable Logic Blocks (CLBs)					Block RAM Blocks	
Device	**Logic Cells**	**Slices**	**Flip-Flops**	**Max Distributed RAM(Kb)**	**DSP1 Slices**	**18 Kb**	**Max (Kb)**
XC6SLX4	3,840	600	4,800	75	8	12	21(
XC6SLX9	9,152	1,430	11,440	90	16	32	57(
XC6SLX16	14,579	2,278	18,224	136	32	32	57(
XC6SLX25	24,051	3,758	30,064	229	38	52	93(

XC6SLX45	43,661	6,822	54,576	401	58	116	2,08
XC6SLX75	74,637	11,662	93,296	692	132	172	3,09
XC6SLX100	101,261	15,822	126,576	976	180	268	4,82
XC6SLX150	147,443	23,038	184,304	1,355	180	268	4,82
XC6SLX25T	24,051	3,758	30,064	229	38	52	93
XC6SLX45T	43,661	6,822	54,576	401	58	116	2,08
XC6SLX75T	74,637	11,662	93,296	692	132	172	3,09
XC6SLX100T	101,261	15,822	126,576	976	180	268	4,82
XC6SLX150T	147,443	23,038	184,304	1,355	180	268	4,82

The evolution of devices by Xilinx and other FPGA manufacturers has been driven by the progress in integrated circuit fabrication technology, which has dramatically increased the density of devices. Xilinx now offers device families fabricated in 45 nm technology (Spartan-6, Artix), to those in 16 nm technology (Kintex, Virtex).

Xilinx Virtex FPGAs

The Virtex family is the flagship of Xilinx offerings. Its Virtex Ultrascale™ devices have an architecture with up to 4.4 M logic cells fabricated in a 20 nm CMOS process. The family is said to address four key factors that influence the solution to complex system-level and

system-on-chip designs: (1) the level of integration, (2) the amount of embedded memory, (3) performance (timing), and (4) subsystem interfaces. The family targets applications requiring a balance of high-performance logic, serial connectivity, signal processing, and embedded processing (e.g., wireless communications). Process rules for leading-edge Virtex parts stand at 20 nm, with a 1 V operating voltage. The rules allow up to 330,000 logic cells and over 200,000 internal flip-flops with clock enable, together with over 10 Mb of block RAM, and 550 MHz clock technology packed into a single die.

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 requiring multiple clock domains and high-frequency I/O. The Virtex architecture is shown in Fig. 7.31, and its IOB is shown in Fig. 7.32. Table 7.8 presents some key features of the device family.

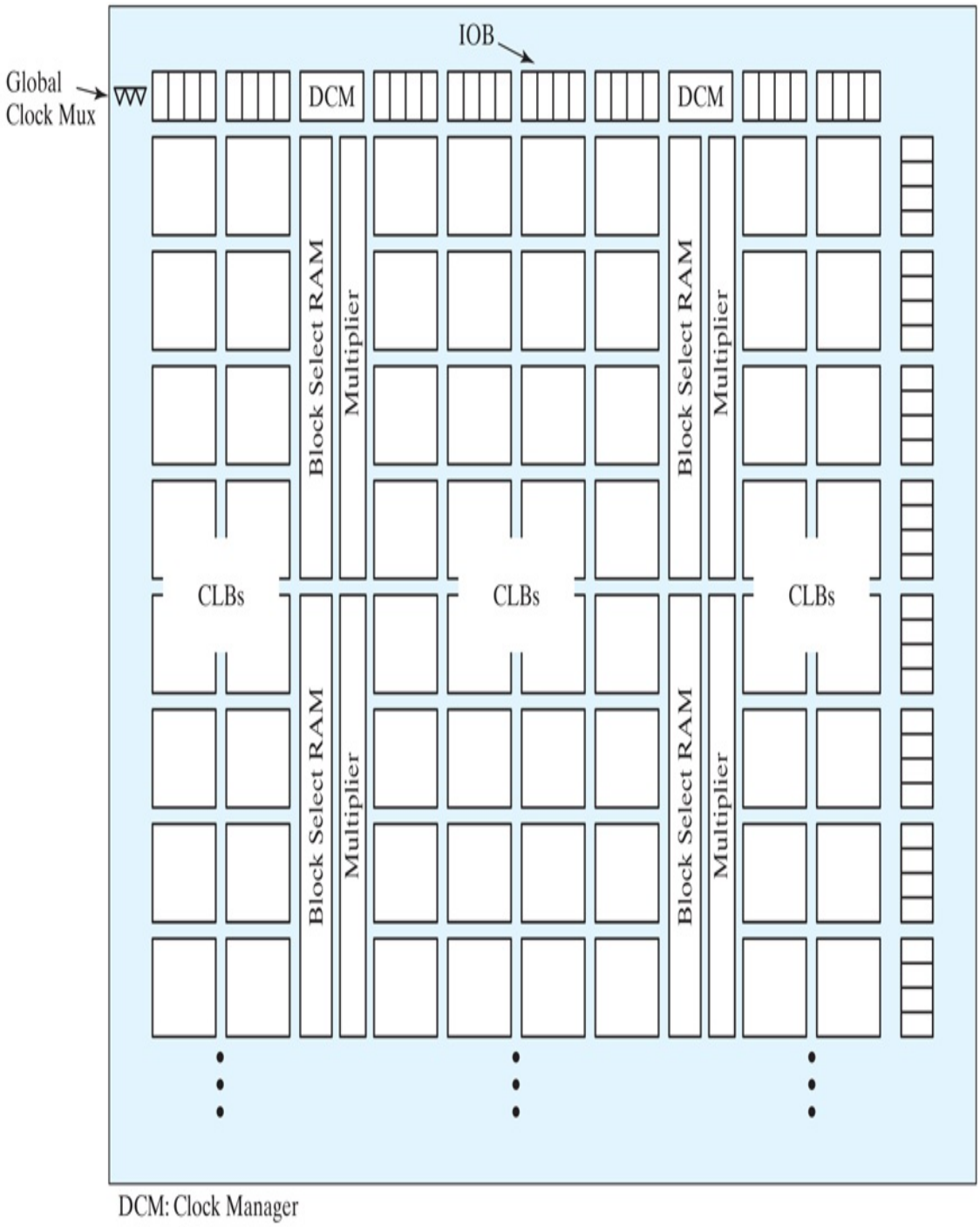


FIGURE 7.31

Virtex overall architecture

Description

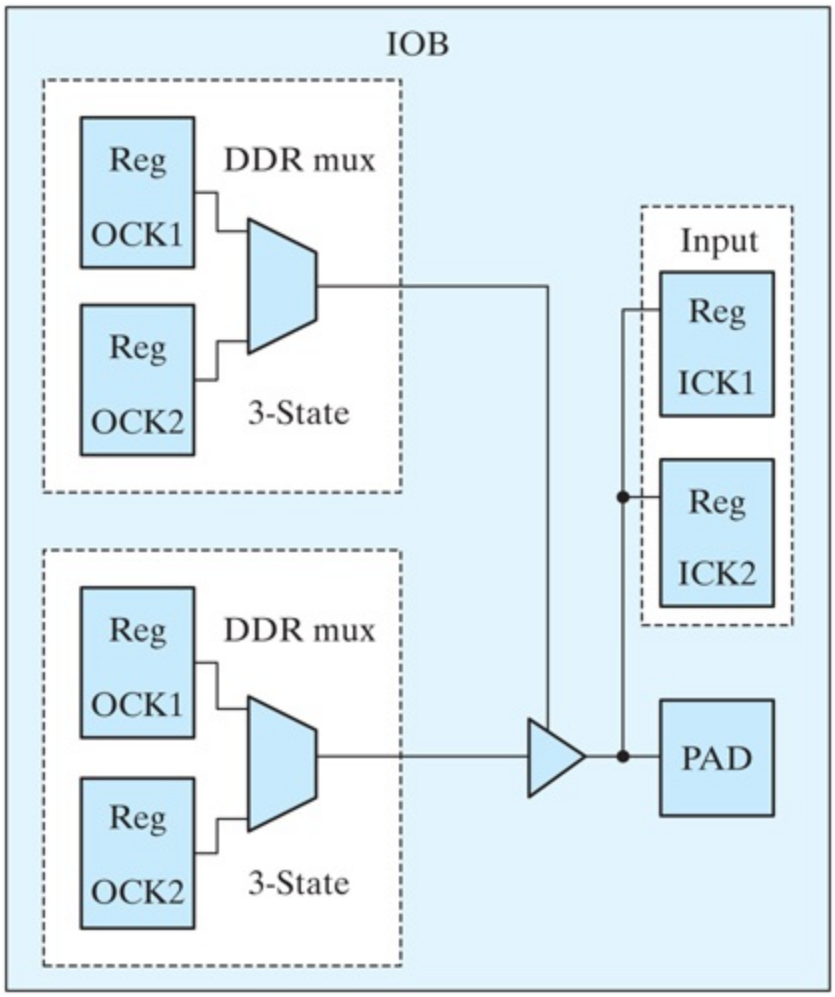


FIGURE 7.32

Virtex IOB block

Description

Table 7.8 *Features of the Virtex Ultrascale device family*

Device	System Logic Cells (K)	CLB Flip-Flops (K)	CLB LUTs (K)	Max Distributed RAM (Mb)	Total Block RAM (Mb)	Clock Mgmt Tiles (CMTs)	DSP Slices

VU3P	862	788	394	12.0	25.3	10	2,280
VU5P	1,314	1,201	601	18.3	36.0	20	3,474
VU7P	1,724	1,576	788	24.1	50.6	20	4,560
VU9P	2,586	2,364	1,182	36.1	75.9	30	6,840
VU11P	2,835	2,592	1,296	36.2	70.9	12	2,088
VU13P	3,780	3,456	1,728	48.3	94.5	16	12,288

PROBLEMS

(Answers to problems marked with * appear at the end of the book.)

1. 7.1 The memory units that follow are specified by the number of words times the number of bits per word. How many address lines and input–output data lines are needed in each case?

    1. 8 K×32
    2. 2 G×8
    3. 16 M×32
    4. 256 K×64

2. 7.2 * Give the number of bytes stored in the memories listed in Problem 7.1.

3. 7.3* Word number 565 in the memory shown in Fig. 7.3 contains the binary equivalent of 1,210. List the 10-bit address and the 16-bit memory content of the word.

4. 7.4 Show the memory cycle timing waveforms (see Fig. 7.4) for the write and read operations. Assume a CPU clock of 2000 MHz and a memory cycle time of 25 ns.

5. 7.5 Write a HDL testbench for the ROM described in Example 7.1. The test program stores the binary equivalent of 710 in address 5 and the binary equivalent of 510 in address 7. Then the two addresses are read to verify their stored contents.

6. 7.6 Enclose the 4×4 RAM of Fig. 7.6 in a block diagram showing all inputs and outputs. Assuming three-state outputs, construct an 8×8 memory using four 4×4 RAM units.

7. 7.7* A 16 K×4 memory uses coincident decoding by splitting the internal decoder into *X*-selection and *Y*-selection.

1. What is the size of each decoder, and how many AND gates are required for decoding the address?

2. Determine the *X* and *Y* selection lines that are enabled when the input address is the binary equivalent of 6,000.

8. 7.8* (a) How many 32 K×8 RAM chips are needed to provide a memory capacity of 256 K bytes?

    1. How many lines of the address must be used to access 256 K bytes? How many of these lines are connected to the address inputs of all chips?

    2. How many lines must be decoded for the chip select inputs? Specify the size of the decoder.

9. 7.9 A DRAM chip uses two-dimensional address multiplexing. It has 13 column address pins, with the row address having one bit more than the column address. What is the capacity of the memory?

10. 7.10* Given the 8-bit data word 01011011, generate the 13-bit composite word for the Hamming code that corrects single errors and detects double errors.

11. 7.11* Obtain the 15-bit Hamming code word for the 11-bit data word 11001001010.

12. 7.12* A 12-bit Hamming code word containing 8 bits of data and 4 parity bits is read from memory. What was the original 8-bit data word that was written into memory if the 12-bit word read out is as follows:

    1. 000011101010

    2. 101110000110

    3. 101111110100

13. 7.13* How many parity check bits must be included with the data word to achieve single-error correction and double-error detection when the data word contains

1. 16 bits.

2. 32 bits.

3. 48 bits.

14. 7.14 It is necessary to formulate the Hamming code for four data bits, D3, D5, D6, and D7, together with three parity bits, P1, P2, and P4.

    1. * Evaluate the 7-bit composite code word for the data word 0010.

    2. Evaluate three check bits, C4, C2, and C1, assuming no error.

    3. Assume an error in bit D5 during writing into memory. Show how the error in the bit is detected and corrected.

    4. Add parity bit P8 to include double-error detection in the code. Assume that errors occurred in bits P2 and D5. Show how the double error is detected.

15. 7.15 Using 64×8 ROM chips with an enable input, construct a 512×8 ROM with eight chips and a decoder.

16. 7.16* A ROM chip of 4,096×8 bits has two chip select inputs and operates from a 5-V power supply. How many pins are needed for the integrated circuit package? Draw a block diagram, and label all input and output terminals in the ROM.

17. 7.17 The 32×6 ROM, together with the 20 line, as shown in Fig. P7.17, converts a six-bit binary number to its corresponding two-digit BCD number. For example, binary 100001 converts to BCD 011 0011 (decimal 33). Specify the truth table for the ROM.

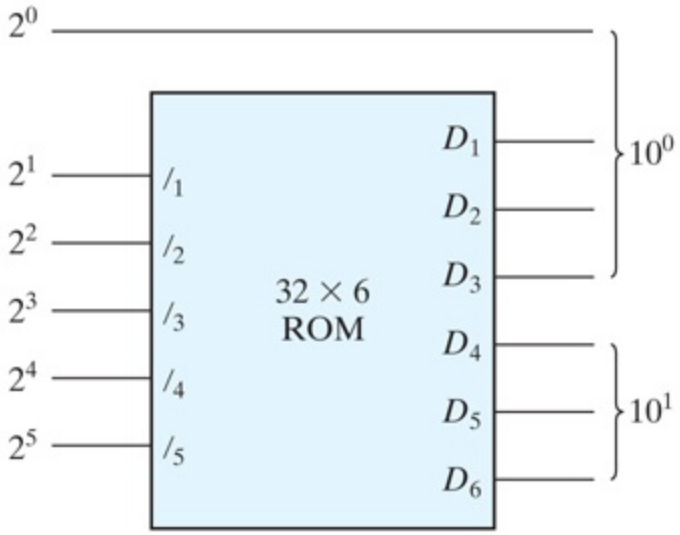


FIGURE P7.17

18. 7.18 Specify the size of a ROM (number of words and number of bits per word) that will accommodate the truth table for the following combinational circuit components:

    1. a binary multiplier that multiplies two 4-bit binary words,

    2. a 4-bit adder–subtractor,

    3. a quadruple two-to-one-line multiplexer with common select and enable inputs, and

    4. a BCD-to-seven-segment decoder with an enable input.

19. 7.19 Tabulate the PLA programming table for the four Boolean functions listed below. Minimize the numbers of product terms.

    A(x, y, z) = Σ(1, 3, 5, 6) B(x, y, z) = Σ(0, 1, 6, 7) C(x, y, z) = Σ(3, 5) D(x, y, z) = Σ(1, 2, 4, 5, 7)

20. 7.20* Tabulate the truth table for an 8×4 ROM that implements the Boolean functions

    A(x, y, z) = Σ(0, 3, 4, 6) B(x, y, z) = Σ(0, 1, 4, 7) C(x, y, z) = Σ(1, 5) D(x, y, z) = Σ(0, 1, 3, 5, 7)

Considering now the ROM as a memory. Specify the memory contents at addresses 1 and 4.

21. 7.21 Derive the PLA programming table for the combinational circuit that squares a three-bit number. Minimize the number of product terms. (See Fig. 7.12 for the equivalent ROM implementation.)

22. 7.22 Derive the ROM programming table for the combinational circuit that squares a 4-bit number. Minimize the number of product terms.

23. 7.23 List the PLA programming table for the BCD-to-excess-3-code converter whose Boolean functions are simplified in Fig. 4.3.

24. 7.24 Repeat Problem 7.23, using a PAL.

25. 7.25* The following is a truth table of a three-input, four-output combinational circuit:

Inputs			**Outputs**			
x	***y***	***z***	***A***	***B***	***C***	***D***
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	1	1	1	0

1 0 1 0 0 0 1

1 1 0 1 0 1 0

1 1 1 0 1 1 1

Tabulate the PAL programming table for the circuit, and mark the fuse map in a PAL diagram similar to the one shown in Fig. 7.17.

26. 7.26 Using the registered macrocell of Fig. 7.19, show the fuse map for a sequential circuit with two inputs *x* and *y* and one flip-flop *A* described by the input equation

    $DA=x \oplus y \oplus A$

27. 7.27 Modify the PAL diagram of Fig. 7.16 by including three clocked *D*-type flip-flops between the OR gates and the outputs, as in Fig. 7.19. The diagram should conform with the block diagram of a sequential circuit. The modification will require three additional buffer–inverter gates and six vertical lines for the flip-flop outputs to be connected to the AND array through programmable connections. Using the modified registered PAL diagram, show the fuse map that will implement a three-bit binary counter with an output carry.

28. 7.28 Draw a PLA circuit to implement the functions

    $F1=A'B+AC+A'BC'$ $F2=(AC+AB+BC)'$

29. 7.29 Develop the programming table for the PLA described in Problem 7.26.

30. 7.30 The memory modeled in HDL Example 7.1 exhibits asynchronous behavior. Write a memory model that is synchronized by a clock signal.

REFERENCES

- 1. Hamming, R. W. 1950. Error Detecting and Error Correcting Codes. *Bell Syst. Tech. J.* 29: 147–160.
- 2. Kitson, B. 1984. *Programmable Array Logic Handbook*. Sunnyvale, CA: Advanced Micro Devices.
- 3. Lin, S. and D. J. Costello, jr. 2004. *Error Control Coding*, 2nd ed., Englewood Cliffs, NJ: Prentice-Hall.
- 4. *Memory Components Handbook*. 1986. Santa Clara, CA: Intel.
- 5. Nelson, V. P., H. T. Nagle, J. D. Irwin, and B. D. Carroll. 1995. *Digital Logic Circuit Analysis and Design*. Upper Saddle River, NJ: Prentice Hall.
- 6. *The Programmable Logic Data Book*, 2nd ed., 1994. San Jose, CA: Xilinx, Inc.
- 7. Tocci, R. J. and N. S. Widmer. 2004. *Digital Systems Principles and Applications*, 9th ed., Upper Saddle River, NJ: Prentice Hall.
- 8. Trimberger, S. M. 1994. *Field Programmable Gate Array Technology*. Boston: Kluwer Academic Publishers.
- 9. Wakerly, J. F. 2006. *Digital Design: Principles and Practices*, 4th ed., Upper Saddle River, NJ: Prentice Hall.

WEB SEARCH TOPICS

- Ferroelectric RAM (FeRAM)
- FPGA
- Gate array
- Phase-Lock Loop
- Programmable array logic
- Programmable logic data book
- RAM
- ROM
- Transceiver