CHAPTER

6

Registers and Register Transfers

In Chapter 3, we studied combinational functional blocks, and in Chapter 4 we examined sequential circuits. Now, we bring the two ideas together and present sequential functional blocks, generally referred to as registers and counters. In Chapter 4, the circuits that were analyzed or designed did not have any particular structure, and the number of flip-flops was small. In contrast, the circuits we consider here have more structure, with multiple stages or cells that are identical or close to identical, making expansion very simple. Registers are particularly useful for storing information during the processing of data, and counters assist in sequencing the processing.

In a digital system, a datapath and a control unit are frequently present at the upper levels of the design hierarchy. A *datapath* consists of processing logic and a collection of registers that performs data processing. A *control unit* is made up of logic that determines the sequence of data-processing operations performed by the datapath. Register transfer notation describes elementary data-processing actions referred to as *microoperations*. *Register transfers* move information between registers, between registers and memory, and through processing logic. Dedicated transfer hardware using multiplexers and shared transfer hardware called *buses* implement these movements of data. The design of the control unit for controlling register transfers is also covered in this chapter. A design procedure for digital systems as combinations of register transfer logic and control logic brings together much of what we have studied thus far.

In the generic computer at the beginning of Chapter 1, registers are used extensively for temporary storage of data in areas aside from memory. Registers of this kind are often large, with at least 32 bits. Overall, sequential functional blocks are used widely in the generic computer. In particular, the CPU and FPU parts of the processor each contain large numbers of registers that are involved in register transfers and execution of microoperations. It is in the CPU and the FPU that data transfers, additions, subtractions, and other microoperations take place. Finally, the connections shown between various electronic parts of the computer are buses, which we discuss for the first time in this chapter.

6-1 Registers and Load Enable

A register includes a set of flip-flops. Since each flip-flop is capable of storing one bit of information, an n-bit register, composed of n flip-flops, is capable of storing n bits of binary information. By the broadest definition, a *register* consists of a set of flip-flops, together with gates that implement their state transitions. This broad definition includes the various sequential circuits considered in Chapter 4. More commonly, the term *register* is applied to a set of flip-flops, possibly with added combinational gates, that perform data-processing tasks. The flip-flops hold data, and the gates determine the new or transformed data to be transferred into the flip-flops.

A *counter* is a register that goes through a predetermined sequence of states upon the application of clock pulses. The gates in the counter are connected in a way that produces the prescribed sequence of binary states. Although counters are a special type of registers, it is common to differentiate them from registers.

Registers and counters are sequential functional blocks that are used extensively in the design of digital systems in general and in digital computers in particular. Registers are useful for storing and manipulating information; counters are employed in circuits that sequence and control operations in a digital system.

The simplest register is one that consists of only flip-flops without external gates. Figure 6-1(a) shows such a register constructed from four D-type flip-flops. The common *Clock* input triggers all flip-flops on the rising edge of each pulse, and the binary information available at the four D inputs is transferred into the 4-bit register. The four Q outputs can be sampled to obtain the binary information stored in the register. The $\overline{Clear}$ input goes to the $\overline{R}$ inputs of all four flip-flops and is used to clear the register to all 0s prior to its clocked operation. This input is labeled $\overline{Clear}$ rather than *Clear*, since a 0 must be applied to reset the flip-flops asynchronously. Activation of the asynchronous $\overline{R}$ inputs to flip-flops during normal clocked operation can lead to circuit designs that are highly delay dependent and that can, therefore, malfunction. Thus, we maintain $\overline{Clear}$ at logic 1 during normal clocked operation, allowing it to be logic 0 only when a system reset is desired. We note that the ability to clear a register to all 0s is optional; whether a clear operation is provided depends upon the use of the register in the system.

The transfer of new information into a register is referred to as *loading* the register. If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done in parallel. A positive clock transition applied to the *Clock* input of the register of Figure 6-1(a) loads all four D inputs into the flip-flops in parallel.

Figure 6-1(b) shows a symbol for the register in Figure 6-1(a). This symbol permits the use of the register in a design hierarchy. It has all inputs to the logic circuit on its left and all outputs from the circuit on the right. The inputs include the clock input with the dynamic indicator to represent positive-edge triggering of the flip-flops. We note that the name *Clear* appears inside the symbol, with a bubble in the signal line on the outside of the symbol. This notation indicates that application of a logic 0 to the signal line activates the clear operation on the flip-flops in the register. If the signal line were labeled outside the symbol, the label would be $\overline{Clear}$.

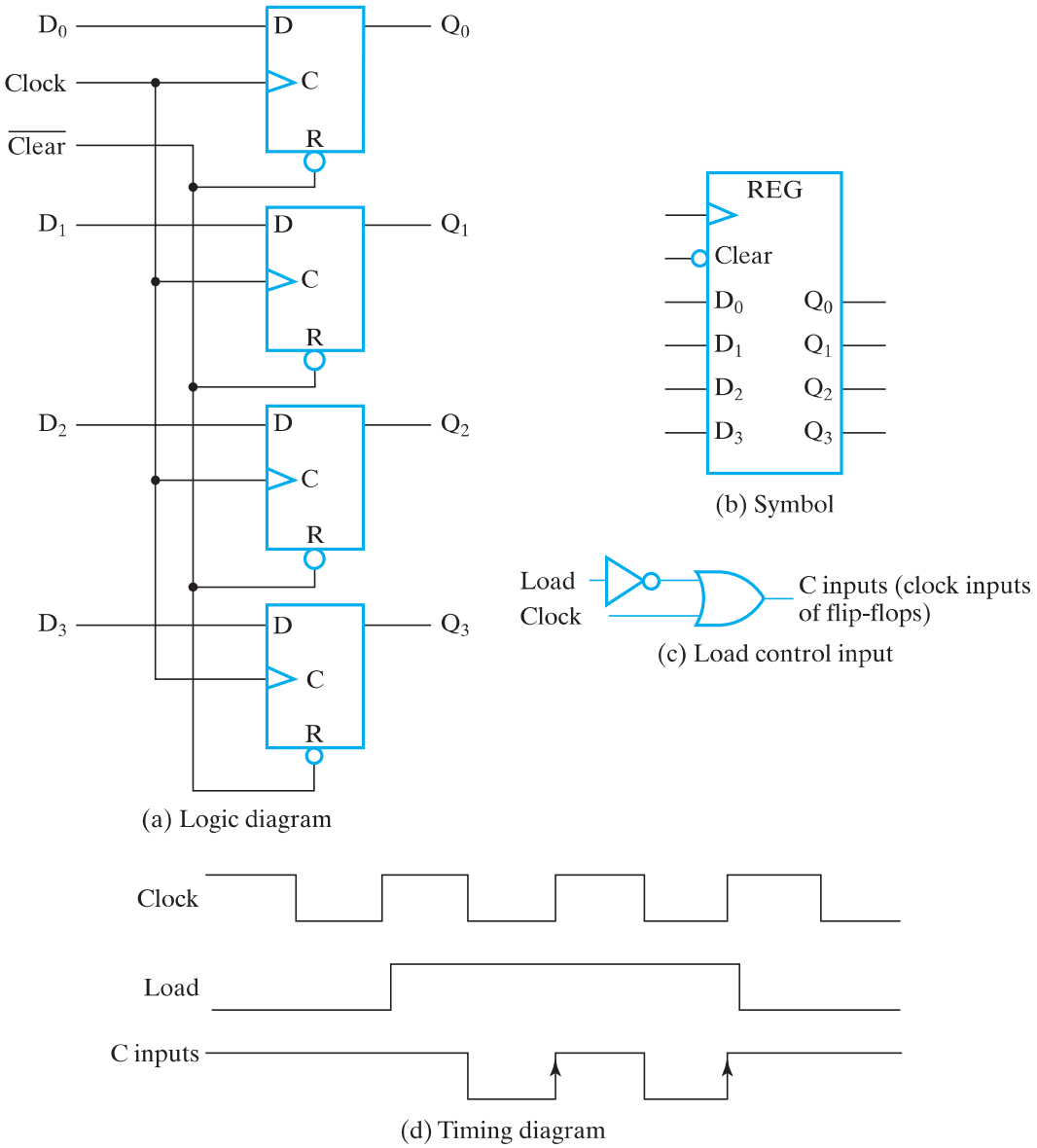


FIGURE 6-1
4-Bit Register

Register with Parallel Load

Most digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied to all flip-flops and registers in the system. In effect, the master clock acts like a heart that supplies a constant beat to all parts of the system. For the design in Figure 6-1(a), the clock can be prevented from reaching the clock input to the circuit if the contents of the register are to be left unchanged. Thus, a separate control signal is used to control the clock cycles during which clock pulses are to have an effect on the register. The clock pulses are prevented from reaching the register when its content is not to be changed. This approach can be implemented with a load control input *Load* combined with the clock, as shown in

Figure 6-1(c). The output of the OR gate is applied to the *C* inputs of the register flip-flops. The equation for the logic shown is

$$C\ inputs = \overline{Load} + Clock$$

When the *Load* signal is 1, *C inputs* = *Clock*, so the register is clocked normally, and new information is transferred into the register on the positive transitions of the clock. When the *Load* signal is 0, *C inputs* = 1. With this constant input applied, there are no positive transitions on *C inputs*, so the contents of the register remain unchanged. The effect of the *Load* signal on the signal *C inputs* is shown in Figure 6-1(d). Note that the clock pulses that appear on *C inputs* are pulses to 0, which end with the positive edge that triggers the flip-flops. These pulses and edges appear when *Load* is 1 and are replaced by a constant 1 when *Load* is 0. In order for this circuit to work correctly, *Load* must be constant at the correct value, either 0 or 1, throughout the interval when *Clock* is 0. One situation in which this occurs is if *Load* comes from a flip-flop that is triggered on a positive edge of *Clock*, a normal circumstance if all flip-flops in the system are positive-edge triggered. Since the clock is turned on and off at the register *C* inputs by the use of a logic gate, the technique is referred to as *clock gating*.

Inserting gates in the clock pulse path produces different propagation delays between *Clock* and the inputs of flip-flops with and without clock gating. If the clock signals arrive at different flip-flops or registers at different times, *clock skew* is said to exist. But to have a truly synchronous system, we must ensure that all clock pulses arrive simultaneously throughout the system so that all flip-flops trigger at the same time. For this reason, in routine designs, control of the operation of the register without using clock gating is advisable. Otherwise, delays must be controlled to drive the clock skew as close to zero as possible. This is applicable in aggressive low-power or high-speed designs.

A 4-bit register with a control input *Load* that is directed through gates into the *D* inputs of the flip-flops, instead of through the *C* inputs, is shown in Figure 6-2(c). This register is based on a bit cell shown in Figure 6-2(a) consisting of a 2-to-1 multiplexer and a *D* flip-flop. The signal *EN* selects between the data bit *D* entering the cell and the value *Q* at the output of the cell. For $EN = 1$, *D* is selected and the cell is loaded. For $EN = 0$, *Q* is selected and the output is loaded back into the flip-flop, preserving its current state. The feedback connection from output to input of the flip-flop is necessary because the *D* flip-flop, unlike other flip-flop types, does not have a "no change" input condition: With each clock pulse, the *D* input determines the next state of the output. To leave the output unchanged, it is necessary to make the *D* input equal to the present value of the output. The logic in Figure 6-2(a) can be viewed as a new type of *D* flip-flop, a *D flip-flop with enable*, having the symbol shown in Figure 6-2(b).

The register is implemented by placing four *D* flip-flops with enables in parallel and connecting the *Load* input to the *EN* inputs. When *Load* is 1, the data on the four inputs is transferred into the register with the next positive clock edge. When *Load* is 0, the current value remains in the register at the next positive clock edge. Note that the clock pulses are applied continuously to the *C* inputs. *Load* determines whether the next pulse accepts new information or leaves the information in the register intact. The transfer of information from inputs to register is

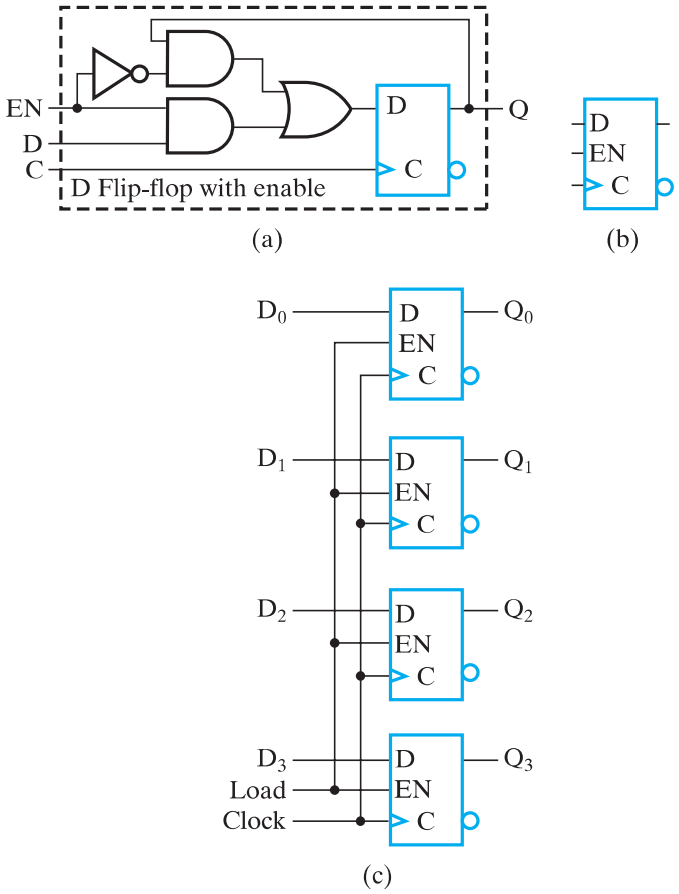


FIGURE 6-2
4-Bit Register with Parallel Load

done simultaneously for all four bits during a single positive pulse transition. This method of transfer is traditionally preferred over clock gating, since it avoids clock skew and the potential for malfunctions of the circuit.

6-2 Register Transfers

A digital system is a sequential circuit made up of interconnected flip-flops and gates. In Chapter 4, we learned that sequential circuits can be specified by means of state tables. To specify a large digital system with state tables is very difficult, if not impossible, because the number of states is prohibitively large. To overcome this difficulty, digital systems are designed using a modular, hierarchical approach. The system is partitioned into subsystems or modules, each of which performs some functional task. The modules are constructed hierarchically from functional blocks such as registers, counters, decoders, multiplexers, buses, arithmetic elements, flip-flops, and primitive gates. The various subsystems communicate with data and control signals to form a digital system.

In most digital system designs, we partition the system into two types of modules: a *datapath*, which performs data-processing operations, and a *control unit*,

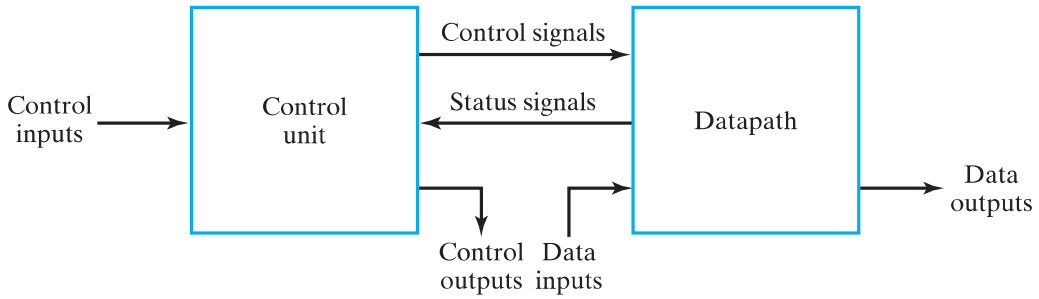


FIGURE 6-3
Interaction Between Datapath and Control Unit

which determines the sequence of those operations. Figure 6-3 shows the general relationship between a datapath and a control unit. *Control signals* are binary signals that activate the various data-processing operations. To activate a sequence of such operations, the control unit sends the proper sequence of control signals to the datapath. The control unit, in turn, receives status bits from the datapath. These status bits describe aspects of the state of the datapath. The status bits are used by the control unit in defining the specific sequence of the operations to be performed. Note that the datapath and control unit may also interact with other parts of a digital system, such as memory and input–output logic, through the paths labeled data inputs, data outputs, control inputs, and control outputs.

Datapaths are defined by their registers and the operations performed on binary data stored in the registers. Examples of register operations are load, clear, shift, and count. The registers are assumed to be basic components of the digital system. The movement of the data stored in registers and the processing performed on the data are referred to as *register transfer operations*. The register transfer operations of digital systems are specified by the following three basic components:

1. the set of registers in the system,
2. the operations that are performed on the data stored in the registers, and
3. the control that supervises the sequence of operations in the system.

A register has the capability to perform one or more *elementary operations* such as load, count, add, subtract, and shift. For example, a right-shift register is a register that can shift data to the right. A counter is a register that increments a number by one. A single flip-flop is a 1-bit register that can be set or cleared. In fact, by this definition, the flip-flops and closely associated gates of any sequential circuit can be called registers.

An elementary operation performed on data stored in registers is called a *microoperation.* Examples of microoperations are loading the contents of one register into another, adding the contents of two registers, and incrementing the contents of a register. A microoperation is usually, but not always, performed in parallel on a vector of bits during one clock cycle. The result of the microoperation may replace the previous binary data in the register. Alternatively, the result may be transferred to another register, leaving the previous data unchanged. The

sequential functional blocks introduced in this chapter are registers that implement one or more microoperations.

The control unit provides signals that sequence the microoperations in a prescribed manner. The results of a current microoperation may determine both the sequence of control signals and the sequence of future microoperations to be executed. Note that the term "microoperation," as used here, does not refer to any particular way of producing the control signals: specifically, it does not imply that the control signals are generated by a control unit based on a technique called microprogramming.

This chapter introduces registers, their implementations and register transfers using a simple register transfer language (RTL) to represent registers and specify the operations on their contents. The register transfer language uses a set of expressions and statements that resemble statements used in HDLs and programming languages. This notation can concisely specify part or all of a complex digital system such as a computer. The specification then serves as a basis for a more detailed design of the system.

6-3 Register Transfer Operations

We denote the registers in a digital system by uppercase letters (sometimes followed by numerals) that indicate the function of the register. For example, a register that holds an address for the memory unit is usually called an address register and can be designated by the name *AR*. Other designations for registers are *PC* for program counter, *IR* for instruction register, and *R*2 for register 2. The individual flip-flops in an n-bit register are typically numbered in sequence from 0 to $n - 1$, starting with 0 in the least significant (often the rightmost) position and increasing toward the most significant position. Since the 0 bit is on the right, this order can be referred to as *little-endian*. The reverse order, with bit 0 on the left, is referred to as *big-endian*.

Figure 6-4 shows representations of registers in block-diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in part (a) of the figure. The individual bits can be identified as in part (b). The numbering of bits represented by just the leftmost and rightmost values at the top of a register box is illustrated by a 16-bit register *R*2 in part (c). A 16-bit program counter, *PC*, is partitioned into two sections in part (d) of the figure. In this case, bits 0 through 7 are assigned the symbol *L* (for low-order byte), and bits 8 through 15 are assigned the symbol *H* (for high-order byte). The label *PC*(*L*), which

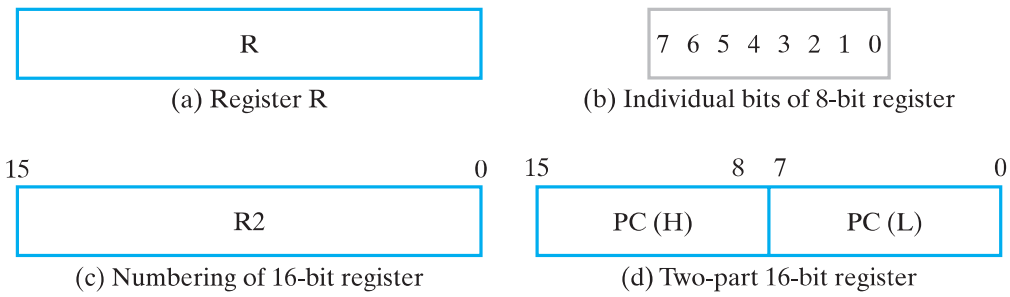


FIGURE 6-4
Block Diagrams of Registers

may also be written $PC(7:0)$, refers to the low-order byte of the register, and $PC(H)$ or $PC(15:8)$ refers to the high-order byte.

Data transfer from one register to another is designated in symbolic form by means of the replacement operator ($\leftarrow$). Thus, the statement

$$R2 \leftarrow R1$$

denotes a transfer of the contents of register $R1$ into register $R2$. Specifically, the statement designates the copying of the contents of $R1$ into $R2$. The register $R1$ is referred to as the *source* of the transfer and the register $R2$ as the *destination*. By definition, the contents of the source register do not change as a result of the transfer—only the contents of the destination register, $R2$, change.

A statement that specifies a register transfer implies that datapath circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want a given transfer to occur not for every clock pulse, but only for specific values of the control signals. This can be specified by a *conditional statement*, symbolized by the *if-then* form

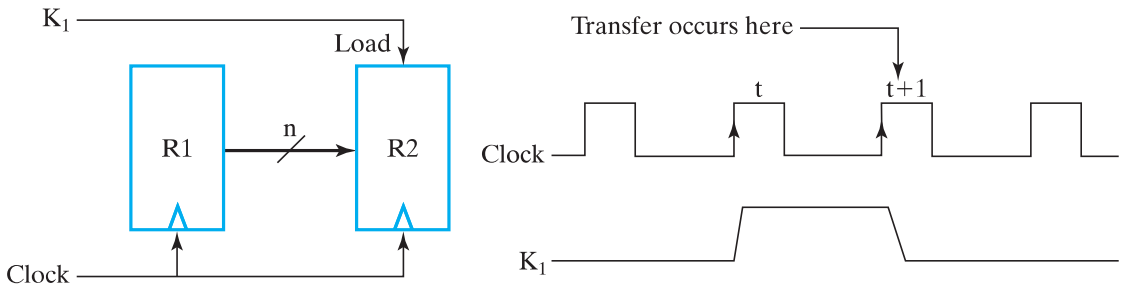
$$\text{if } (K_1 = 1) \text{ then } (R2 \leftarrow R1)$$

where K_1 is a control signal generated in the control unit. In fact, K_1 can be any Boolean function that evaluates to 0 or 1. A more concise way of writing the if-then form is

$$K_1\text{: } R2 \leftarrow R1$$

This control condition, terminated with a colon, symbolizes the requirement that the transfer operation be executed by the hardware only if $K_1 = 1$.

Every statement written in register-transfer notation presupposes a hardware construct for implementing the transfer. Figure 6-5 shows a block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n is used to indicate the number of bits in the register-transfer path from $R1$ to $R2$. When the width of the path is known, n is replaced by an actual number. Register $R2$ has a load control input that is activated by the control signal K_1. It is assumed that the signal is synchronized with the same clock as the one applied to the register. The flip-flops are assumed to be positive-edge



□ **FIGURE 6-5**
Transfer from $R1$ to $R2$ when $K_1 = 1$

TABLE 6-1
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$AR, R2, DR, IR$
Parentheses	Denotes a part of a register	$R2(1), R2(7{:}0), AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2,\ R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$

triggered by this clock. As shown in the timing diagram, K_1 is set to 1 on the rising edge of a clock pulse at time t. The next positive transition of the clock at time $t + 1$ finds $K_1 = 1$, and the inputs of $R2$ are loaded into the register in parallel. In this case, K_1 returns to 0 on the positive clock transition at time $t + 1$, so that only a single transfer from $R1$ to $R2$ occurs.

Note that the clock is not included as a variable in the register-transfer statements. It is assumed that all transfers occur in response to a clock transition. Even though the control condition K_1 becomes active at time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock, at time $t + 1$.

The basic symbols we use in register-transfer notation are listed in Table 6-1. Registers are denoted by an uppercase letter, possibly followed by one or more uppercase letters and numerals. Parentheses are used to denote a part of a register by specifying the range of bits in the register or by giving a symbolic name to a portion of the register. The left-pointing arrow denotes a transfer of data and the direction of transfer. A comma is used to separate two or more register transfers that are executed at the same time. For example, the statement

$$K_3\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers simultaneously for a positive clock edge at which $K_3 = 1$. Such an exchange is possible with registers made of flip-flops but presents a difficult timing problem with registers made of latches. Square brackets are used in conjunction with a memory transfer. The letter M designates a memory word, and the register enclosed inside the square brackets provides the address of the word in memory. This is explained in more detail in Chapter 8.

6-4 Register Transfers in VHDL and Verilog

Although there are some similarities, the register-transfer language used here differs from both VHDL and Verilog. In particular, different notation is used in each of the three languages. Table 6-2 compares the notation for many identical or similar register-transfer operations in the three languages. As you study this chapter and others to follow, this table will assist you in relating descriptions in the text RTL to the corresponding descriptions in VHDL or Verilog.

TABLE 6-2
Textbook RTL, VHDL, and Verilog Symbols for Register Transfers

Operation	Text RTL	VHDL	Verilog
Combinational assignment	=	<= (concurrent)	assign = (nonblocking)
Register transfer	←	<= (concurrent)	<= (nonblocking)
Addition	+	+	+
Subtraction	−	−	−
Bitwise AND	∧	and	&
Bitwise OR	∨	or	|
Bitwise XOR	⊕	xor	^
Bitwise NOT	– (overline)	not	~
Shift left (logical)	Sl	sll	<<
Shift right (logical)	Sr	srl	>>
Vectors/registers	A(3:0)	A(3 down to 0)	A[3:0]
Concatenation	‖	&	{ , }

6-5 MICROOPERATIONS

A microoperation is an elementary operation performed on data stored in registers or in memory. The microoperations most often encountered in digital systems are of four types:

1. *Transfer* microoperations, which transfer binary data from one register to another.
2. *Arithmetic* microoperations, which perform arithmetic operations on data in registers.
3. *Logic* microoperations, which perform bit manipulation on data in registers.
4. *Shift* microoperations, which shift data in registers.

A given microoperation may be of more than one type. For example, a 1s complement operation is both an arithmetic microoperation and a logic microoperation.

Transfer microoperations were introduced in the previous section. This type of microoperation does not change the binary data bits as they move from the source register to the destination register. The other three types of microoperations can produce new binary data and, hence, new information. In digital systems, basic sets of operations are used to form sequences that implement more complicated operations. In this section, we define a basic set of microoperations, symbolic notation for these microoperations, and descriptions of the digital hardware that implements them.

Arithmetic Microoperations

We define the basic arithmetic microoperations as add, subtract, increment, decrement, and complement. The statement

$$R0 \leftarrow R1 + R2$$

specifies an add operation. It states that the contents of register $R2$ are to be added to the contents of register $R1$ and the sum transferred to register $R0$. To implement this statement with hardware, we need three registers and a combinational component that performs the addition, such as a parallel adder. The other basic arithmetic operations are listed in Table 6-3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify 2s complement subtraction by the statement

$$R0 \leftarrow R1 + \overline{R2} + 1$$

where $\overline{R2}$ specifies the 1s complement of $R2$. Adding 1 to $\overline{R2}$ gives the 2s complement of $R2$. Finally, adding the 2s complement of $R2$ to the contents of $R1$ is equivalent to $R1 - R2$.

The increment and decrement microoperations are symbolized by a plus-one and minus-one operation, respectively. These operations are implemented by using a special combinational circuit, an adder–subtractor, or a binary up–down counter with parallel load.

Multiplication and division are not listed in Table 6-3. Multiplication can be represented by the symbol * and division by /. These two operations are not included in the basic set of arithmetic microoperations because they are assumed to be implemented by sequences of basic microoperations. However, multiplication can be considered as a microoperation if implemented by a combinational circuit. In such a case, the result is transferred into a destination register at the clock edge after all signals have propagated through the entire combinational circuit.

TABLE 6-3
Arithmetic Microoperations

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1s complement)
$R2 \leftarrow \overline{R2} + 1$	2s complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2s complement of $R2$ transferred to $R0$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

There is a direct relationship between the statements written in register-transfer notation and the registers and digital functions required for their implementation. To illustrate, consider the following two statements:

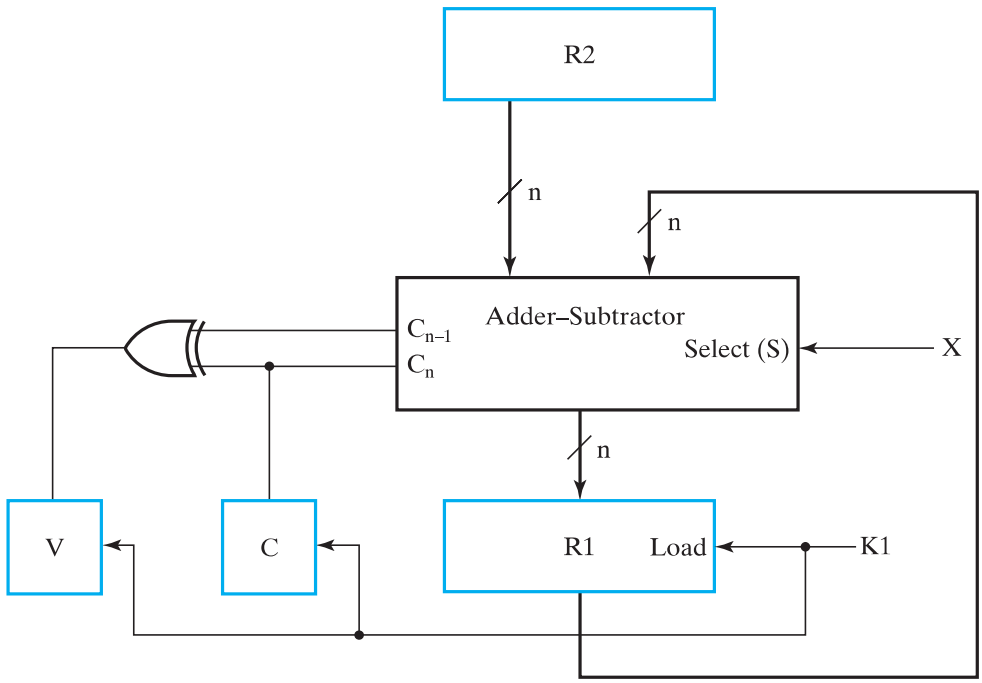
$$\overline{X}K_1: R1 \leftarrow R1 + R2$$
$$XK_1: R1 \leftarrow R1 + \overline{R2} + 1$$

Control variable K_1 activates an operation to add or subtract. If, at the same time, control variable X is equal to 0, then $\overline{X}K_1 = 1$,and the contents of $R2$ are added to the contents of $R1$. If X is equal to 1, then $XK_1 = 1$, and the contents of $R2$ are subtracted from the contents of $R1$. Note that the two control conditions are Boolean functions and reduce to 0 when $K_1 = 0$, a condition that inhibits the execution of both operations simultaneously.

A block diagram, showing the implementation of the preceding two statements, is given in Figure 6-6. An n-bit adder–subtractor, similar to the one shown in Figure 3-45, receives its input data from registers $R1$ and $R2$. The sum or difference is applied to the inputs of $R1$. The *Select* input S of the adder–subtractor selects the operation in the circuit. When $S = 0$, the two inputs are added, and when $S = 1$, $R2$ is subtracted from $R1$. Applying the control variable X to the S input activates the required operation. The output of the adder–subtractor is loaded into $R1$ on any positive clock edge at which $\overline{X}K_1 = 1$ or $XK_1 = 1$. We can simplify this to just K_1, since

$$\overline{X}K_1 + XK_1 = (\overline{X} + X)K_1 = K_1$$

Thus, the control variable X selects the operation, and the control variable K_1 loads the result into $R1$.



□ **FIGURE 6-6**
Implementation of Add and Subtract Microoperations

TABLE 6-4
Logic Microoperations

Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1s complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

Based on the discussion of overflow in Section 3-11, the overflow output is transferred to flip-flop V, and the output carry from the most significant bit of the adder–subtractor is transferred to flip-flop C, as shown in Figure 6-6. These transfers occur when $K_1 = 1$ and are not represented in the register-transfer statements;—if desired, we could show them as additional simultaneous transfers.

Logic Microoperations

Logic microoperations are useful in manipulating the bits stored in a register. These operations consider each bit in the register separately and treat it as a binary variable. The symbols for the four basic logic operations are shown in Table 6-4. The NOT microoperation, represented by a bar over the source register name, complements all bits and thus is the same as the 1s complement. The symbol $\wedge$ is used to denote the AND microoperation and the symbol $\vee$ to denote the OR microoperation. By using these special symbols, we can distinguish between the add microoperation represented by a $+$ and the OR microoperation. Although the $+$ symbol has two meanings, we can distinguish between them by noting where the symbol occurs. If the $+$ occurs in a microoperation, it denotes addition. If the $+$ occurs in a control or Boolean function, it denotes OR. The OR microoperation will always use the $\vee$ symbol. For example, in the statement

$$(K_1 + K_2): R1 \leftarrow R2 + R3,\ R4 \leftarrow R5 \vee R6$$

the $+$ between K_1 and K_2 is an OR operation between two variables in a control condition. The $+$ between $R2$ and $R3$ specifies an add microoperation. The OR microoperation is designated by the symbol $\vee$ between registers $R5$ and $R6$. The logic microoperations can be easily implemented with a group of gates, one for each bit position. The NOT of a register of n bits is obtained with n NOT gates in parallel. The AND microoperation is obtained using a group of n AND gates, each receiving a pair of corresponding inputs from the two source registers. The outputs of the AND gates are applied to the corresponding inputs of the destination register. The OR and exclusive-OR microoperations require a similar arrangement of gates.

The logic microoperations can change bit values, clear a group of bits, or insert new bit values into a register. The following examples show how the bits stored in the 16-bit register $R1$ can be selectively changed by using a logic microoperation and a logic operand stored in the 16-bit register $R2$.

The AND microoperation can be used for clearing one or more bits in a register to 0. The Boolean equations $X \cdot 0 = 0$ and $X \cdot 1 = X$ dictate that, when ANDed with 0, a binary variable X produces a 0, but when ANDed with 1, the variable remains unchanged. A given bit or group of bits in a register can be cleared to 0 if ANDed with 0. Consider the following example:

10101101 10101011	$R1$	(data)
00000000 11111111	$R2$	(mask)
00000000 10101011	$R1 \leftarrow R1 \wedge R2$	

The 16-bit logic operand in $R2$ has 0s in the high-order byte and 1s in the low-order byte. By ANDing the contents of $R2$ with the contents of $R1$, it is possible to clear the high-order byte of $R1$ and leave the bits in the low-order byte unchanged. Thus, the AND operation can be used to selectively clear bits of a register. This operation is sometimes called *masking out* the bits, because it masks or deletes all 1s in the *data* in $R1$, based on bit positions that are 0 in the *mask* provided in $R2$.

The OR microoperation is used to set one or more bits in a register. The Boolean equations $X + 1 = 1$ and $X + 0 = X$ dictate that, when ORed with 1, the binary variable X produces a 1, but when ORed with 0, the variable remains unchanged. A given bit or group of bits in a register can be set to 1 if ORed with 1. Consider the following example:

10101101 10101011	$R1$	(data)
11111111 00000000	$R2$	(mask)
11111111 10101011	$R1 \leftarrow R1 \vee R2$	

The high-order byte of $R1$ is set to all 1s by ORing it with all 1s in the $R2$ operand. The low-order byte remains unchanged because it is ORed with 0s.

The XOR (exclusive-OR) microoperation can be used to complement one or more bits in a register. The Boolean equations $X \oplus 1 = \overline{X}$ and $X \oplus 0 = X$ dictate that, when a binary variable X is XORed with 1, it is complemented, but when XORed with 0, the variable remains unchanged. By XORing a bit or group of bits in register $R1$ with 1s in selected positions in $R2$, it is possible to complement the bits in the selected positions in $R1$. Consider the following example:

10101101 10101011	$R1$	(data)
11111111 00000000	$R2$	(mask)
01010010 10101011	$R1 \leftarrow R1 \oplus R2$	

The high-order byte in $R1$ is complemented after the XOR operation with $R2$, and the low-order byte is unchanged.

TABLE 6-5
Examples of Shifts

Type	Symbolic Designation	Eight-Bit Examples: Source $R2$	Eight-Bit Examples: After Shift: Destination $R1$
Shift left	$R1 \leftarrow \text{sl } R2$	10011110	00111100
Shift right	$R1 \leftarrow \text{sr } R2$	11100101	01110010

Shift Microoperations

Shift microoperations are used for lateral movement of data. The contents of a source register can be shifted either right or left. A *left shift* is toward the most significant bit, and a *right shift* is toward the least significant bit. Shift microoperations are used in the serial transfer of data. They are also used for manipulating the contents of registers in arithmetic, logical, and control operations. The destination register for a shift microoperation may be the same as or different from the source register. We use strings of letters to represent the shift microoperations defined in Table 6-5. For example,

$$R0 \leftarrow \text{sr } R0, \quad R1 \leftarrow \text{sl } R2$$

are two microoperations that respectively specify a one-bit shift to the right of the contents of register $R0$ and a transfer of the contents of $R2$ shifted one bit to the left into register $R1$. The contents of $R2$ are not changed by this shift.

For a left-shift microoperation, we call the rightmost bit of the destination register the *incoming bit*. For a right-shift microoperation, we define the leftmost bit of the destination register as the incoming bit. The incoming bit may have different values, depending upon the type of shift microoperation. Here we assume that, for sr and sl, the incoming bit is 0, as shown in the examples in Table 6-5. The *outgoing bit* is the leftmost bit of the source register for the left-shift operation and the rightmost bit of the source register for the right-shift operation. For the left and right shifts shown, the outgoing bit value is simply discarded. In Chapter 9, we will explore other types of shifts that treat incoming and outgoing bits differently.

6-6 Microoperations on a Single Register

This section covers the implementation of one or more microoperations with a single register as the destination of all primary results. The single register may also serve as a source of an operand for binary and unary operations. Due to the close ties between a single set of storage elements and the microoperations, the

combinational logic implementing the microoperations is assumed to be a part of the register and is called *dedicated logic* of the register. This is in contrast to logic which is shared by multiple destination registers. In this case, the combinational logic implementing the microoperations is called *shared logic* for the set of destination registers.

The combinational logic implementing the microoperations described in the previous section can use one or more functional blocks from Chapter 3 or can be designed specifically for the register. Initially, functional blocks will be used in combination with D flip-flops or D flip-flops with enable. A simple technique using multiplexers for selection is introduced to allow multiple microoperations on a single register. Next, single- and multiple-function registers that perform shifting and counting are designed.

Multiplexer-Based Transfers

There are occasions when a register receives data from two or more different sources at different times. Consider the following conditional statement having an *if-then-else* form:

$$\text{if } (K_1 = 1) \text{ then } (R0 \leftarrow R1) \text{ else if } (K_2 = 1) \text{ then } (R0 \leftarrow R2)$$

The value in register $R1$ is transferred to register $R0$ when control signal K_1 equals 1. When $K_1 = 0$, the value in register $R2$ is transferred to $R0$ when K_2 equals 1. Otherwise, the contents of $R0$ remains unchanged. The conditional statement may be broken into two parts using the following control conditions:

$$K_1\colon R0 \leftarrow R1, \quad \overline{K}_1 K_2\colon R0 \leftarrow R2$$

This specifies hardware connections from two registers, $R1$ and $R2$, to one common destination register $R0$. In addition, making a selection between two source registers must be based on values of the control variables K_1 and K_2.

The block diagram for a circuit with 4-bit registers that implements the conditional register-transfer statements using a multiplexer is shown in Figure 6-7(a). The quad 2-to-1 multiplexer selects between the two source registers. For $K_1 = 1$, $R1$ is loaded into $R0$, irrespective of the value of K_2. For $K_1 = 0$ and $K_2 = 1$, $R2$ is loaded into $R0$. When both K_1 and K_2 are equal to 0, the multiplexer selects $R2$ as the input to $R0$, but, because the control function, $K_2 + K_1$, connected to the LOAD input of $R0$ equals 0, the contents of $R0$ remain unchanged.

The detailed logic diagram for the hardware implementation is shown in Figure 6-7(b). The diagram uses functional block symbols based upon detailed logic for the registers in Figure 6-2 and for a quad 2-to-1 multiplexer from Chapter 3. Note that since this diagram represents just a part of a system, there are inputs and outputs that are not yet connected. Also, the clock is not shown in the block diagram, but is shown in the detailed diagram. It is important to relate the information given in a block diagram such as Figure 6-7(a) with the detailed

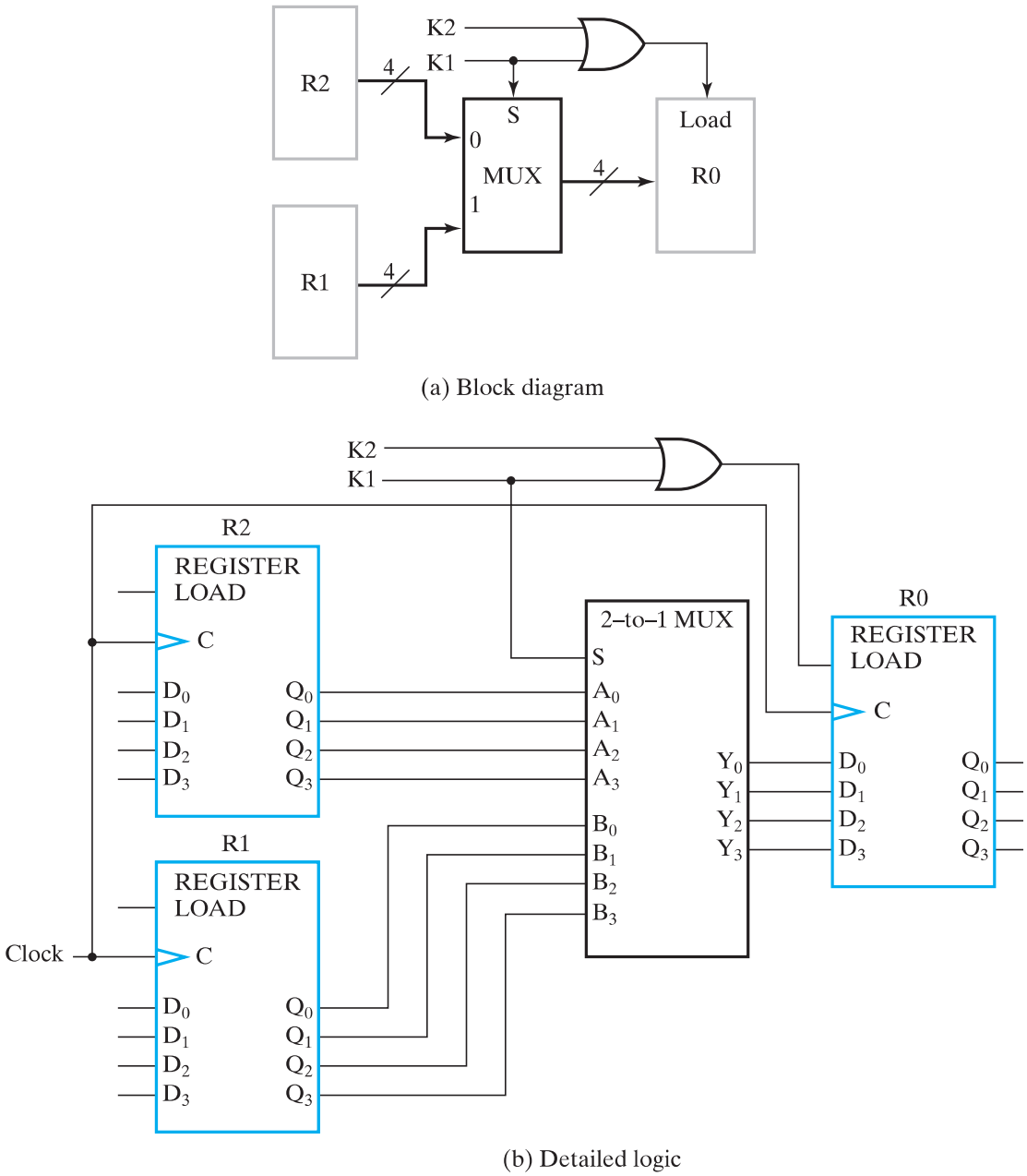


FIGURE 6-7
Use of Multiplexers to Select Between Two Registers

wiring connections in the corresponding logic diagram in Figure 6-7(b). In order to save space, we often omit the detailed logic diagrams in designs. However, it is possible to obtain a logic diagram with detailed wiring from the corresponding block diagram and a library of functional blocks. In fact, such a procedure is performed by computer programs used for automated logic synthesis.

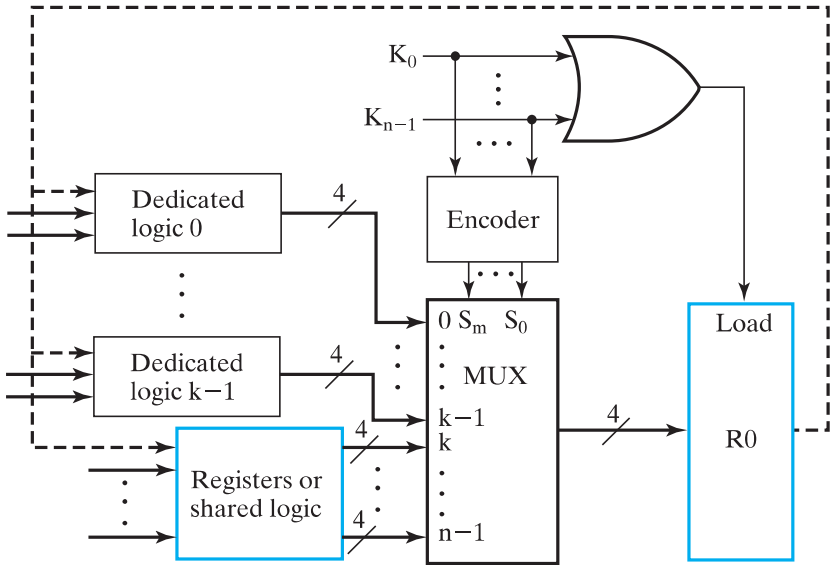


FIGURE 6-8
Generalization of Multiplexer Selection for *n* Sources

The preceding example can be generalized by allowing the multiplexer to have *n* sources and these sources to be register outputs or combinational logic implementing microoperations. This generalization results in the block diagram shown in Figure 6-8. The diagram assumes that each source is either the outputs of a register or of combinational logic implementing one or more microinstructions. In those cases in which the microoperations are dedicated to the register, the corresponding dedicated logic is included as a part of the register. In Figure 6-8, the first *k* sources are dedicated logic and the last $n - k$ sources are either registers or shared logic. The control signals that select a given source are either a single control variable or the OR of all control signals corresponding to the microoperations associated with the source. To force *R*0 to load for a microoperation, these control signals are ORed together to form the *Load* signal. Since it is assumed that only one of the control signals is 1 at any time, these signals must be encoded to provide the selection codes for the multiplexer. Two modifications to the given structure are possible. The control signals could be applied directly to a $2 \times n$ AND-OR circuit (i.e., a multiplexer with the decoder deleted). Alternatively, the control signals could already be encoded, omitting the use of the all-zero code, so that the OR gate still forms the *Load* signal correctly.

Shift Registers

A register capable of shifting its stored bits laterally in one or both directions is called a *shift register*. The logical configuration of a shift register consists of a chain of flip-flops, with the output of one flip-flop connected to the input of

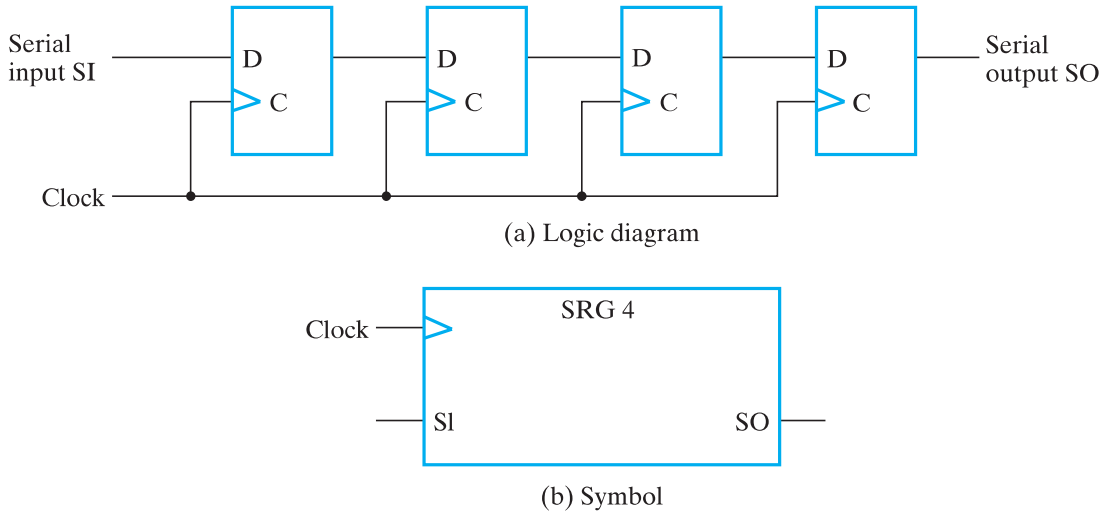


FIGURE 6-9
4-Bit Shift Register

the next flip-flop. All flip-flops have a common clock-pulse input that activates the shift.

The simplest possible shift register uses only flip-flops, as shown in Figure 6-9(a). The output of a given flip-flop is connected to the *D* input of the flip-flop at its right. The clock is common to all flip-flops. The *serial input SI* is the input to the leftmost flip-flop. The *serial output SO* is taken from the output of the rightmost flip-flop. A symbol for the shift register is given in Figure 6-9(b).

Sometimes it is necessary to control the register so that it shifts only on select positive clock edges. For the shift register in Figure 6-9, the shift can be controlled by connecting the clock through the logic shown in Figure 6-1(c), with *Shift* replacing *Load*. Again, due to clock skew, this is usually not the most desirable approach. Thus, we learn next that the shift operation can be controlled through the *D* inputs of the flip-flops rather than through the clock inputs *C*.

Shift Register with Parallel Load If all flip-flop outputs of a shift register are accessible, then information entered serially by shifting can be taken out in parallel from the flip-flop outputs. If a parallel load capability is also added to a shift register, then data entered in parallel can be shifted out serially. Thus, a shift register with accessible flip-flop outputs and parallel load can be used for converting incoming parallel data to outgoing serial data and vice versa.

The logic diagram for a 4-bit shift register with parallel load and the symbol for this register are shown in Figure 6-10. There are two control inputs, one for the shift and the other for the load. Each stage of the register consists of a *D* flip-flop, an OR gate, and three AND gates. The first AND gate enables the shift operation. The second AND gate enables the input data. The third AND gate restores the contents of the register when no operation is required.

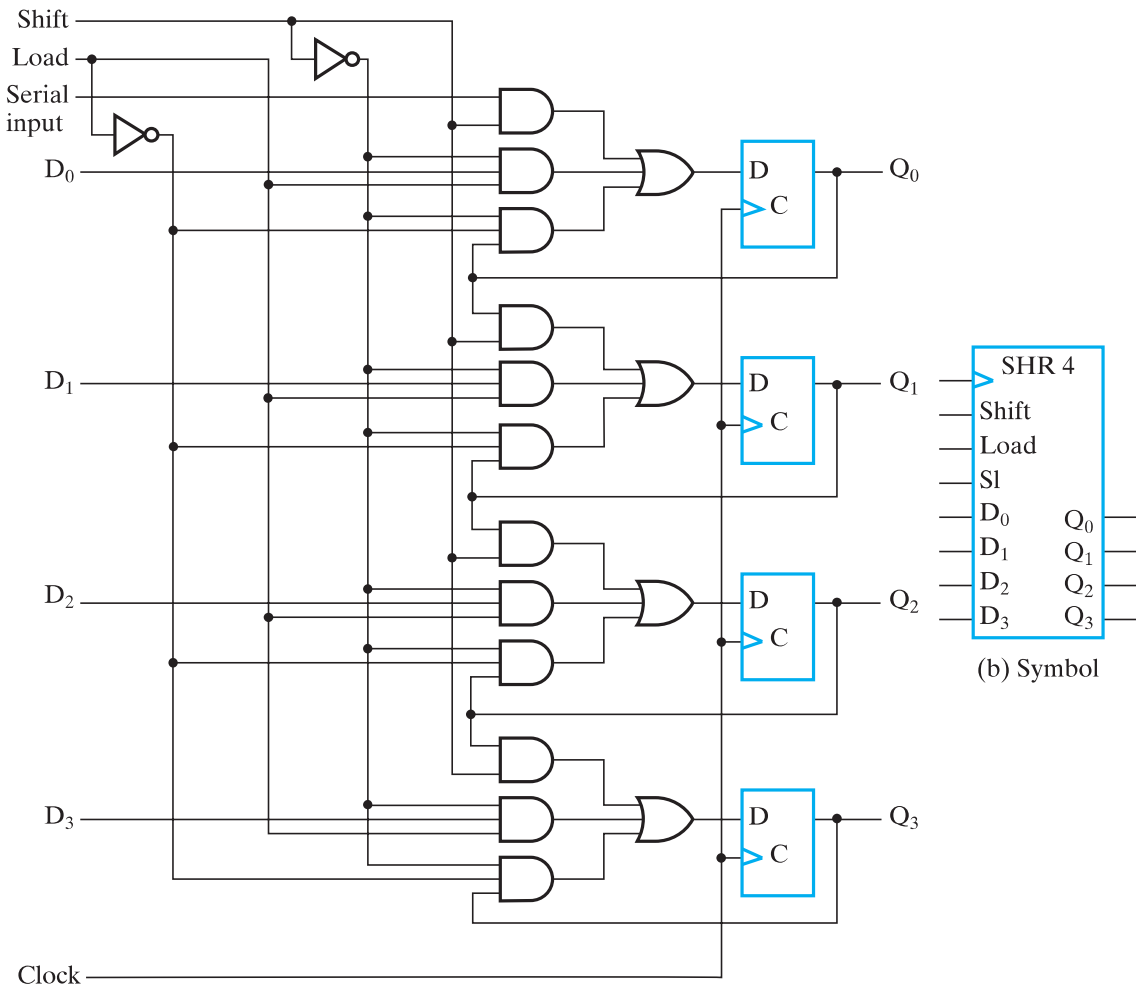


FIGURE 6-10
Shift Register with Parallel Load

The operation of this register is specified in Table 6-6 and is also given by the register transfers:

$$Shift\text{: } Q \leftarrow \text{sl}\, Q$$

$$\overline{Shift} \cdot Load\text{: } Q \leftarrow D$$

TABLE 6-6
Function Table for the Register of Figure 6-10

Shift	Load	Operation
0	0	No change (Hold)
0	1	Load parallel data
1	×	Shift left (down) from Q_0 to Q_3

The "No change" operation, also called "Hold", is implicit if neither of the conditions for transfers is satisfied. When both the shift and load control inputs are 0, the third AND gate in each stage is enabled, and the output of each flip-flop is applied to its own D input. A positive transition of the clock restores the contents to the register, and the output is unchanged. When the shift input is 0 and the load input is 1, the second AND gate in each stage is enabled, and the input D_i is applied to the D input of the corresponding flip-flop. The next positive clock transition transfers the parallel input data into the register. When the shift input is equal to 1, the first AND gate in each stage is enabled and the other two are disabled. Since the *Load* input is disabled by the $\overline{Shift}$ input on the second AND gate, we mark it with a don't-care condition in the *Shift* row of the table. When a positive edge occurs on the clock, the shift operation causes the data from the serial input SI to be transferred to flip-flop Q_0, the output of Q_0 to be transferred to flip-flop Q_1, and so on down the line. Note that because of the way the circuit is drawn, the shift occurs in the downward direction. If we rotate the page a quarter-turn clockwise, the register shifts from right to left.

Shift registers are often used to interface digital systems that are distant from each other. For example, suppose it is necessary to transmit an n-bit quantity between two points. If the distance is large, it is expensive to use n lines to transmit the n bits in parallel. It may be more economical to use a single line and transmit the information serially, one bit at a time. The transmitter loads the n-bit data in parallel into a shift register and then transmits the data serially along the common line. The receiver accepts the data serially into a shift register. When all n bits are accumulated, they can be taken in parallel from the outputs of the register. Thus, the transmitter performs a parallel-to-serial conversion of data, and the receiver does a serial-to-parallel conversion.

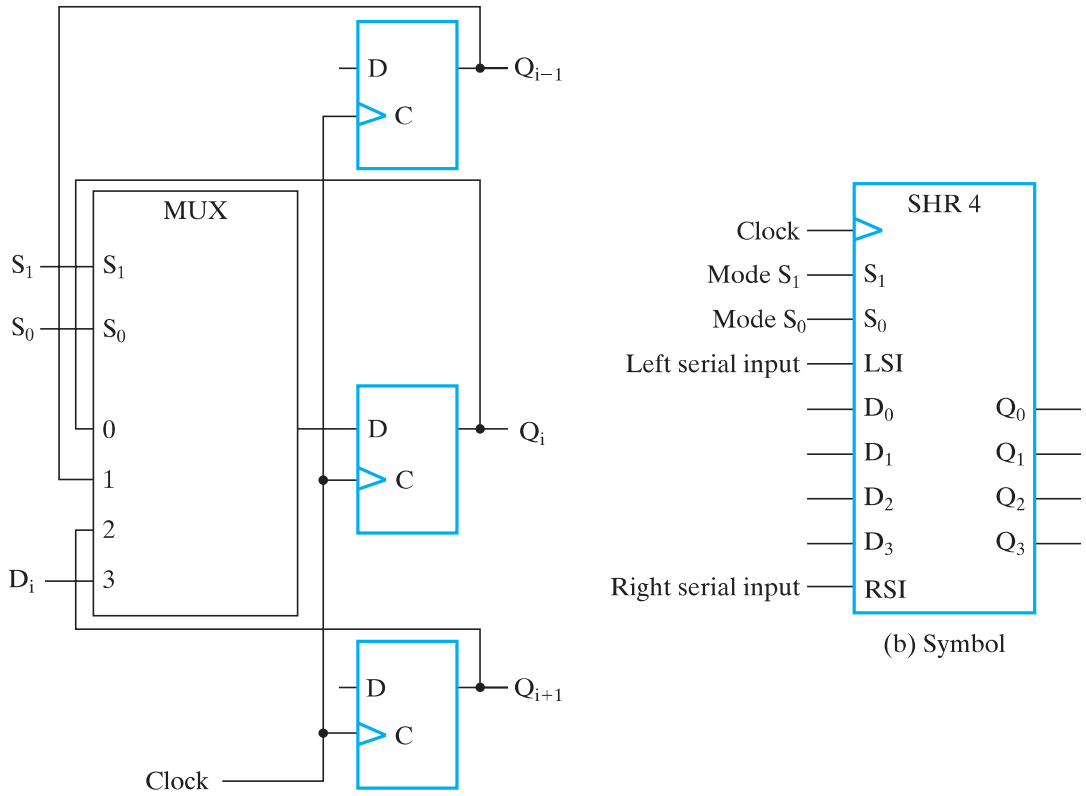
Bidirectional Shift Register A register capable of shifting in only one direction is called a *unidirectional shift register*. A register that can shift in both directions is a *bidirectional shift register*. It is possible to modify the circuit of Figure 6-10, by adding a fourth AND gate in each stage, for shifting the data in the upward direction. An investigation of the resultant circuit will reveal that the four AND gates, together with the OR gate in each stage, constitute a multiplexer with the selection inputs controlling the operation of the register.

One stage of a bidirectional shift register with parallel load is shown in Figure 6-11(a). Each stage consists of a D flip-flop and a 4–to–1-line multiplexer. The two selection inputs S_1 and S_0 select one of the multiplexer inputs to apply to the D flip-flop. The selection lines control the mode of operation of the register according to Table 6-7 and the register transfers:

$$\overline{S}_1 \cdot S_0\colon\ Q \leftarrow \mathrm{sl}\, Q$$

$$S_1 \cdot \overline{S}_0\colon\ Q \leftarrow \mathrm{sr}\, Q$$

$$S_1 \cdot S_0\colon\ Q \leftarrow D$$



(a) Logic diagram of one typical stage

(b) Symbol

FIGURE 6-11
Bidirectional Shift Register with Parallel Load

The "No Change" operation is implicit if none of the conditions for transfers is satisfied. When the mode control $S_1S_0 = 00$, input 0 of the multiplexer is selected. This forms a path from the output of each flip-flop into its own input. The next clock transition transfers the current stored value back into each flip-flop, and no change of state occurs. When $S_1S_0 = 01$, the terminal marked 1 on the multiplexer

TABLE 6-7
Function Table for the Register of Figure 6-11

Mode Control		Register Operation
S_1	S_0	
0	0	No change (Hold)
0	1	Shift left
1	0	Shift right
1	1	Parallel load

has a path to the D input of each flip-flop. These paths cause a shift-left operation, with the bits being moved toward the most significant bit (down in the figure). The serial input is transferred into the first stage, and the content of each stage, Q_{i-1}, is transferred into stage Q_i. When $S_1S_0 = 10$, a shift-right operation results in a second serial input that enters the last stage. In addition, the value in each stage Q_{i+1} is transferred into stage Q_i (up in the figure). Finally, when $S_1S_0 = 11$, the binary information on each parallel input line is transferred into the corresponding flip-flop, resulting in a parallel load.

Figure 6-11(b) shows a symbol for the bidirectional shift register from Figure 6-11(a). Note that both a left serial input (*LSI*) and a right serial input (*RSI*) are provided. If serial outputs are desired, Q_3 is used for left shift and Q_0 for right shift.

Ripple Counter

A register that goes through a prescribed sequence of distinct states upon the application of a sequence of input pulses is called a *counter*. The input pulses may be clock pulses or may originate from some other source, and they may occur at regular or irregular intervals of time. In our discussion of counters, we assume clock pulses, but other signals can be substituted for the clock. The sequence of states may follow the binary number sequence or any other prescribed sequence of states. A counter that follows the binary number sequence is called a *binary counter*. An n-bit binary counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$.

Counters are available in two categories: ripple counters and synchronous counters. In a ripple counter, the flip-flop output transitions serve as the sources for triggering the changes in other flip-flops. In other words, the C inputs of some of the flip-flops are triggered not by the common clock pulse, but rather by the transitions that occur on other flip-flop outputs. In a synchronous counter, the C inputs of all flip-flops receive the common clock pulse, and the change of state is determined from the present state of the counter. Synchronous counters are discussed in the next two subsections. Here we present the binary ripple counter and explain its operation.

The logic diagram of a 4-bit binary ripple counter is shown in Figure 6-12. The counter is constructed from D flip-flops connected such that the application of a positive edge to the C input of each flip-flop causes the flip-flop to complement its state. The complemented output of each flip-flop is connected to the C input of the next most significant flip-flop. The flip-flop holding the least significant bit receives the incoming clock pulses. Positive-edge triggering makes each flip-flop complement its value when the signal on its C input goes through a positive transition. The positive transition occurs when the complemented output of the previous flip-flop, to which C is connected, goes from 0 to 1. A 1-level signal on *Reset* driving the R inputs clears the register to all zeros asynchronously.

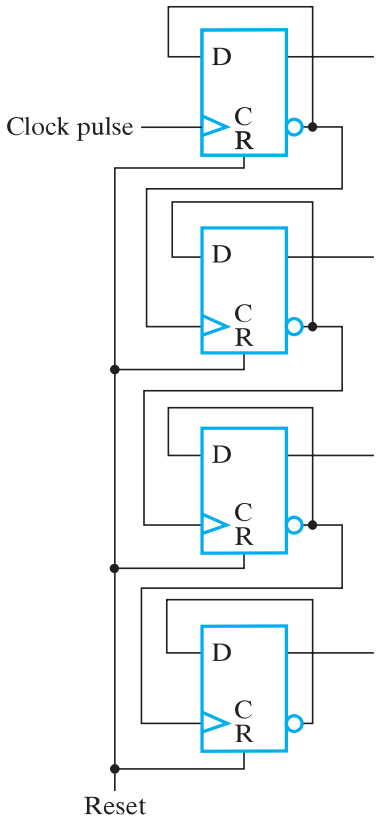


FIGURE 6-12
4-Bit Ripple Counter

To understand the operation of a binary ripple counter, let us examine the upward counting sequence given in the left half of Table 6-8. The count starts at binary 0 and increments by one with each count pulse. After the count of 15, the counter goes back to 0 to repeat the count. The least significant bit (Q_0) is complemented by each count pulse. Every time that Q_0 goes from 1 to 0, $\overline{Q}_0$ goes from 0 to 1, complementing Q_1. Every time that Q_1 goes from 1 to 0, it complements Q_2. Every time that Q_2 goes from 1 to 0, it complements Q_3, and so on for any higher-order bits in the ripple counter. For example, consider the transition from count 0011 to 0100. Q_0 is complemented with the count pulse positive edge. Since Q_0 goes from 1 to 0, it triggers Q_1 and complements it. As a result, Q_1, goes from 1 to 0, which complements Q_2, changing it from 0 to 1. Q_2 does not trigger Q_3, because $\overline{Q}_2$ produces a negative transition, and the flip-flops respond only to positive transitions. Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time. The counter goes from 0011 to 0010 (Q_0 from 1 to 0), then to 0000 (Q_1 from 1 to 0), and finally to 0100 (Q_2 from 0 to 1). The flip-flops change one at a time in quick succession as the signal propagates through the counter in a ripple fashion from one stage to the next.

A ripple counter that counts downward gives the sequence in the right half of Table 6-8. Downward counting can be accomplished by connecting the true output of each flip-flop to the C input of the next flip-flop.

The advantage of ripple counters is their simple hardware. Unfortunately, they are asynchronous circuits and, with added logic, can become circuits with delay

TABLE 6-8
Counting Sequence of Binary Counter

Upward Counting Sequence				Downward Counting Sequence			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	0
0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

dependence and unreliable operation. This is particularly true for logic that provides feedback paths from counter outputs back to counter inputs. Also, due to the length of time required for the ripple to finish, large ripple counters can be slow circuits. As a consequence, synchronous binary counters are favored in all but low-power designs, where ripple counters have an advantage. (See Problem 6-9.)

Synchronous Binary Counters

Synchronous counters, in contrast to ripple counters, have the clock applied to the *C* inputs of all flip-flops. Thus, the common clock pulse triggers all flip-flops simultaneously rather than one at a time, as in a ripple counter. A synchronous binary counter that counts up by 1 can be constructed from the incrementer in Figure 3-52 and *D* flip-flops, as shown in Figure 6-13(a). The carry output *CO* is added by not placing an x value on the C_4 output before the contraction of an adder to the incrementer in Figure 3-52. Output *CO* is used to extend the counter to more stages.

Note that the flip-flops trigger on the positive-edge transition of the clock. The polarity of the clock is not essential here, as it was for the ripple counter. The synchronous counter can be designed to trigger with either the positive or the negative clock transition.

Serial and Parallel Counters We will use the synchronous counter in Figure 6-13 to demonstrate two alternative designs for binary counters. In Figure 6-13(a), a chain of 2-input AND gates is used to provide information to each stage about the state of the prior stages in the counter. This is analogous to the carry logic in the ripple carry adder. A counter that uses such logic is said to have *serial gating* and is referred to

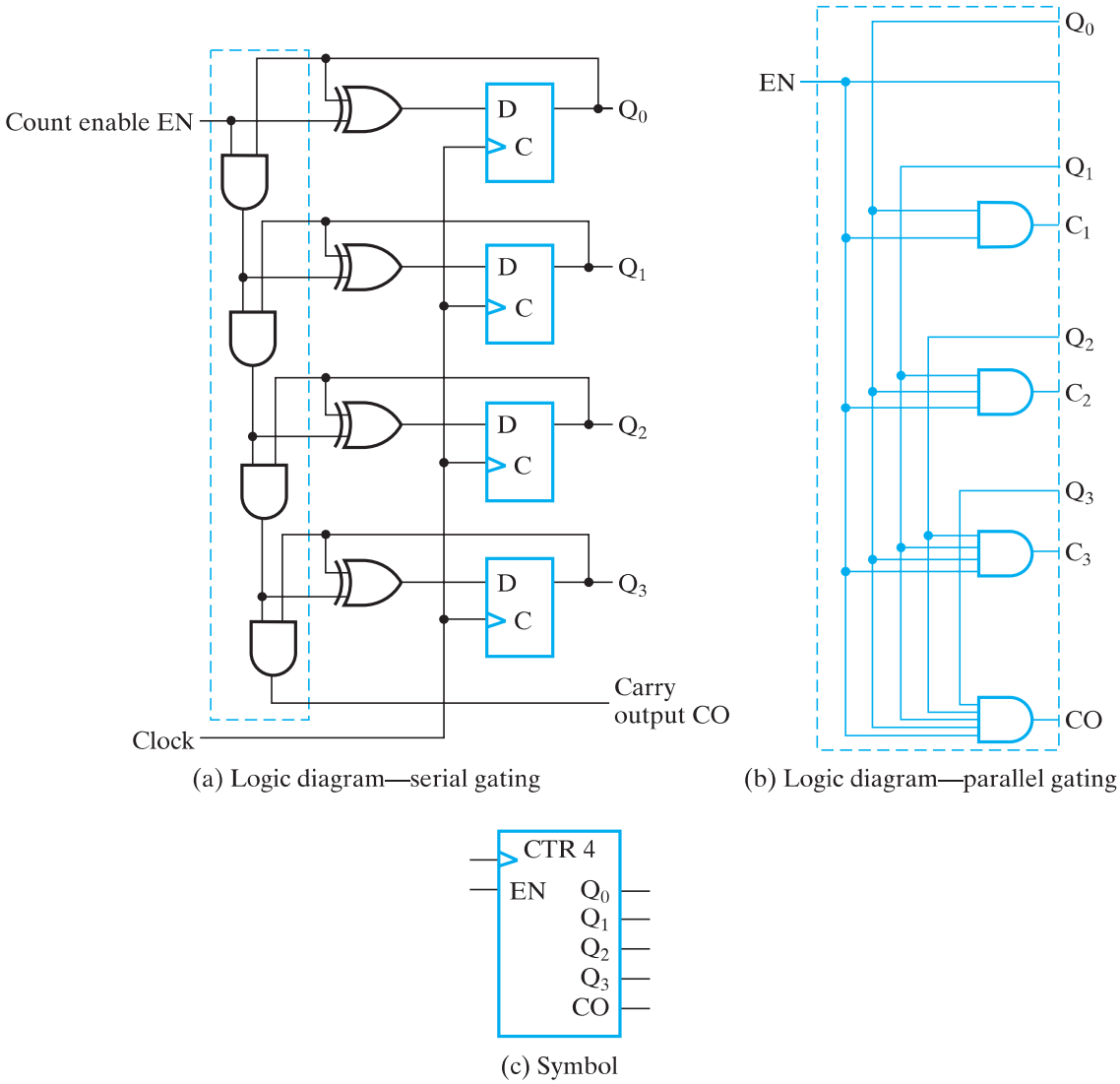


FIGURE 6-13
4-Bit Synchronous Binary Counter

as a *serial counter*. The analogy to the ripple carry adder suggests that there might be counter logic analogous to the carry lookahead adder. Such logic can be derived by contracting a carry lookahead adder, with the result shown in Figure 6-13(b). This logic can simply replace that in the blue box in Figure 6-13(a) to produce a counter with *parallel gating*, called a *parallel counter*. The advantage of parallel gating logic is that, in going from state 1111 to state 0000, only one AND-gate delay occurs instead of the four AND-gate delays that occur for the serial counter. This reduction in delay allows the counter to operate much faster.

If we connect two 4-bit parallel counters together by connecting the *CO* output of one to the *EN* input of the other, the result is an 8-bit serial-parallel counter. This counter has two 4-bit parallel parts connected in series with each other. The idea can be extended to counters of any length. Again, employing the analogy to carry lookahead adders, additional levels of gating logic can be introduced to replace the serial

connections between the 4-bit segments. The added reduction in delay that results is useful for constructing large, fast counters.

The symbol for the 4-bit counter using positive-edge triggering is shown in Figure 6-13(c).

Up–Down Binary Counter A synchronous count-down binary counter goes through the binary states in reverse order from 1111 to 0000 and back to 1111 to repeat the count. The logic diagram of a synchronous count-down binary counter is similar to the circuit for the binary up-counter, except that a decrementer is used instead of an incrementer. The two operations can be combined to form a counter that can count both up and down, which is referred to as an up–down binary counter. Such a counter can be designed by contracting the adder–subtractor in Figure 3-45 into an incrementer–decrementer and adding the D flip-flops. The counter counts up for $S = 0$ and down for $S = 1$.

Alternatively, an up–down counter with ENABLE can be designed directly from counter behavior. It needs a mode input to select between the two operations. We designate this mode select input by S, with $S = 0$ for up-counting and $S = 1$ for down-counting. Let variable EN be a count enable input, with $EN = 1$ for normal up- or down-counting and $EN = 0$ for disabling both counts. A 4-bit up–down binary counter can be described by the following flip-flop input equations:

$$D_{A0} = Q_0 \oplus EN$$

$$D_{A1} = Q_1 \oplus ((Q_0 \cdot \overline{S} + \overline{Q}_0 \cdot S) \cdot EN)$$

$$D_{A2} = Q_2 \oplus ((Q_0 \cdot Q_1 \cdot \overline{S} + \overline{Q}_0 \cdot \overline{Q}_1 \cdot S) \cdot EN)$$

$$D_{A3} = Q_3 \oplus ((Q_0 \cdot Q_1 \cdot Q_2 \cdot \overline{S} + \overline{Q}_0 \cdot \overline{Q}_1 \cdot \overline{Q}_2 \cdot S) \cdot EN)$$

The logic diagram of the circuit can be easily obtained from the input equations but is not included here. It should be noted that the equations, as written, provide parallel gating using distinct carry logic for up-counting and down-counting. It is also possible to use two distinct serial gating chains. In contrast, the counter derived using the incrementer–decrementer uses only a single carry chain. Overall, the logic cost is similar.

Binary Counter with Parallel Load Counters employed in digital systems quite often require a parallel-load capability for transferring an initial binary number into the counter prior to the count operation. Two inputs control the operation, *Load* and *Count*. These inputs can take on four combinations, but only three operations are provided: *Load* (10), *Count* (01), and *Hold* (00). The effect of the remaining input combination (11) will be considered shortly. The implementation uses an incrementer plus $2n + 1$ ENABLEs, a NOT gate, and n 2-input OR gates as shown in Figure 6-14. The first n ENABLEs with enable input *Load* are used to enable and disable the parallel load of input data, D. The second n ENABLEs with enable input $\overline{Load}$ on the incrementer outputs are used to disable both the count and hold operations when $Load = 1$. When $Load = 0$, both count and hold are enabled. Without the additional ENABLE, $Count = 1$, causes counting, and $Count = 0$, the hold operation

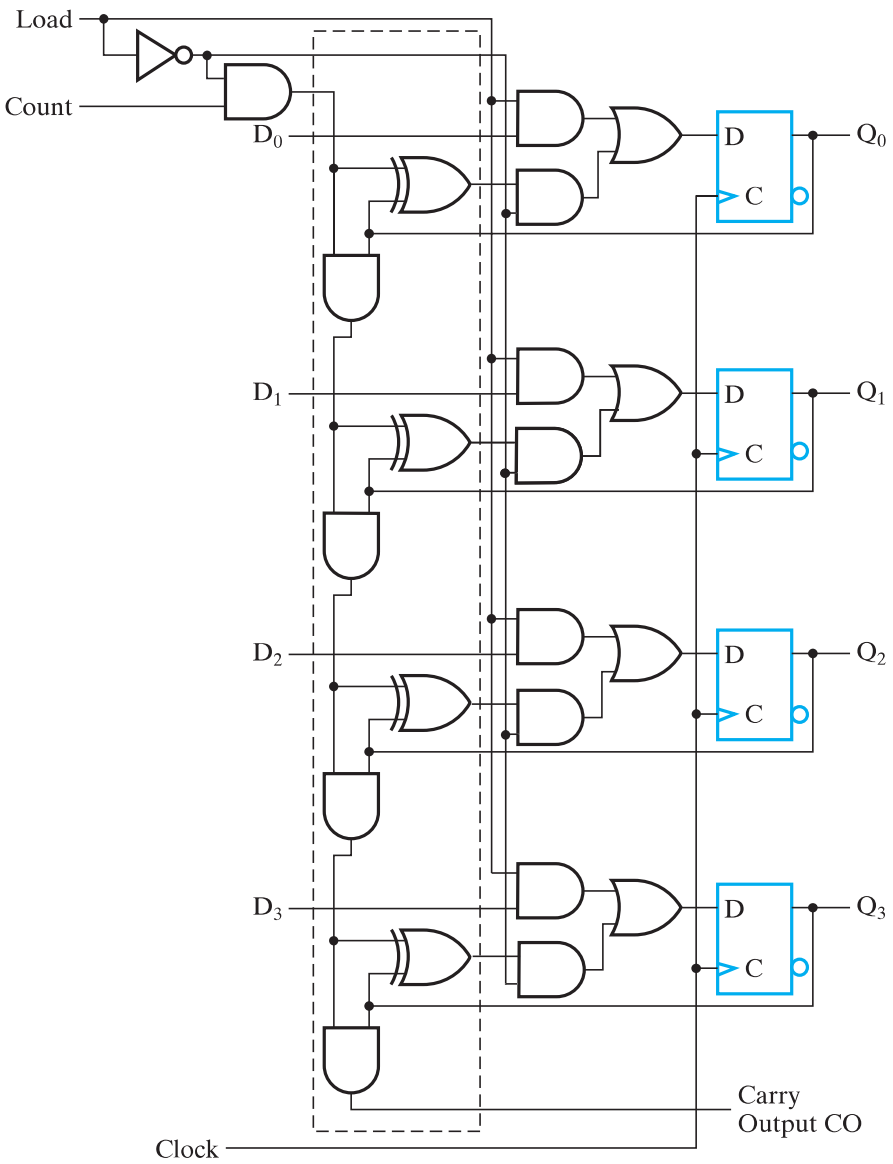


FIGURE 6-14
4-Bit Binary Counter with Parallel Load

occurs. What about the (11) combination? Counting is disabled by the $\overline{Load}$ signal and loading is enabled by *Load*. But what about the output *CO*? With *Count* = 1, the carry chain for the incrementer is active and can produce *CO* equal to 1. But *CO* should not be active outside of the counting operation. To deal with this problem, *Count* is enabled using $\overline{Load}$. With *Load* = 1, then $\overline{Load}$ = 0, which disables *Count* from going into the carry chain and forces *CO* to 0. Thus, for (11), a load occurs. This is sometimes described as *Load* overriding *Count*. When 4-bit counters are concatenated to form 4*n*-bit counters, for the first state, a count control input is attached to *Count* in the least significant stage. For all other stages, *CO* from the prior state is attached to *Count*. Counters with parallel load are very useful in the design of digital computers. In subsequent chapters, we often refer to them as registers with load and increment operations.

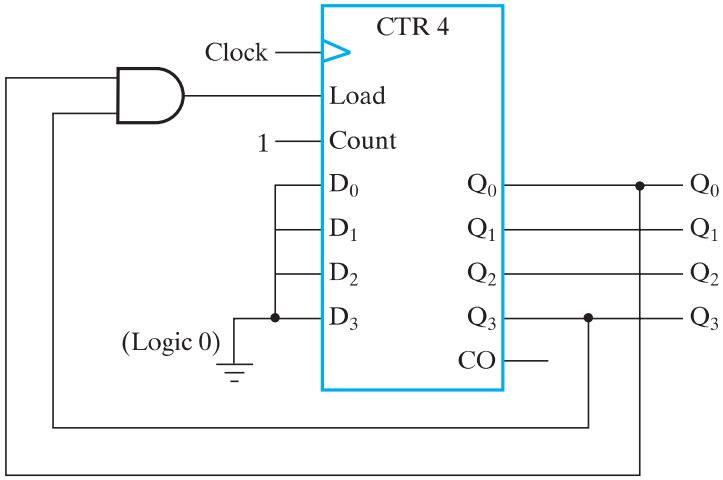


FIGURE 6-15
BCD Counter

The binary counter with parallel load can be converted into a synchronous BCD counter (without load input) by connecting an external AND gate to it, as shown in Figure 6-15. The counter starts with an all-zero output, and the count input is always active. As long as the output of the AND gate is 0, each positive clock edge increments the counter by 1. When the output reaches the count of 1001, both Q_0 and Q_3 become 1, making the output of the AND gate equal to 1. This condition makes *Load* active—so on the next clock transition, the counter does not count, but is loaded from its four inputs. Since all four inputs are connected to logic 0, 0000 is loaded into the counter following the count of 1001. Thus, the circuit counts from 0000 through 1001, followed by 0000, as required for a BCD counter.

Other Counters

Counters can be designed to generate any desired number of states in sequence. A *divide-by-N counter* (also known as a *modulo-N counter*) is a counter that goes through a repeated sequence of *N* states. The sequence may follow the binary count or may be any other arbitrary sequence. In either case, the design of the counter follows the procedure presented in Chapter 4 for the design of synchronous sequential circuits. To demonstrate this procedure, we present the design of two counters: a BCD counter and a counter with an arbitrary sequence of states.

BCD Counter As shown in the previous section, a BCD counter can be obtained from a binary counter with parallel load. It is also possible to design a BCD counter directly using individual flip-flops and gates. Assuming *D*-type flip-flops for the counter, we list the present states and corresponding next states in Table 6-9. An output *Y* is included in the table. This output is equal to 1 when the present state is 1001. In this way, *CO* can enable the count of the next decade while its own decade switches from 1001 to 0000.

The flip-flop input equations for *D* are obtained from the next-state values listed in the table and can be simplified by means of K-maps. The unused states for

TABLE 6-9
State Table and Flip-Flop Inputs for BCD Counter

Present State				Next State				Output
Q_8	Q_4	Q_2	Q_1	$D_8 = Q_8(t+1)$	$D_4 = Q_4(t+1)$	$D_2 = Q_2(t+1)$	$D_1 = Q_1(t+1)$	Y
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	1	0
0	0	1	1	0	1	0	0	0
0	1	0	0	0	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	0	1	0	0	0	0	1

minterms 1010 through 1111 are used as don't-care conditions. The simplified input equations for the BCD counter are

$$D_1 = \overline{Q}_1$$

$$D_2 = Q_2 \oplus Q_1\overline{Q}_8$$

$$D_4 = Q_4 \oplus Q_1Q_2$$

$$D_8 = Q_8 \oplus (Q_1Q_8 + Q_1Q_2Q_4)$$

$$CO = Q_1Q_8$$

Synchronous BCD counters can be cascaded to form counters for decimal numbers of any length. The cascading is done by replacing D_1 with $D_1 = Q_1 \oplus CI$, where CI is an input driven by CO from the next lower BCD counter. Also, CI needs to be ANDed with the product terms to the right of each of the XOR symbols in each of the equations for D_2 through D_8.

Arbitrary Count Sequence Suppose we wish to design a counter that has a repeated sequence of six states, as listed in Table 6-10. In this sequence, flip-flops B and C repeat the binary count 00, 01, 10, while flip-flop A alternates between 0 and 1 every three counts. Thus, the count sequence for the counter is not straight binary, and two states, 011 and 111, are not included in the count. The D flip-flop input equations can be simplified using minterms 3 and 7 as don't-care conditions. The simplified functions are

$$D_A = A \oplus B$$

$$D_B = C$$

$$D_C = \overline{B}\,\overline{C}$$

TABLE 6-10
State Table and Flip-Flop Inputs for Counter

Present State			Next State		
A	B	C	DA = A(t + 1)	DB = B(t + 1)	DC = C(t + 1)
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0

The logic diagram of the counter is shown in Figure 6-16(a). Since there are two unused states, we analyze the circuit to determine their effect. The state diagram obtained is drawn in Figure 6-16(b). This diagram indicates that if the circuit ever goes to one of the unused states, the next count pulse transfers it to one of the valid states, and the circuit then continues to count correctly.

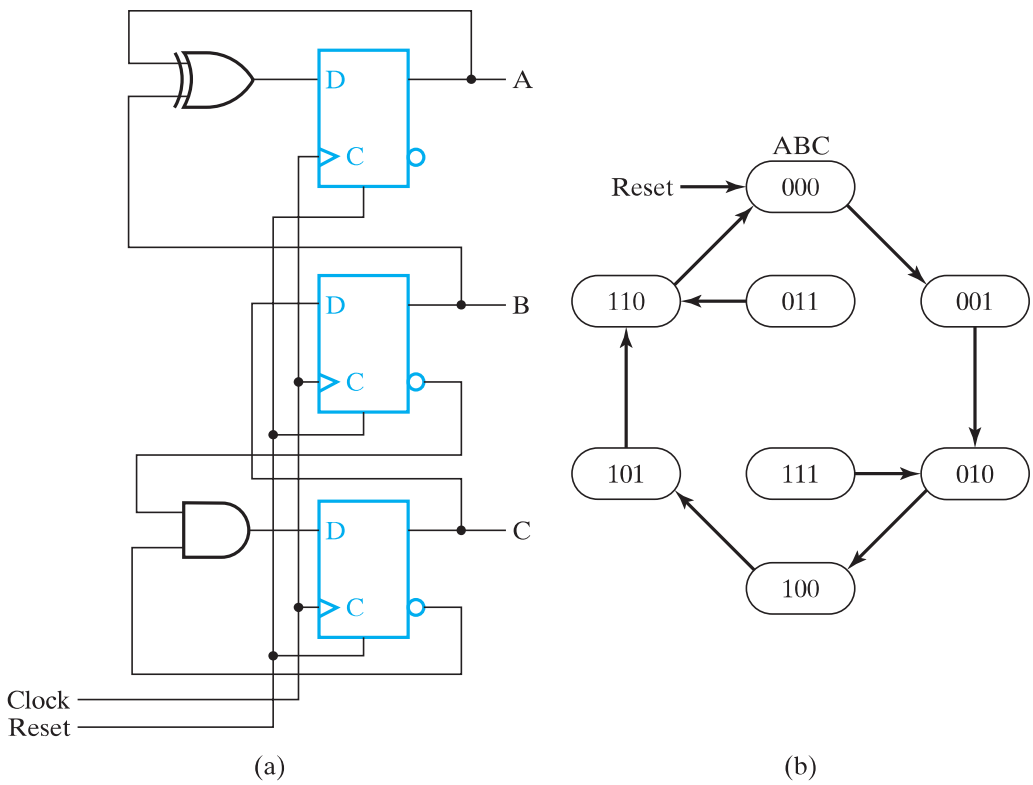


FIGURE 6-16
Counter with Arbitrary Count

6-7 Register-Cell Design

In Section 3-8, we discussed iterative combinational circuits. In this chapter, we connect such circuits to flip-flops to form sequential circuits. A single-bit cell of an iterative combinational circuit, connected to a flip-flop that provides the output, forms a two-state sequential circuit called a *register cell*. We can design an n-bit register with one or more associated microoperations by designing a register cell and making n copies of it. Depending on whether the output of the flip-flop is an input to the iterative circuit cell, the register cell may have its next state dependent on its present state and inputs or on its inputs only. If the dependency is only on inputs, then cell design for the iterative combinational circuit and attachment of the iterative circuit to flip-flops is appropriate. If, however, the state of the flip-flop is fed back to the inputs of the iterative circuit cell, sequential design methods can also be applied. The next example illustrates simple register-cell design in such a case.

EXAMPLE 6-1 Register-Cell Design

A register A is to implement the following register transfers:

AND: $A \leftarrow A \wedge B$
EXOR: $A \leftarrow A \oplus B$
OR: $A \leftarrow A \vee B$

Unless specified otherwise, we assume that

1. Only one of AND, EXOR, and OR is equal to 1, and
2. For all of AND, EXOR, and OR equal to 0, the content of A remains unchanged.

A simple design approach for a register cell with conditions 1 and 2 uses a register with parallel load constructed from D flip-flops with Enable ($EN = LOAD$) from Figure 6-2. For this approach, the expression for $LOAD$ is the OR of all control signals that cause a transfer to occur. The expression for D_i consists of an OR of the AND of each control signal with the operation on the right-hand side of the corresponding transition.

For this example, the resulting equations for LOAD and D_i are

$$\text{LOAD} = \text{AND} + \text{EXOR} + \text{OR}$$
$$D_i = A(t+1)_i = \text{AND} \cdot A_i B_i + \text{EXOR} \cdot (A_i \overline{B}_i + \overline{A}_i B_i) + \text{OR} \cdot (A_i + B_i)$$

The equation for D_i has an implementation similar to that used for the selection part of a multiplexer in which a set of ENABLE blocks drive an OR gate. AND, EXOR, and OR are enabling signals, and the remaining part of the respective terms in D_i consists of the function enabled.

A more complex approach is to design directly for D flip-flops using a sequential circuit design approach rather than the ad hoc approach based on parallel load flip-flops.

We find a coded state table with A as the state variable and output, and AND, $EXOR$, OR, and B as inputs, as shown in Table 6-11. The assumption that at most one

TABLE 6-11
State Table and Flip-Flop Inputs for Example 6-1

Present State A	Next State A(t + 1)						
	(AND = 0) (EXOR = 0) (OR = 0)	(OR = 1) (B = 0)	(OR = 1) (B = 1)	(EXOR = 1) (B = 0)	(EXOR = 1) (B = 1)	(AND = 1) (B = 0)	(AND = 1) (B = 1)
0	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1

of the three control variables AND, $EXOR$, and OR is 1 is instrumental in defining the column headings. From the table, the equation for D_i can be written as:

$$D_i = A(t+1)_i = \text{AND} \cdot A_i \cdot B_i + \text{EXOR} \cdot (A_i\overline{B}_i + \overline{A}_iB_i) + \text{OR} \cdot (A_i + B_i) + \overline{AND} \cdot \overline{EXOR} \cdot \overline{OR} \cdot A_i$$

In attempting to simplify this equation, it is important to note that factors involving only the control variables can be shared between register cells since they are the same for each cell. On the other hand, factors including variables A_i or B_i are implemented in each cell, so the gate-input cost is multiplied by n, the number of cells. In order to easily separate out the factors involving condition variables only, we rewrite D_i in terms of minterms of variables A_i and B_i:

$$\begin{aligned} D_i &= (AND + OR + \overline{AND} \cdot \overline{EXOR} \cdot \overline{OR})(A_iB_i) + (EXOR + OR + \overline{AND} \cdot \overline{EXOR} \cdot \overline{OR})(A_i\overline{B}_i) + (\text{EXOR} + \text{OR})(\overline{A}_iB_i) \\ &= (AND + OR + \overline{EXOR})(A_iB_i) + (EXOR + OR + \overline{AND})(A_i\overline{B}_i) + (\text{EXOR} + \text{OR})(\overline{A}_iB_i) \end{aligned}$$

The terms $OR + AND + \overline{EXOR}$, $EXOR + OR$, and $(EXOR + OR) + \overline{AND}$ do not depend on the values A_i and B_i associated with any of the cells. The logic for these terms can be shared by all of the register cells. Using C_1, C_2, and C_3 as intermediate variables, the following set of equations results:

$$\begin{aligned} C_1 &= OR + AND + \overline{EXOR} \\ C_2 &= OR + EXOR \\ C_3 &= C_2 + \overline{AND} \\ D_i &= C_1A_iB_i + C_3A_i\overline{B}_i + C_2\overline{A}_iB_i \end{aligned}$$

The logic shared by all of the cells and the logic for register cell i are given in Figure 6-17. Before comparing these results with those from the simple approach, we can apply similar simplification and logic sharing to the results of the simple approach:

$$\begin{aligned} C_1 &= \text{OR} + \text{AND} \\ C_2 &= \text{OR} + \text{EXOR} \end{aligned}$$

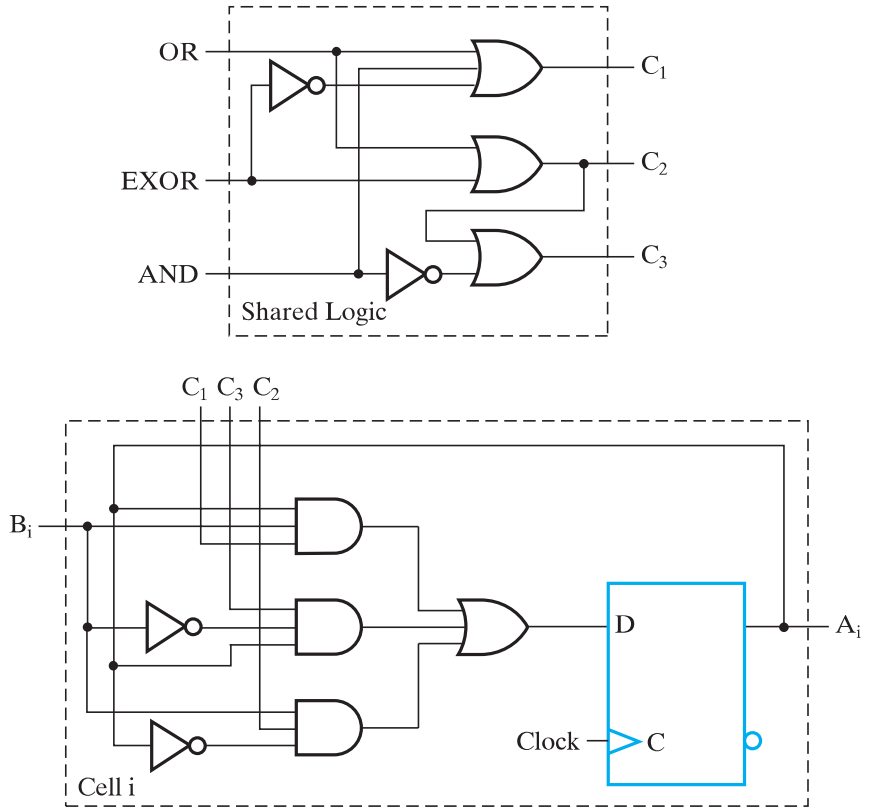


FIGURE 6-17
Logic Diagram—Register-Cell Design Example 6-1

$$D_i = C_1A_iB_i + C_2A_i\overline{B}_i + C_2\overline{A}_iB_i$$
$$LOAD = C_1 + C_2$$
$$D_{i,FF} = \text{LOAD} \cdot D_i + \overline{LOAD} \cdot A_i$$

If these equations are used directly the cost of the simple approach for a 16-cell design is about 40% higher. So by designing a custom register cell using a *D* flip-flop rather than finding input logic for a *D* flip-flop with enable, the cost can be reduced. Further, with the decrease in the number of levels of logic, the delay may also be reduced. ■

In the preceding example, there are no lateral connections between adjacent cells. Among the operations requiring lateral connections are shifts, arithmetic operations, and comparisons. One approach to the design of these structures is to combine combinational designs given in Chapter 3 with selection logic and flip-flops. A generic approach for multifunctional registers using flip-flops with parallel load is shown in Figure 6-8. This simple approach bypasses register-cell design but, if directly implemented, can result in excessive logic and too many lateral connections. The alternative is to do a custom register-cell design. In such designs, a critical factor is the definition of the lateral connection(s) needed. Also, different operations can be defined by controlling input to the least significant cell of the cell

cascade. The custom design approach is illustrated in the next example by the design of a multifunctional register cell.

EXAMPLE 6-2 Register-Cell Design

A register A is to implement the following register transfers:

SHL: $A \leftarrow \text{sl } A$
EXOR: $A \leftarrow A \oplus B$
ADD: $A \leftarrow A + B$

Unless specified otherwise, we assume that

1. Only one of SHL, EXOR, and ADD is equal to 1, and
2. For all of SHL, EXOR, and ADD equal to 0, the content of A remains unchanged.

A simple approach to designing a register cell with conditions 1 and 2 is to use a parallel load with enable EN equal to LOAD. For this approach, the expression for LOAD is the OR of all control signals that cause a transfer to occur. The implementation for D_i consists of an AND-OR, with each AND having a control signal and the logic for the operation on the right-hand side as its inputs.

For this example, the resulting equations for LOAD and D_i are

$$\text{LOAD} = \text{SHL} + \text{EXOR} + \text{ADD}$$
$$D_i = A(t+1)_i = \text{SHL} \cdot A_{i-1} + \text{EXOR} \cdot (A_i \oplus B_i) + \text{ADD} \cdot ((A_i \oplus B_i) \oplus C_i)$$
$$C_{i+1} = (A_i \oplus B_i)C_i + A_iB_i$$

These equations can be used without modification or can be optimized.

Now, suppose, that we do a custom design assuming that all of the register cells are identical. This means that the least and most significant cells will be the same as those internal to the cell chain. Because of this, the value of C_0 must be specified and the use, if any, of C_n must be determined for each of the three operations. For the left shift, a zero fill of the vacated rightmost bit is assumed, giving $C_0 = 0$. Since C_0 is not involved in the EXOR operation, it can be assumed to be a don't-care. Finally, for the addition, C_0 either can be assumed to be 0 or can be left as a variable to permit a carry from a previous addition to be injected. We assume that C_0 equals 0 for addition, since no additional carry-in is specified by the register transfer statement.

Our first formulation goal is to minimize lateral connections between cells. Two of the three operations, left shift and addition, require a lateral connection to the left (i.e., toward the most significant end of the cell chain). Our goal is to use one signal for both operations, say, C_i. It already exists for the addition but must be redefined to handle both the addition and the left shift. Also in our custom design, the parallel load flip-flop will be replaced by a D flip-flop. We can now formulate the state table for the register cell shown in Table 6-12:

$$D_i = A(t+1)_i = \overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}} \cdot A_i + \text{SHL} \cdot C_i + \text{EXOR} \cdot (A_i \oplus B_i) + \text{ADD} \cdot (A_i \oplus B_i \oplus C_i)$$

$$C_{i+1} = \text{SHL} \cdot A_i + \text{ADD} \cdot ((A_i \oplus B_i)C_i + A_iB_i)$$

TABLE 6-12
State Table and Flip-Flop Inputs for Register-Cell Design in Example 6-2

Present State A_i	Inputs	Next State $A_i(t+1)$/Output C_{i+1}											
	SHL = 0	SHL = 1	1	1	1	EXOR = 1	1	ADD = 1	1	1	1
	EXOR = 0	B_i = 0	0	1	1	B_i = 0	1	B_i = 0	0	1	1
	ADD = 0	C_i = 0	1	0	1			C_i = 0	1	0	1
0	0/X	0/0	1/0	0/0	1/0	0/X	1/X	0/0	1/0	1/0	0/1
1	1/X	0/1	1/1	0/1	1/1	1/X	0/X	1/0	0/1	0/1	1/1

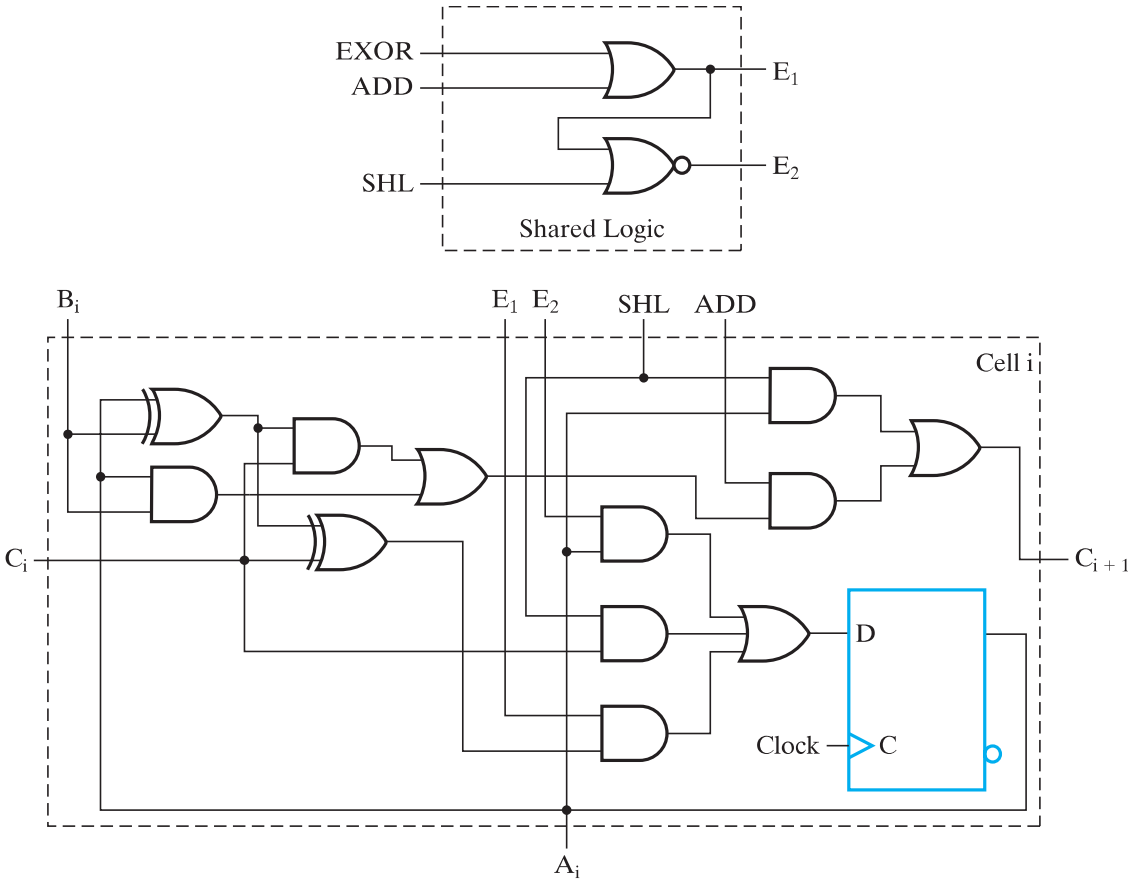


FIGURE 6-18
Logic Diagram—Register-Cell Design Example 6-2

The term $A_i \oplus B_i$ appears in both the EXOR and ADD terms. In fact, if $C_i = 0$ during the EXOR operation, then the functions for the sum in ADD and for EXOR can be identical. In the C_{i+1} equation, since SHL and ADD are both 0 when EXOR is 1, C_i is 0 for all cells in the cascade except the least significant one. For the least significant cell, the specification states that $C_0 = 0$. Thus, input values C_i are 0 for all cells in register A. So we can combine the ADD and EXOR operations as follows:

$$D_i = A(t+1)_i = \overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}} \cdot A_i + \text{SHL} \cdot C_i + (\text{EXOR} + \text{ADD}) \cdot ((A_i \oplus B_i) \oplus C_i)$$

The expressions $\overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}}$ and EXOR + ADD, which are independent of A_i, B_i, and C_i, can be shared by all cells. The resulting equations are

$$E_1 = \text{EXOR} + \text{ADD}$$

$$E_2 = \overline{E1 + \text{SHL}}$$

$$D_i = E_2 \cdot A_i + \text{SHL} \cdot C_i + E_1 \cdot ((A_i \oplus B_i) \oplus C_i)$$

$$C_{i+1} = \text{SHL} \cdot A_i + \text{ADD} \cdot ((A_i \oplus B_i)C_i + A_iB_i)$$

The resulting register cell appears in Figure 6-18. Comparing this result with the register cell for the simple design, we note the following two differences:

1. Only one lateral connection between cells exists instead of two.
2. Logic has been very efficiently shared by the addition and the EXOR operation.

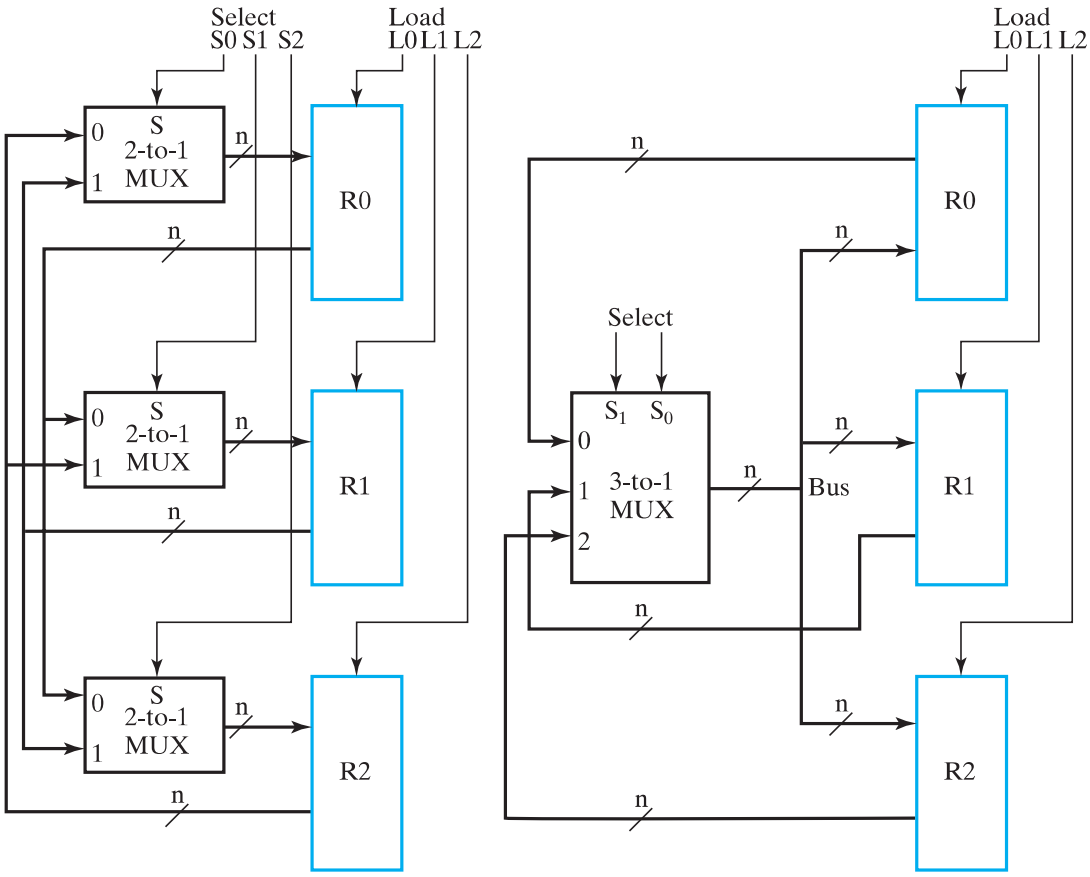
The custom cell design has produced connection and logic savings not present in the block-level design with or without optimization. ■

6-8 Multiplexer and Bus-Based Transfers for Multiple Registers

A typical digital system has many registers. Paths must be provided to transfer data from one register to another. The amount of logic and the number of interconnections may be excessive if each register has its own dedicated set of multiplexers. A more efficient scheme for transferring data between registers is a system that uses a shared transfer path called a *bus*. A bus is characterized by a set of common lines, with each line driven by selection logic. Control signals for the logic select a single source and one or more destinations on any clock cycle for which a transfer occurs.

In Section 6-4, we saw that multiplexers and parallel load registers can be used to implement dedicated transfers from multiple sources. A block diagram for such transfers between three registers is shown in Figure 6-19(a). There are three n-bit 2-to-1 multiplexers, each with its own select signal. Each register has its own load signal. The same system based on a bus can be implemented by using a single n-bit 3-to-1 multiplexer and parallel load registers. If a set of multiplexer outputs is shared as a common path, these output lines are a bus. Such a system with a single bus for transfers between three registers is shown in Figure 6-19(b). The control input pair, *Select*, determines the contents of the single source register that will appear on the multiplexer outputs (i.e., on the bus). The *Load* inputs determine the destination register or registers to be loaded with the bus data.

In Table 6-13, transfers using the single-bus implementation of Figure 6-19(b) are illustrated. The first transfer is from $R2$ to $R0$. *Select* equals 10, selecting input $R2$ to the multiplexer. *Load* signal $L0$ for register $R0$ is 1, with all other loads at 0, causing the contents of $R2$ on the bus to be loaded into $R0$ on the next positive clock transition. The second transfer in the table illustrates the loading of the contents of $R1$ into both $R0$ and $R2$. The source $R1$ is selected because *Select* is equal to 01. In



(a) Dedicated multiplexers

(b) Single bus

□ **FIGURE 6-19**
Single Bus versus Dedicated Multiplexers

this case, $L2$ and $L0$ are both 1, causing the contents of $R1$ on the bus to be loaded into registers $R0$ and $R2$. The third transfer, an exchange between $R0$ and $R1$, is impossible in a single clock cycle, since it requires two simultaneous sources, $R0$ and $R1$, on the single bus. Thus, this transfer requires at least two buses or a bus combined with a dedicated path from one of the registers to the other. Note that such a transfer can be executed on the dedicated multiplexers in Figure 6-19(a). So, for a single-bus system, simultaneous transfers with different sources in a single clock cycle are

□ **TABLE 6-13**
Examples of Register Transfers Using the Single Bus in Figure 6-19(b)

	Select		Load		
Register Transfer	**S1**	**S0**	**L2**	**L1**	**L0**
$R0 \leftarrow R2$	1	0	0	0	1
$R0 \leftarrow R1$, $R2 \leftarrow R1$	0	1	1	0	1
$R0 \leftarrow R1$, $R1 \leftarrow R0$			Impossible		

impossible, whereas for the dedicated multiplexers, any combination of transfers is possible. Hence, the reduction in hardware that occurs for a single bus in place of dedicated multiplexers results in limitations on simultaneous transfers.

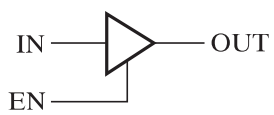
If we assume that only single-source transfers are needed, then we can use Figure 6-19 to compare the complexity of the hardware in dedicated versus bus-based systems. First of all, assume a multiplexer design, as in Figure 3-27. In Figure 6-19(a), there are $2n$ AND gates and n OR gates per multiplexer (not counting inverters), for a total of $9n$ gates. In contrast, in Figure 6-19(b), the bus multiplexer requires only $3n$ AND gates and n OR gates, for a total of $4n$ gates. Also, the data input connections to the multiplexers are reduced from $6n$ to $3n$. Thus, the cost of the selection hardware is reduced by about half.

High-Impedance Outputs

Another method for constructing a bus involves a type of gate called a *three-state buffer*. Thus far, we have considered gates that have only output values logic 0 and logic 1. In this section, we introduce an important structure, the three-state buffer, that provides a third output value referred to as the *high-impedance state* and denoted by Hi-Z or just plain Z or z. The Hi-Z value behaves as an open circuit, which means that, looking back into the circuit, we find that the output appears to be disconnected internally. Thus, the output appears not to be there at all and, thus, is incapable of driving any attached inputs. Gates with Hi-Z output capability have two very useful properties. First of all, Hi-Z outputs can be connected together, provided that no two or more gates drive the line at the same time to opposite 0 and 1 values. In contrast, gates with only logic 0 and logic 1 outputs cannot have their outputs connected together. Second, an output in the Hi-Z states, since it appears as an open circuit, can have an input attached to it internally, so that the Hi-Z output can act as both an output and an input. This is referred to as a bidirectional input/output. Instead of carrying signals in just one direction, interconnections between Hi-Z outputs can carry information in both directions. This feature reduces significantly the number of interconnections required.

High-impedance outputs may appear on any gate, but here we restrict consideration to a primitive gate structure with a single data input, a three-state buffer. As the name implies, a three-state logic output exhibits three distinct states. Two of the "states" are the logic 1 and logic 0 of conventional logic. The third "state" is the Hi-Z value, which, for three-state logic, is referred to as the Hi-Z *state*.

The graphic symbol and truth table for a 3-state buffer are given in Figure 6-20(a). The symbol in Figure 6-20(a) is distinguished from the symbol for a normal buffer by the enable input, *EN*, entering the bottom of the buffer symbol. From the truth table in

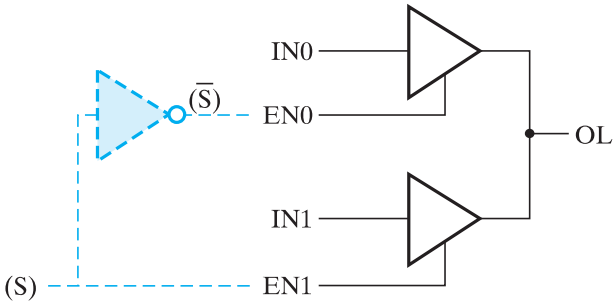


(a) Logic symbol

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

(b) Truth table

□ **FIGURE 6-20**
Three-State Buffer



(a) Logic Diagram

EN1	EN0	IN1	IN0	OL
0	0	X	X	Hi-Z
(S) 0	$(\overline{S})$ 1	X	0	0
0	1	X	1	1
1	0	0	X	0
1	0	1	X	1
1	1	0	0	0
1	1	1	1	1
1	1	0	1	
1	1	1	0	

(b) Truth table

FIGURE 6-21
Three-State Buffers Forming a Multiplexed Line OL

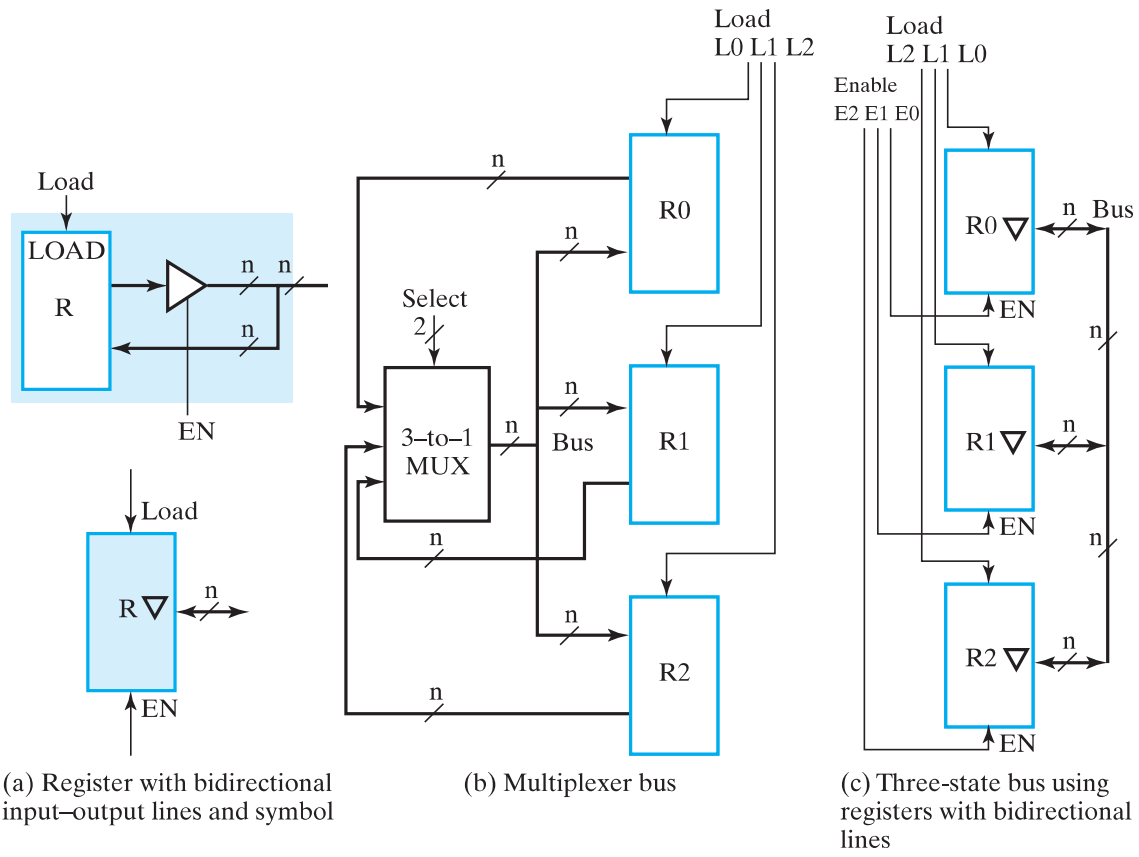
Figure 6-20(b), if $EN = 1$, OUT is equal to IN, behaving like a normal buffer. But for $EN = 0$, the output value is high impedance (Hi-Z), regardless of the value of IN.

Three-state buffer outputs can be connected together to form a multiplexed output line. Figure 6-21(a) shows two 3-state buffers with their outputs connected to form output line OL. We are interested in the output of this structure in terms of the four inputs $EN1$, $EN0$, $IN1$, and $IN0$. The output behavior is given by the truth table in Figure 6-21(b). For $EN1$ and $EN0$ equal to 0, both buffer outputs are Hi-Z. Since both appear as open circuits, OL is also an open circuit, represented by a Hi-Z value. For $EN1 = 0$ and $EN0 = 1$, the output of the top buffer is $IN0$ and the output of bottom buffer is Hi-Z. Since the value of $IN0$ combined with an open circuit is just $IN0$, OL has value $IN0$, giving the second and third rows of the truth table. A corresponding, but opposite, case occurs for $EN1 = 1$ and $EN0 = 0$, so OL has value $IN1$, giving the fourth and fifth rows of the truth table. For $EN1$ and $EN0$ both 1, the situation is more complicated. If $IN1 = IN0$, then their mutual value appears at OL. But if $IN1 \neq IN0$, then their values conflict at the output. The conflict results in an electrical current flowing from the buffer output that is at 1 into the buffer output that is at 0. This current is often large enough to cause heating and may even destroy the circuit, as symbolized by the "smoke" icons in the truth table. Clearly, such a situation must be avoided. The designer must ensure that $EN0$ and $EN1$ never equal 1 at the same time. In the general case, for n 3-state buffers attached to a bus line, EN can equal 1 for only one of the buffers and must be 0 for the rest. One way to ensure this

is to use a decoder to generate the *EN* signals. For the two-buffer case, the decoder is just an inverter with select input *S*, as shown in dotted lines in Figure 6-21(a). It is interesting to examine the truth table with the inverter in place. It consists of the shaded area of the table in Figure 6-21(b). Clearly, the value on *S* selects between inputs *IN*0 and *IN*1. Further, the circuit output *OL* is never in the Hi-Z state.

Three-State Bus

A bus can be constructed with the three-state buffers introduced above instead of multiplexers. This has the potential for additional reductions in the number of connections. But why use three-state buffers instead of a multiplexer, particularly for implementing buses? The reason is that many three-state buffer outputs can be connected together to form a bit line of a bus, and this bus is implemented using only one level of logic gates. On the other hand, in a multiplexer, such a large number of sources means a high fan-in OR, which requires multiple levels of OR gates, introducing more logic and increasing delay. In contrast, three-state buffers provide a practical way to construct fast buses with many sources, so they are often preferred in such cases. More important, however, is the fact that signals can travel in two directions on a three-state bus. Thus, the three-state bus can use the same interconnection to carry signals into and out of a logic circuit. This feature, which is most important when crossing chip boundaries, is illustrated in Figure 6-22(a). The figure



(a) Register with bidirectional input–output lines and symbol

(b) Multiplexer bus

(c) Three-state bus using registers with bidirectional lines

FIGURE 6-22
Three-State Bus versus Multiplexer Bus

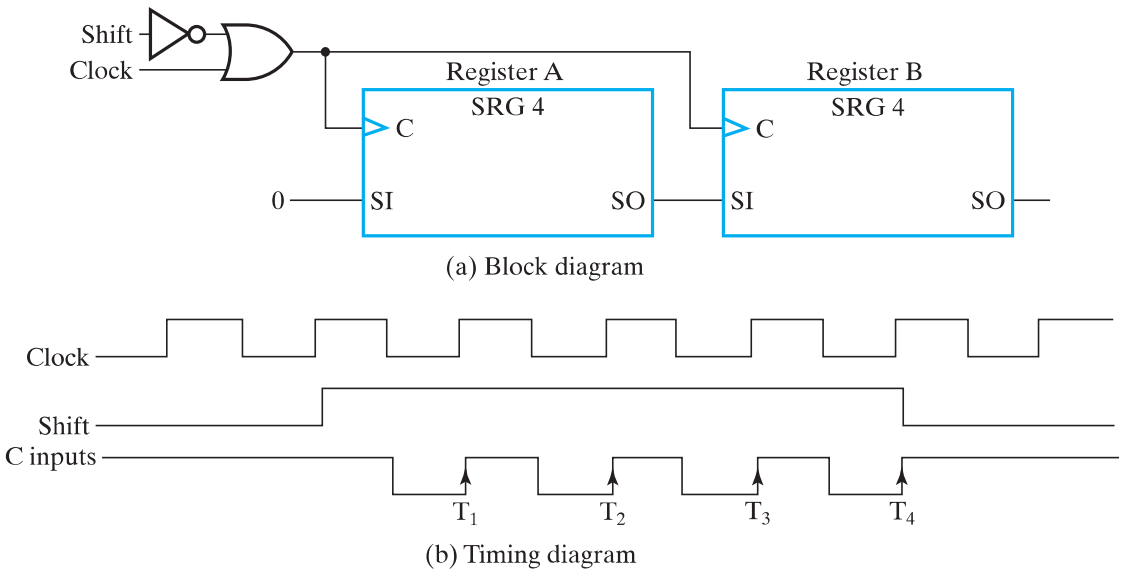
shows a register with n lines that serve as both inputs and outputs lying across the boundary of the shaded area. If the three-state buffers are enabled, then the lines are outputs; if the three-state buffers are disabled, then the lines can be inputs. The symbol for this structure is also given in the figure. Note that the bidirectional bus lines are represented by a two-headed arrow. Also, a small inverted triangle denotes the three-state outputs of the register.

Figures 6-22(b) and (c) show a multiplexer-implemented bus and a three-state bus, respectively, for comparison. The symbol from Figure 6-22(a) for a register with bidirectional input–output lines is used in Figure 6-22(c). In contrast to the situation in Figure 6-19, where dedicated multiplexers were replaced by a bus, these two implementations are identical in terms of their register–transfer capability. Note that, in the three-state bus, there are only three data connections to the set of register blocks for each bit of the bus. The multiplexer-implemented bus has six data connections per bit to the set of register blocks. This reduction in the number of data connections by half, along with the ability to easily construct a bus with many sources, makes the three-state bus an attractive alternative. The use of such bidirectional input–output lines is particularly effective between logic circuits in different physical packages.

6-9 Serial Transfer and Microoperations

A digital system is said to operate in a serial mode when information in the system is transferred or manipulated one bit at a time. Information is transferred one bit at a time by shifting the bits out of one register and into a second register. This transfer method is in contrast to parallel transfer, in which all the bits of the register are transferred at the same time.

The serial transfer of information from register A to register B is done with shift registers, as shown in the block diagram of Figure 6-23(a). The serial output of register A is connected to the serial input of register B. The serial input of register A receives 0s while its data is transferred to register B. It is also possible for register A to receive



□ **FIGURE 6-23**
Serial Transfer

other binary information, or if we want to maintain the data in register A, we can connect its serial output to its serial input so that the information is circulated back into the register. The initial content of register B is shifted out through its serial output and is lost unless it is transferred back into register A, to a third shift register, or to other storage. The shift control input *Shift* determines when and how many times the registers are shifted. The registers using *Shift* are controlled by means of the logic from Figure 6-23(a), which allows the clock pulses to pass to the shift register clock inputs only when *Shift* has the value logic 1.

In Figure 6-23, each shift register has four stages. The logic that supervises the transfer must be designed to enable the shift registers, through the *Shift* signal, for a fixed time of four clock pulses. Shift register enabling is shown in the timing diagram for the clock gating logic in Figure 6-23(b). Four pulses find *Shift* in the active state, so that the output of the logic connected to the clock inputs of the registers produces four pulses: T_1, T_2, T_3, and T_4. Each positive transition of these pulses causes a shift in both registers. After the fourth pulse, *Shift* changes back to 0 and the shift registers are disabled. We note again that, for positive-edge triggering, the pulses on the clock inputs are 0, and the inactive level when no pulses are present is a 1 rather than a 0.

Now suppose that the binary content of register A before the shift is 1011, that of register B is 0010, and the *SI* of register A is logic 0. Then the serial transfer from A to B occurs in four steps, as shown in Table 6-14. With the first pulse T_1, the rightmost bit of A is shifted into the leftmost bit of B, the leftmost bit of A receives a 0 from the serial input, and at the same time, all other bits of A and B are shifted one position to the right. The next three pulses perform identical operations, shifting the bits of A into B one at a time while transferring 0s to A. After the fourth shift, the logic supervising the transfer changes the *Shift* signal to 0 and the shifts stop. Register B contains 1011, which is the previous value of A. Register A contains all 0s.

The difference between serial and parallel modes of operation should be apparent from this example. In the parallel mode, information is available from all bits of a register, and all bits can be transferred simultaneously during one clock pulse. In the serial mode, the registers have a single serial input and a single serial output, and information is transferred one bit at a time.

Serial Addition

Operations in digital computers are usually done in parallel because of the faster speed attainable. Serial operations are slower, but have the advantage of requiring

TABLE 6-14
Example of Serial Transfer

Timing Pulse	Shift Register A				Shift Register B			
Initial value	1	0	1	1	0	0	1	0
After T_1	0	1	0	1	1	0	0	1
After T_2	0	0	1	0	1	1	0	0
After T_3	0	0	0	1	0	1	1	0
After T_4	0	0	0	0	1	0	1	1

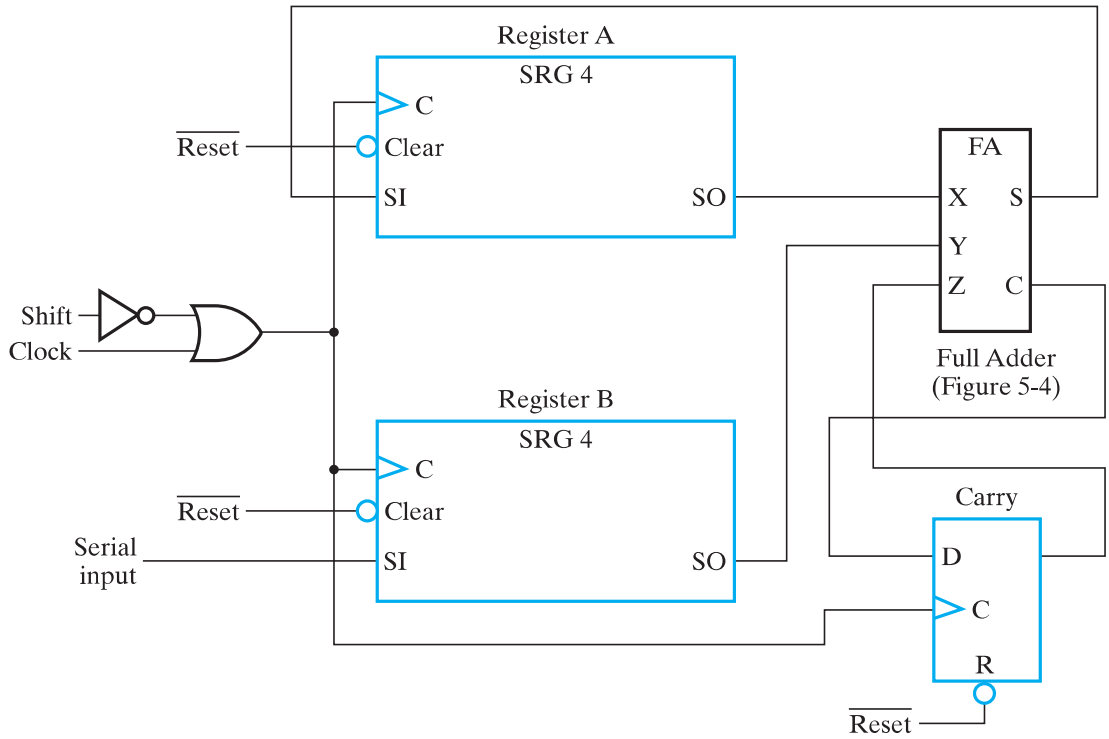


FIGURE 6-24
Serial Addition

less hardware. To demonstrate the serial mode of operation, we will show the operation of a serial adder. Also, we compare the serial adder to the parallel counterpart presented in Section 3-9 to illustrate the time–space trade-off in design.

The two binary numbers to be added serially are stored in two shift registers. Bits are added, one pair at a time, through a single full-adder (FA) circuit, as shown in Figure 6-24. The carry out of the full adder is transferred into a *D* flip-flop. The output of this carry flip-flop is then used as the carry input for the next pair of significant bits. The sum bit on the *S* output of the full adder could be transferred into a third shift register, but we have chosen to transfer the sum bits into register *A* as the contents of the register are shifted out. The serial input of register *B* can receive a new binary number as its contents are shifted out during the addition.

The operation of the serial adder is as follows: Register *A* holds the augend, register *B* holds the addend, and the carry flip-flop has been reset to 0. The serial outputs of *A* and *B* provide a pair of significant bits for the full adder at *X* and *Y*. The output of the carry flip-flop provides the carry input at *Z*. When *Shift* is set to 1, the OR gate enables the clock for both registers and the flip-flop. Each clock pulse shifts both registers once to the right, transfers the sum bit from *S* into the leftmost flip-flop of *A*, and transfers the carry output into the carry flip-flop. Shift control logic enables the registers for as many clock pulses as there are bits in the registers (four pulses in this example). For each pulse, a new sum bit is transferred to *A*, a new carry is transferred to the flip-flop, and both registers are shifted once to the right. This process continues until the shift control logic changes *Shift* to 0. Thus, the addition is accomplished by passing each pair of bits and the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, back into register *A*.

Initially, we can reset register A, register B, and the *Carry* flip-flop to 0. Then we shift the first number into B. Next, the first number from B is added to the 0 in A. While B is being shifted through the full adder, we can transfer a second number to it through its serial input. The second number can be added to the contents of register A at the same time that a third number is transferred serially into register B. Serial addition may be repeated to form the addition of two, three, or more numbers, with their sum accumulated in register A.

A comparison of the serial adder with the parallel adder described in Section 3-9 provides an example of space–time trade-off. The parallel adder has n full adders for n-bit operands, whereas the serial adder requires only one full adder. Excluding the registers from both, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit because it includes the carry flip-flop. The serial circuit also takes n clock cycles to complete an addition. Identical circuits, such as the n full adders in the parallel adder, connected together in a chain constitute an example of an *iterative logic array*. If the values on the carries between the full adders are regarded as state variables, then the states from the least significant end to the most significant end are the same as the states appearing in sequence on the flip-flop output in the serial adder. Note that in the iterative logic array the states appear in space, but in the sequential circuit the states appear in time. By converting from one of these implementations to the other, one can make a space–time trade-off. The parallel adder in space is n times larger than the serial adder (ignoring the area of the carry flip-flop), but it is n times faster. The serial adder, although it is n times slower, is n times smaller in space. This gives the designer a significant choice in emphasizing speed or area, where more area translates into more cost.

6-10 Control of Register Transfers

In Section 6-2, we divided a digital system into two major components, a datapath and a control unit. Likewise, the binary information stored in a digital computer can be classified as either data or control information. As we saw earlier in this chapter, data is manipulated in a datapath by using microoperations implemented with register transfers. These operations are implemented with adder–subtractors, shifters, registers, multiplexers, and buses. The control unit provides signals that activate the various microoperations within the datapath to perform the specified processing tasks. The control unit also determines the sequence in which the various actions are performed. This separation of a system into two components and separation of the tasks performed carries over to the design process. The datapath and control unit are usually designed separately, but in close coordination with each other.

Generally, the timing of all registers in a synchronous digital system is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including those in the control unit. To prevent clock pulses from changing the state of all registers on every clock cycle, some registers have a load control signal that enables and disables the loading of new data into the register. The binary variables that control the selection inputs of multiplexers, buses, and processing logic and the load control inputs of registers are generated by the control unit.

The control unit that generates the signals for sequencing the microoperations is a sequential circuit with states that dictate the control signals for the system. At