Registers, Counters, and the Memory Unit

7-1 INTRODUCTION

A clocked sequential circuit consists of a group of flip-flops and combinational gates connected to form a feedback path. The flip-flops are essential because, in their absence, the circuit reduces to a purely combinational circuit (provided there is no feedback path). A circuit with only flip-flops is considered a sequential circuit even in the absence of combinational gates. Certain MSI circuits that include flip-flops are classified by the operation that they perform rather than the name sequential circuit. Two such MSI components are registers and counters.

A register is a group of binary cells suitable for holding binary information. A group of flip-flops constitutes a register, since each flip-flop is a binary cell capable of storing one bit of information. An n-bit register has a group of n flip-flops and is capable of storing any binary information containing n bits. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops and gates that affect their transition. The flip-flops hold binary information and the gates control when and how new information is transferred into the register.

Counters were introduced in Section 6-8. A counter is essentially a register that goes through a predetermined sequence of states upon the application of input pulses. The gates in a counter are connected in such a way as to produce a prescribed sequence of binary states in the register. Although counters are a special type of register, it is common to differentiate them by giving them a special name.

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. A random-access memory (RAM) differs from a read-only memory (ROM) in that a RAM can transfer the stored information out (read) and is also capable of receiving new information in for storage (write). A more appropriate name for such a memory would be *read–write memory*.

Registers, counters, and memories are extensively used in the design of digital systems in general and digital computers in particular. Registers can also be used to facilitate the design of sequential circuits. Counters are useful for generating timing variables to sequence and control the operations in a digital system. Memories are essential for storage of programs and data in a digital computer. Knowledge of the operation of these components is indispensable for the understanding of the organization and design of digital systems.

7-2 REGISTERS

Various types of registers are available in MSI circuits. The simplest possible register is one that consists of only flip-flops without any external gates. Figure 7-1 shows such a register constructed with four D-type flip-flops and a common clock-pulse input. The clock pulse input, CP, enables all flip-flops, so that the information presently available at the four inputs can be transferred into the 4-bit register. The four outputs can be sampled to obtain the information presently stored in the register.

The way that the flip-flops in a register are triggered is of primary importance. If the flip-flops are constructed with gated D-type latches, as in Fig. 6-5, then information present at a data (D) input is transferred to the Q output when the enable (CP) is 1, and the Q output follows the input data as long as the CP signal remains 1. When CP goes to 0, the information that was present at the data input just before the transition is retained at the Q output. In other words, the flip-flops are sensitive to the pulse duration, and the register is enabled for as long as $CP = 1$. A register that responds to the pulse duration is commonly called a *gated latch*, and the CP input is frequently labeled with the variable G (instead of CP). Latches are suitable for use as temporary storage of binary information that is to be transferred to an external destination.

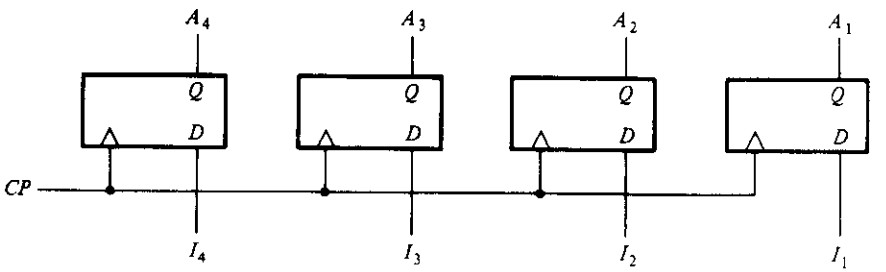


FIGURE 7-1
4-bit register

As explained in Section 6-3, a flip-flop can be used in the design of clocked sequential circuits provided that its clock input responds to the pulse transition rather than the pulse duration. This means that the flip-flops in the register must be of the edge-triggered or master–slave type. A group of flip-flops sensitive to pulse duration is usually called a latch, whereas a group of flip-flops sensitive to pulse transition is called a register. In subsequent discussions, we will assume that any group of flip-flops drawn constitutes a register and that all flip-flops are of the edge-triggered or master–slave type.

Register with Parallel Load

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 single clock pulse, we say that the loading is done in parallel. A pulse applied to the *CP* input of the register of Fig. 7-1 will load all four inputs in parallel. In this configuration, the clock pulse must be inhibited from the *CP* terminal if the content of the register must be left unchanged. In other words, the *CP* input acts as an enable signal that controls the loading of new information into the register. When *CP* goes to 1, the input information is loaded into the register. If *CP* remains at 0, the content of the register is not changed. Note that the change of state in the outputs occurs at the positive edge of the pulse. If a flip-flop changes state at the negative edge, there will be a small circle under the triangle symbol in the *CP* input of the flip-flop.

Most digital systems have a master-clock generator that supplies a continuous train of clock pulses. All clock pulses are applied to all flip-flops and registers in the system. The master-clock generator acts like a pump that supplies a constant beat to all parts of the system. A separate control signal then decides what specific clock pulses will have an effect on a particular register. In such a system, the clock pulses must be ANDed with the control signal, and the output of the AND gate is then applied to the *CP* terminal of the register shown in Fig. 7-1. When the control signal is 0, the output of the AND gate is 0, and the stored information in the register remains unchanged. Only when the control signal is a 1 does the clock pulse pass through the AND gate and into the *CP* terminal for new information to be loaded into the register. Such a control variable is called a *load* control input.

Inserting an AND gate in the path of clock pulses means that logic is performed with clock pulses. The insertion of logic gates produces propagation delays between the master-clock generator and the clock inputs of flip-flops. To fully synchronize the system, we must ensure that all clock pulses arrive at the same time to all inputs of all flip-flops so that they can all change simultaneously. Performing logic with clock pulses inserts variable delays and may throw the system out of synchronism. For this reason, it is advisable (but not necessary, as long as the delays are taken into consideration) to apply clock pulses directly to all flip-flops and control the operation of the register with other inputs, such as the *R* and *S* inputs of an *RS* flip-flop.

A 4-bit register with a load control input using *RS* flip-flops is shown in Fig. 7-2. The *CP* input of the register receives continuous synchronized pulses, which are applied

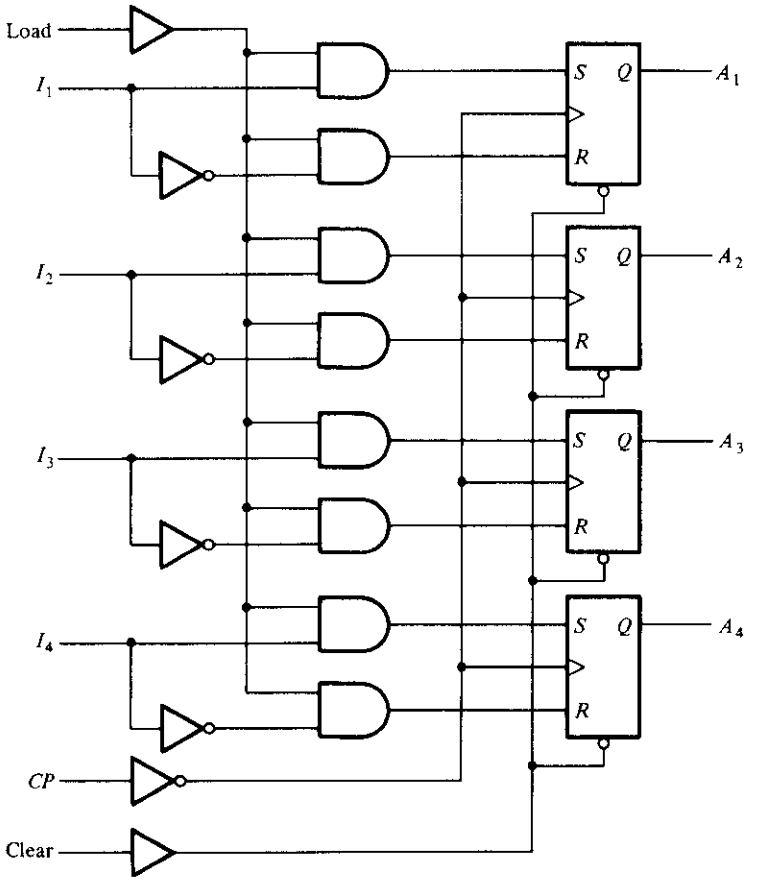


FIGURE 7-2
4-bit register with parallel load

to all flip-flops. The inverter in the *CP* path causes all flip-flops to be triggered by the negative edge of the incoming pulses. The purpose of the inverter is to reduce the loading of the master-clock generator. This is because the *CP* input is connected to only one gate (the inverter) instead of the four gate inputs that would have been required if the connections were made directly into the flip-flop clock inputs (marked with small triangles).

The *clear* input goes to a special terminal in each flip-flop through a noninverting buffer gate. When this terminal goes to 0, the flip-flop is cleared asynchronously. The clear input is useful for clearing the register to all 0's prior to its clocked operation. The clear input must be maintained at 1 during normal clocked operations (see Fig. 6-15).

The *load* input goes through a buffer gate (to reduce loading) and through a series of AND gates to the *R* and *S* inputs of each flip-flop. Although clock pulses are continu-

ously present, it is the load input that controls the operation of the register. The two AND gates and the inverter associated with each input I determine the values of R and S. If the load input is 0, both R and S are 0, and no change of state occurs with any clock pulse. Thus, the load input is a control variable that can prevent any information change in the register as long as its input is 0. When the load control goes to 1, inputs I_1 through I_4 specify what binary information is loaded into the register on the next clock pulse. For each I that is equal to 1, the corresponding flip-flop inputs are $S = 1$, $R = 0$. For each I that is equal to 0, the corresponding flip-flop inputs are $S = 0$, $R = 1$. Thus, the input value is transferred into the register provided the load input is

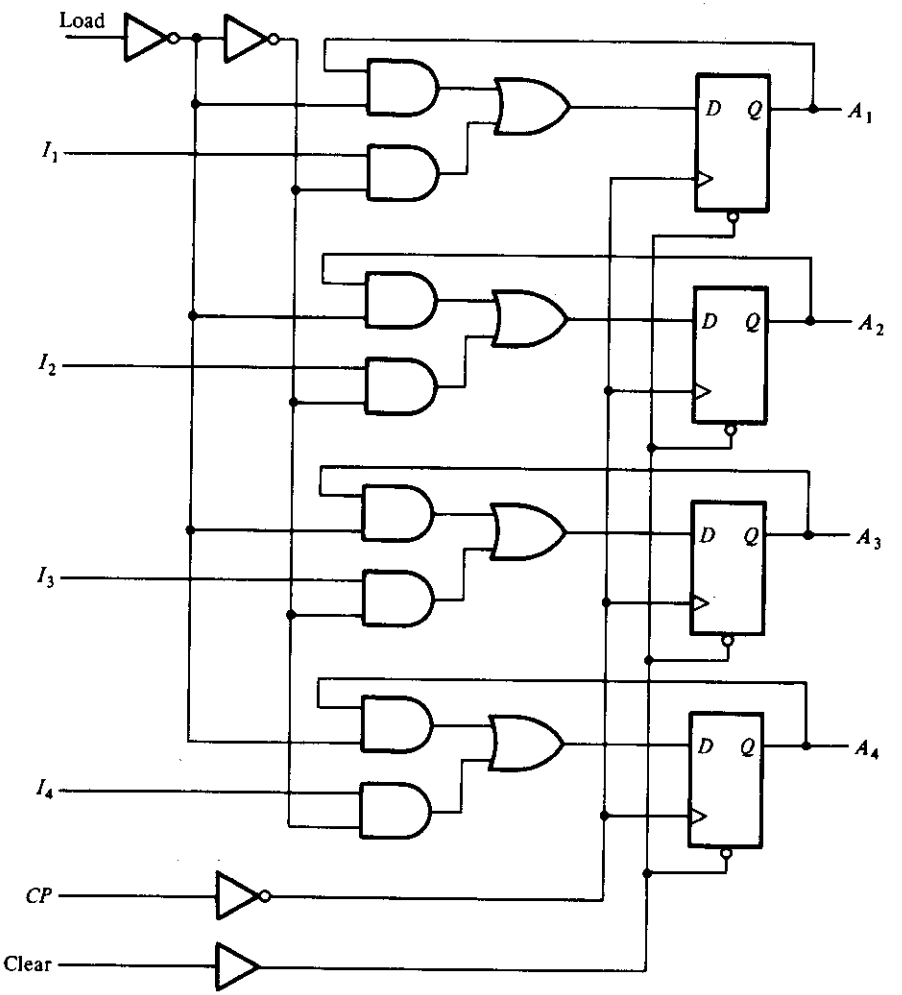


FIGURE 7-3
Register with parallel load using D flip-flops

1, the clear input is 1, and a clock pulse goes from 1 to 0. This type of transfer is called a *parallel-load* transfer because all bits of the register are loaded simultaneously. If the buffer gate associated with the load input is changed to an inverter gate, then the register is loaded when the load input is 0 and inhibited when the load input is 1.

A register with parallel load can be constructed with D flip-flops, as shown in Fig. 7-3. The clock and clear inputs are the same as before. When the load input is 1, the I inputs are transferred into the register on the next clock pulse. When the load input is 0, the circuit inputs are inhibited and the D flip-flops are reloaded with their present value, thus maintaining the content of the register. The feedback connection in each flip-flop is necessary when a D type is used because a D flip-flop 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 Q output in each flip-flop.

Sequential-Logic Implementation

We saw in Chapter 6 that a clocked sequential circuit consists of a group of flip-flops and combinational gates. Since registers are readily available as MSI circuits, it becomes convenient at times to employ a register as part of the sequential circuit. A block diagram of a sequential circuit that uses a register is shown in Fig. 7-4. The present state of the register and the external inputs determine the next state of the register and the values of external outputs. Part of the combinational circuit determines the next state and the other part generates the outputs. The next state value from the combinational circuit is loaded into the register with a clock pulse. If the register has a load input, it must be set to 1; otherwise, if the register has no load input (as in Fig. 7-1), the next state value will be transferred automatically every clock pulse.

The combinational-circuit part of a sequential circuit can be implemented by any of the methods discussed in Chapter 5. It can be constructed with SSI gates, with ROM, or with a programmable logic array (PLA). By using a register, it is possible to reduce the design of a sequential circuit to that of a combinational circuit connected to a register.

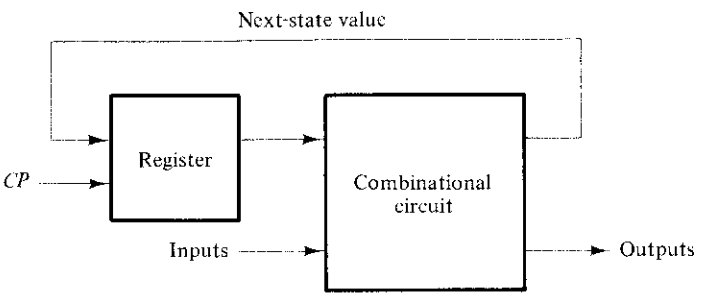


FIGURE 7-4
Block diagram of a sequential circuit

Example 7-1

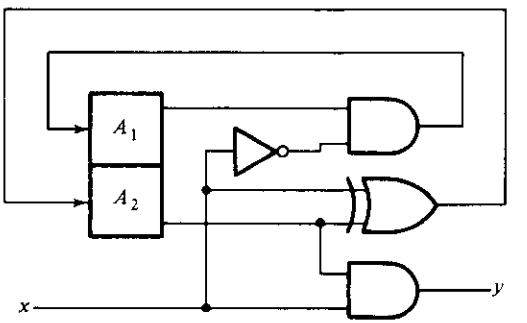
Design the sequential circuit whose state table is listed in Fig. 7-5(a).

The state table specifies two flip-flops, A_1 and A_2; one input, x; and one output, y. The next-state and output information is obtained directly from the table:

$$A_1(t + 1) = \Sigma\ (4, 6)$$
$$A_2(t + 1) = \Sigma\ (1, 2, 5, 6)$$
$$y(A_1, A_2, x) = \Sigma\ (3, 7)$$

Present state		Input	Next state		Output
A_1	A_2	x	A_1	A_2	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

(a) State table



(b) Logic diagram

FIGURE 7-5
Example of sequential-circuit implementation

The minterm values are for variables A_1, A_2, and x, which are the present-state and input variables. The functions for the next state and output can be simplified by means of maps to give

$$A_1(t + 1) = A_1x'$$
$$A_2(t + 1) = A_2 \oplus x$$
$$y = A_2x$$

The logic diagram is shown in Fig. 7-5(b). ■

Example 7-2

Repeat Example 7-1, but now use a ROM and a register.

The ROM can be used to implement the combinational circuit and the register will provide the flip-flops. The number of inputs to the ROM is equal to the number of flip-flops plus the number of external inputs. The number of outputs of the ROM is equal to the number of flip-flops plus the number of external outputs. In this case, we have three inputs and three outputs for the ROM; so its size must be 8 × 3. The implementation is shown in Fig. 7-6. The ROM truth table is identical to the state table with "present state" and "inputs" specifying the address of ROM and "next state" and "outputs" specifying the ROM outputs. The next-state values must be connected from the ROM outputs to the register inputs. ■

ROM truth table

Address			Outputs		
1	2	3	1	2	3
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

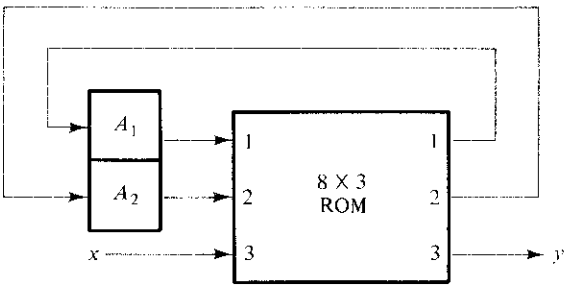


FIGURE 7-6
Sequential circuit using a register and a ROM

7-3 SHIFT REGISTERS

A register capable of shifting its binary information either to the right or to the left is called a *shift register*. The logical configuration of a shift register consists of a chain of flip-flops connected in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive a common clock pulse that causes the shift from one stage to the next.

The simplest possible shift register is one that uses only flip-flops, as shown in Fig. 7-7. The Q output of a given flip-flop is connected to the D input of the flip-flop at its right. Each clock pulse shifts the contents of the register one bit position to the right. The *serial input* determines what goes into the leftmost flip-flop during the shift. The *serial output* is taken from the output of the rightmost flip-flop prior to the application of a pulse. Although this register shifts its contents to the right, if we turn the page upside down, we find that the register shifts its contents to the left. Thus, a unidirectional shift register can function either as a shift-right or as a shift-left register.

The register in Fig. 7-7 shifts its contents with every clock pulse during the negative edge of the pulse transition. (This is indicated by the small circle associated with the clock input in all flip-flops.) If we want to control the shift so that it occurs only with certain pulses but not with others, we must control the *CP* input of the register. It will be shown later that the shift operations can be controlled through the D inputs of the flip-flops rather than through the *CP* input. If, however, the shift register in Fig. 7-7 is

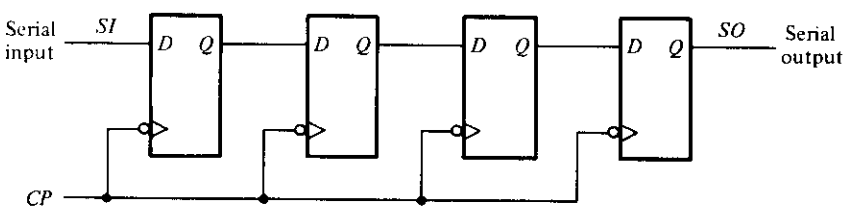


FIGURE 7-7
Shift register

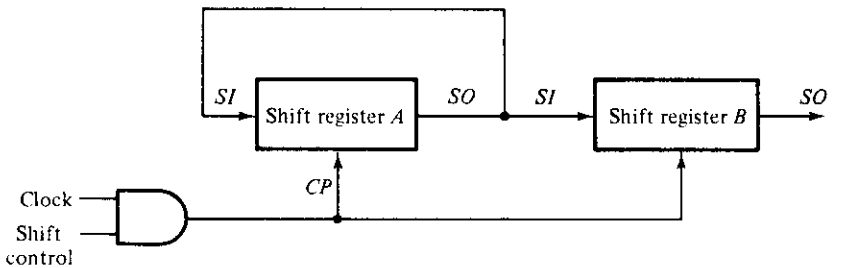
used, the shift can easily be controlled by means of an external AND gate, as shown in what follows.

Serial Transfer

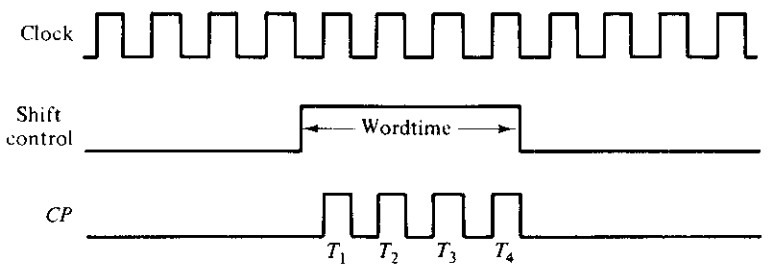
A digital system is said to operate in a serial mode when information is transferred and manipulated one bit at a time. The content of one register is transferred to another by shifting the bits from one register to the other. The information is transferred one bit at a time by shifting the bits out of the source register into the destination register.

The serial transfer of information from register *A* to register *B* is done with shift registers, as shown in the block diagram of Fig. 7-8(a). The serial output (*SO*) of register *A* goes to the serial input (*SI*) of register *B*. To prevent the loss of information stored in the source register, the *A* register is made to circulate its information by connecting the serial output to its serial input terminal. The initial content of register *B* is shifted out through its serial output and is lost unless it is transferred to a third shift register. The shift-control input determines when and by how many times the registers are shifted. This is done by the AND gate that allows clock pulses to pass into the *CP* terminals only when the shift control is 1.

Suppose the shift registers have four bits each. The control unit that supervises the transfer must be designed in such a way that it enables the shift registers, through the



(a) Block diagram



(b) Timing diagram

FIGURE 7-8
Serial transfer from register *A* to register *B*

shift-control signal, for a fixed time duration equal to four clock pulses. This is shown in the timing diagram of Fig. 7-8(b). The shift-control signal is synchronized with the clock and changes value just after the negative edge of a clock pulse. The next four clock pulses find the shift-control signal in the 1 state, so the output of the AND gate connected to the *CP* terminals produces four pulses, T_1, T_2, T_3, and T_4. The fourth pulse changes the shift control to 0 and the shift registers are disabled.

Assume that the binary content of A before the shift is 1011 and that of B, 0010. The serial transfer from A to B will occur in four steps, as shown in Table 7-1. After the first pulse, T_1, the rightmost bit of A is shifted into the leftmost bit of B and, at the same time, this bit is circulated into the leftmost position of A. The other bits of A and B are shifted once to the right. The previous serial output from B is lost and its value changes from 0 to 1. The next three pulses perform identical operations, shifting the bits of A into B, one at a time. After the fourth shift, the shift control goes to 0 and both registers A and B have the value 1011. Thus, the content of A is transferred into B, while the content of A remains unchanged.

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. The information is transferred one bit at a time while the registers are shifted in the same direction.

Computers may operate in a serial mode, a parallel mode, or in a combination of both. Serial operations are slower because of the time it takes to transfer information in and out of shift registers. Serial computers, however, require less hardware to perform operations because one common circuit can be used over and over again to manipulate the bits coming out of shift registers in a sequential manner. The time interval between clock pulses is called the *bit time*, and the time required to shift the entire contents of a shift register is called the *word time*. These timing sequences are generated by the control section of the system. In a parallel computer, control signals are enabled during one clock-pulse interval. Transfers into registers are in parallel, and they occur upon application of a single clock pulse. In a serial computer, control signals must be maintained for a period equal to one word time. The pulse applied every bit time transfers the result of the operation, one at a time, into a shift register. Most computers operate in a parallel mode because this is a faster mode of operation.

TABLE 7-1
Serial-Transfer Example

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

Bidirectional Shift Register with Parallel Load

Shift registers can be used for converting serial data to parallel data, and vice versa. If we have access to all the flip-flop outputs of a shift register, then information entered serially by shifting can be taken out in parallel from the outputs of the flip-flops. If a parallel-load capability is added to a shift register, then data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.

Some shift registers provide the necessary input and output terminals for parallel transfer. They may also have both shift-right and shift-left capabilities. The most general shift register has all the capabilities listed below. Others may have only some of these functions, with at least one shift operation.

1. A *clear* control to clear the register to 0.
2. A *CP* input for clock pulses to synchronize all operations.
3. A *shift-right* control to enable the shift-right operation and the *serial input* and *output* lines associated with the shift right.
4. A *shift-left* control to enable the shift-left operation and the *serial input* and *output* lines associated with the shift left.
5. A *parallel-load* control to enable a parallel transfer and the n input lines associated with the parallel transfer.
6. n parallel output lines.
7. A control state that leaves the information in the register unchanged even though clock pulses are continuously applied.

A register capable of shifting both right and left is called a *bidirectional shift register*. One that can shift in only one direction is called a *unidirectional shift register*. If the register has both shift and parallel-load capabilities, it is called a *shift register with parallel load*.

The diagram of a shift register that has all the capabilities listed above is shown in Fig. 7-9. It consists of four D flip-flops, although RS flip-flops could be used provided an inverter is inserted between the S and R terminals. The four multiplexers (MUX) are part of the register and are drawn here in block diagram form. (See Fig. 5-16 for the logic diagram of the multiplexer.) The four multiplexers have two common selection variables, s_1 and s_0. Input 0 in each MUX is selected when $s_1s_0 = 00$, input 1 is selected when $s_1s_0 = 01$, and similarly for the other two inputs to the multiplexers.

The s_1 and s_0 inputs control the mode of operation of the register as specified in the function entries of Table 7-2. When $s_1s_0 = 00$, the present value of the register is applied to the D inputs of the flip-flops. This condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock pulse transfers into each flip-flop the binary value it held previously, and no change of state occurs. When $s_1s_0 = 01$, terminals 1 of the multiplexer inputs have a path to the D inputs of the flip-flops. This causes a shift-right operation, with the serial input transferred into flip-flop A_4. When $s_1s_0 = 10$, a shift-left operation results, with the other serial input going into

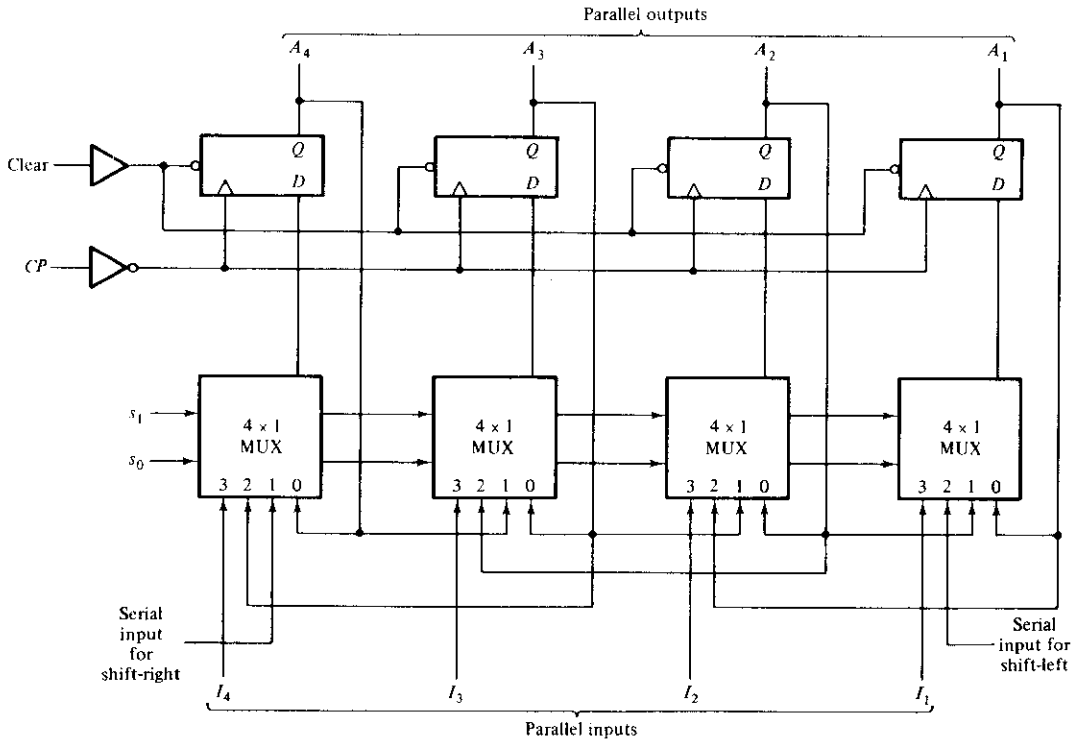


FIGURE 7-9
4-bit bidirectional shift register with parallel load

flip-flop A_1. Finally, when $s_1 s_0 = 11$, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock pulse.

A bidirectional shift register with parallel load is a general-purpose register capable of performing three operations: shift left, shift right, and parallel load. Not all shift registers available in MSI circuits have all these capabilities. The particular application dictates the choice of one MSI shift register over another.

TABLE 7-2
Function Table for the Register of Fig. 7-9

Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Serial Addition

Operations in digital computers are mostly done in parallel because this is a faster mode of operation. Serial operations are slower but require less equipment. To demonstrate the serial mode of operation, we present here the design of a serial adder. The parallel counterpart was discussed in Section 5-2.

The two binary numbers to be added serially are stored in two shift registers. Bits are added one pair at a time, sequentially, through a single full-adder (FA) circuit, as shown in Fig. 7-10. The carry out of the full-adder is transferred to a *D* flip-flop. The output of this flip-flop is then used as an input carry for the next pair of significant bits. The two shift registers are shifted to the right for one word-time period. The sum bits from the *S* output of the full-adder could be transferred into a third shift register. By shifting the sum into *A* while the bits of *A* are shifted out, it is possible to use one register for storing both the augend and the sum bits. The serial input (*SI*) of register *B* is able to receive a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is as follows. Initially, the *A* register holds the augend, the *B* register holds the addend, and the carry flip-flop is cleared to 0. The serial outputs (*SO*) of *A* and *B* provide a pair of significant bits for the full-adder at *x* and *y*. Output *Q* of the flip-flop gives the input carry at *z*. The shift-right control enables both registers and the carry flip-flop; so at the next clock pulse, both registers are shifted

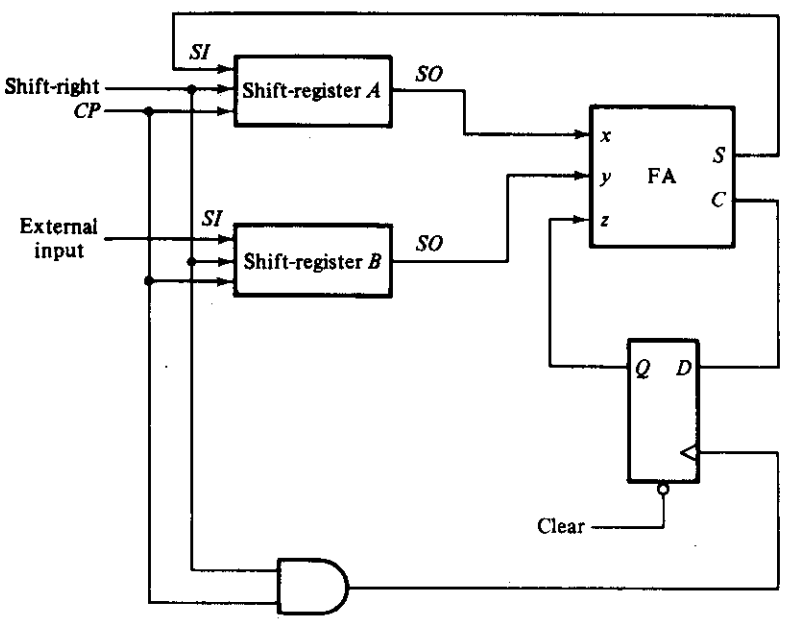


FIGURE 7-10
Serial adder

once to the right, the sum bit from S enters the leftmost flip-flop of A, and the output carry is transferred into flip-flop Q. The shift-right control enables the registers for a number of clock pulses equal to the number of bits in the registers. For each succeeding clock pulse, a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This process continues until the shift-right control is disabled. Thus, the addition is accomplished by passing each pair of bits together with the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, into register A.

If a new number has to be added to the contents of register A, this number must be first transferred serially into register B. Repeating the process once more will add the second number to the previous number in A.

Comparing the serial adder with the parallel adder described in Section 5-2, we note the following differences. The parallel adder must use registers with parallel-load capability, whereas the serial adder uses shift registers. The number of full-adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full-adder circuit and a carry flip-flop. Excluding the registers, the parallel adder is a purely combinational circuit, whereas the serial adder is a sequential circuit. The sequential circuit in the serial adder consists of a full-adder circuit and a flip-flop that stores the output carry. This is typical in serial operations because the result of a bit-time operation may depend not only on the present inputs but also on previous inputs.

To show that bit-time operations in serial computers may require a sequential circuit, we will redesign the serial adder by considering it a sequential circuit.

Example 7-3 Design a serial adder using a sequential-logic procedure.

First, we must stipulate that two shift registers are available to store the binary numbers to be added serially. The serial outputs from the registers are designated by variables x and y. The sequential circuit to be designed will not include the shift registers; they will be inserted later to show the complete unit. The sequential circuit proper has two inputs, x and y, that provide a pair of significant bits, an output S that generates the sum bit, and flip-flop Q for storing the carry. The present state of Q provides the present value of the carry. The clock pulse that shift the registers enables flip-flop Q to load the next carry. This carry is then used with the next pair of bits in x and y. The state table that specifies the sequential circuit is given in Table 7-3.

The present state of Q is the present value of the carry. The present carry in Q is added together with inputs x and y to produce the sum bit in output S. The next state of Q is equivalent to the output carry. Note that the state-table entries are identical to the entries in a full-adder truth table, except that the input carry is now the present state of Q and the output carry is now the next state of Q.

If we use a D flip-flop for Q, we obtain the same circuit as in Fig. 7-10 because the input requirements of the D input are the same as the next-state values. If we use a JK flip-flop for Q, we obtain the input excitation requirements listed in Table 7-3. The three Boolean functions of interest are the flip-flop input functions for JQ and KQ and

TABLE 7-3
Excitation Table for a Serial Adder

Present State	Inputs		Next State	Output	Flip-Flop Inputs	
Q	x	y	Q	S	JQ	KQ
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

output S. These functions are specified in the excitation table and can be simplified by means of maps:

$$JQ = xy$$
$$KQ = x'y' = (x + y)'$$
$$S = x \oplus y \oplus Q$$

As shown in Fig. 7-11, the circuit consists of three gates and a JK flip-flop. The two shift registers are also included in the diagram to show the complete serial adder. Note that output S is a function not only of x and y, but also of the present state of Q. The next state of Q is a function of the present values of x and y that come out of the serial outputs of the shift registers.

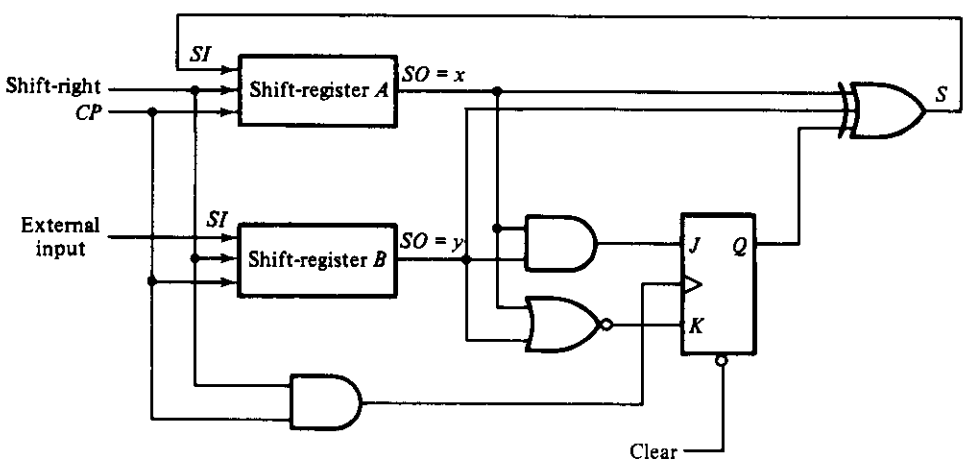


FIGURE 7-11
Second form of serial adder

■

7-4 RIPPLE COUNTERS

MSI counters come in two categories: ripple counters and synchronous counters. In a ripple counter, the flip-flop output transition serves as a source for triggering other flip-flops. In other words, the *CP* inputs of all flip-flops (except the first) are triggered not by the incoming pulses, but rather by the transition that occurs in other flip-flops. In a

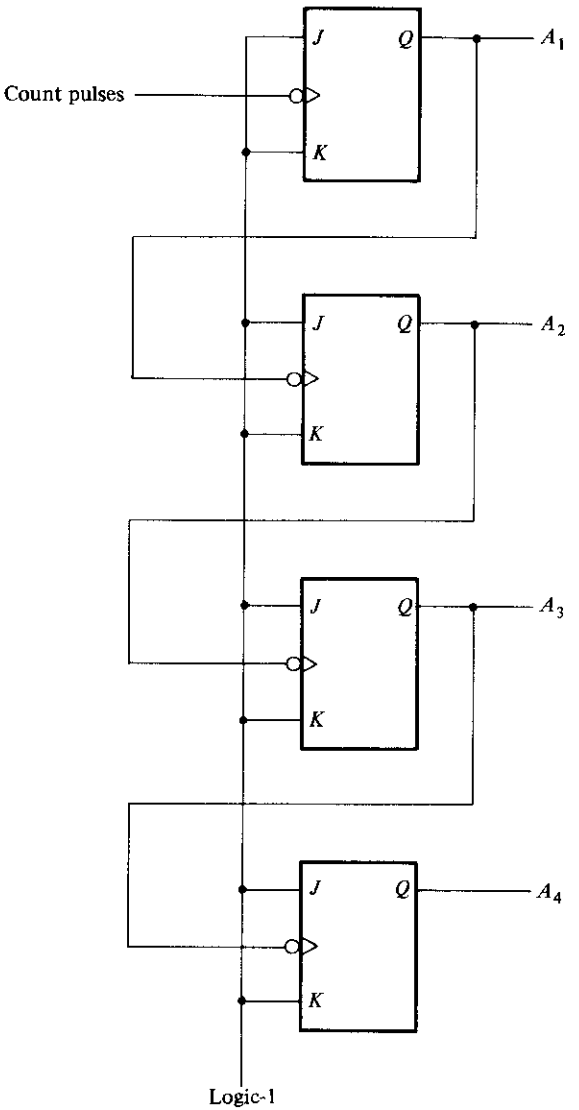


FIGURE 7-12
4-bit binary ripple counter

synchronous counter, the input pulses are applied to all *CP* inputs of all flip-flops. The change of state of a particular flip-flop is dependent on the present state of other flip-flops. Synchronous MSI counters are discussed in the next section. Here we present some common MSI ripple counters and explain their operation.

Binary Ripple Counter

A binary ripple counter consists of a series connection of complementing flip-flops (*T* or *JK* type), with the output of each flip-flop connected to the *CP* input of the next higher-order flip-flop. The flip-flop holding the least significant bit receives the incoming count pulses. The diagram of a 4-bit binary ripple counter is shown in Fig. 7-12. All *J* and *K* inputs are equal to 1. The small circle in the *CP* input indicates that the flip-flop complements during a negative-going transition or when the output to which it is connected goes from 1 to 0. To understand the operation of the binary counter, refer to its count sequence given in Table 7-4. It is obvious that the lowest-order bit A_1 must be complemented with each count pulse. Every time A_1 goes from 1 to 0, it complements A_2. Every time A_2 goes from 1 to 0, it complements A_3, and so on. For example, take the transition from count 0111 to 1000. The arrows in the table emphasize the transitions in this case. A_1 is complemented with the count pulse. Since A_1 goes from 1 to 0, it triggers A_2 and complements it. As a result, A_2 goes from 1 to 0, which in turn complements A_3. A_3 now goes from 1 to 0, which complements A_4. The output transition of A_4, if connected to a next stage, will not trigger the next flip-flop since it goes from 0 to 1. The flip-flops change one at a time in rapid succession, and the signal propagates through the counter in a *ripple* fashion. Ripple counters are sometimes called *asynchronous counters*.

TABLE 7-4
Count Sequence for a Binary Ripple Counter

Count Sequence					
A_4	A_3	A_2	A_1	Conditions for Complementing Flip-Flops	
0	0	0	0	Complement A_1	
0	0	0	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2
0	0	1	0	Complement A_1	
0	0	1	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2;
					A_2 will go from 1 to 0 and complement A_3
0	1	0	0	Complement A_1	
0	1	0	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2
0	1	1	0	Complement A_1	
0	1	1	1	Complement A_1	A_1 will go from 1 to 0 and complement A_2;
					A_2 will go from 1 to 0 and complement A_3;
					A_3 will go from 1 to 0 and complement A_4
1	0	0	0	and so on . . .	

A binary counter with a reverse count is called a binary *down-counter*. In a down-counter, the binary count is decremented by 1 with every input count pulse. The count of a 4-bit down-counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15. The circuit of Fig. 7-12 will function as a binary down-counter if the outputs are taken from the complement terminals Q' of all flip-flops. If only the normal outputs of flip-flops are available, the circuit must be modified slightly as described next.

A list of the count sequence of a count-down binary counter shows that the lowest-order bit must be complemented with every count pulse. Any other bit in the sequence is complemented if its previous lower-order bit goes from 0 to 1. Therefore, the diagram of a binary down-counter looks the same as in Fig. 7-12, provided all flip-flops trigger on the positive edge of the pulse. (The small circles in the *CP* inputs must be absent.) If negative-edge-triggered flip-flops are used, then the *CP* input of each flip-flop must be connected to the Q' output of the previous flip-flop. Then when Q goes from 0 to 1, Q' will go from 1 to 0 and complement the next flip-flop as required.

BCD Ripple Counter

A decimal counter follows a sequence of ten states and returns to 0 after the count of 9. Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits. The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit. If BCD is used, the sequence of states is as shown in the state diagram of Fig. 7-13. This is similar to a binary counter, except that the state after 1001 (code for decimal digit 9) is 0000 (code for decimal digit 0).

The design of a decimal ripple counter or of any ripple counter not following the binary sequence is not a straightforward procedure. The formal tools of logic design can serve only as a guide. A satisfactory end product requires the ingenuity and imagination of the designer.

The logic diagram of a BCD ripple counter is shown in Fig. 7-14. The four outputs are designated by the letter symbol Q with a numeric subscript equal to the binary weight of the corresponding bit in the BCD code. The flip-flops trigger on the negative

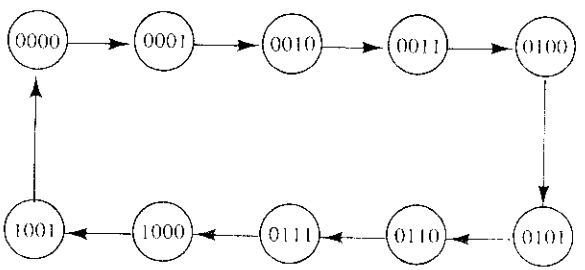


FIGURE 7-13
State diagram of a decimal BCD counter

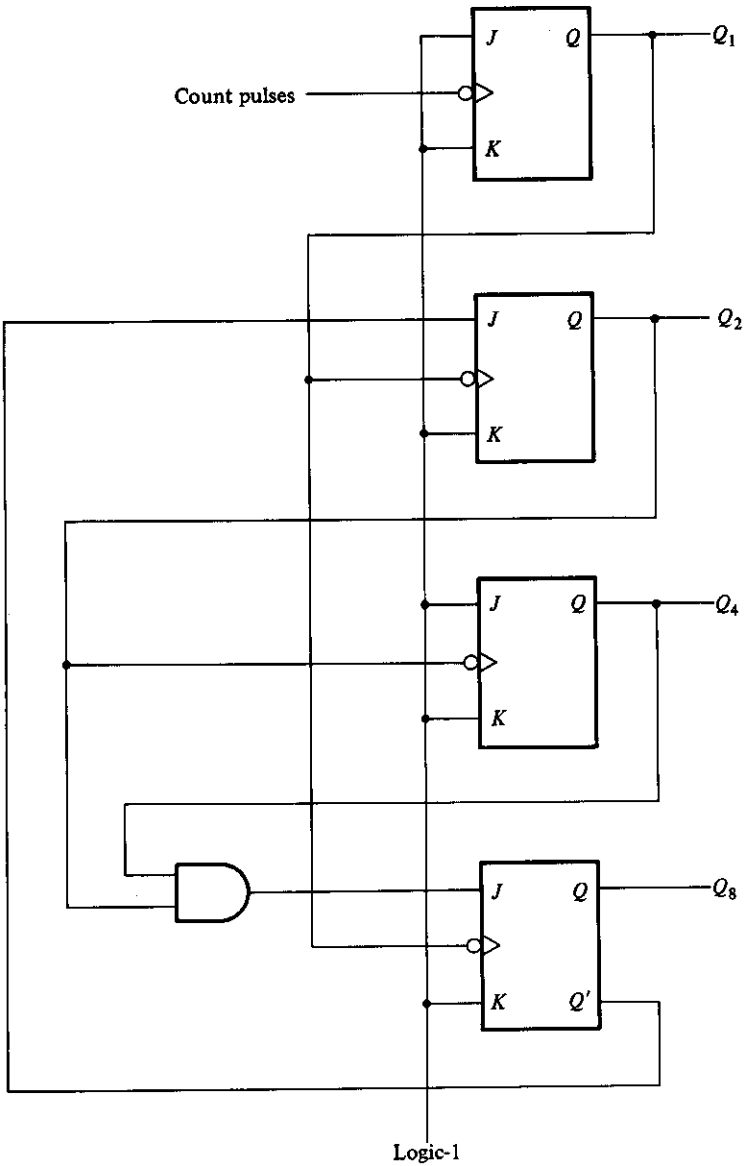


FIGURE 7-14
BCD ripple counter

edge, i.e., when the *CP* signal goes from 1 to 0. Note that the output of Q_1 is applied to the *CP* inputs of both Q_2 and Q_8 and the output of Q_2 is applied to the *CP* input of Q_4. The *J* and *K* inputs are connected either to a permanent 1 signal or to outputs of flip-flops, as shown in the diagram.

A ripple counter is an asynchronous sequential circuit and cannot be described by Boolean equations developed for describing clocked sequential circuits. Signals that affect the flip-flop transition depend on the order in which they change from 1 to 0. The operation of the counter can be explained by a list of conditions for flip-flop transitions. These conditions are derived from the logic diagram and from knowledge of how a *JK* flip-flop operates. Remember that when the *CP* input goes from 1 to 0, the flip-flop is set if $J = 1$, is cleared if $K = 1$, is complemented if $J = K = 1$, and is left unchanged if $J = K = 0$. The following are the conditions for each flip-flop state transition:

1. Q_1 is complemented on the negative edge of every count pulse.
2. Q_2 is complemented if $Q_8 = 0$ and Q_1 goes from 1 to 0. Q_2 is cleared if $Q_8 = 1$ and Q_1 goes from 1 to 0.
3. Q_4 is complemented when Q_2 goes from 1 to 0.
4. Q_8 is complemented when $Q_4Q_2 = 11$ and Q_1 goes from 1 to 0. Q_8 is cleared if either Q_4 or Q_2 is 0 and Q_1 goes from 1 to 0.

To verify that these conditions result in the sequence required by a BCD ripple counter, it is necessary to verify that the flip-flop transitions indeed follow a sequence of states as specified by the state diagram of Fig. 7-13. Another way to verify the operation of the counter is to derive the timing diagram for each flip-flop from the conditions just listed. This diagram is shown in Fig. 7-15 with the binary states listed after each clock pulse. Q_1 changes state after each clock pulse. Q_2 complements every time Q_1 goes from 1 to 0 as long as $Q_8 = 0$. When Q_8 becomes 1, Q_2 remains cleared at 0. Q_4 complements every time Q_2 goes from 1 to 0. Q_8 remains cleared as long as Q_2 or Q_4 is 0. When both Q_2 and Q_4 become 1's, Q_8 complements when Q_1 goes from 1 to 0. Q_8 is cleared on the next transition of Q_1.

The BCD counter of Fig. 7-14 is a *decade* counter, since it counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999,

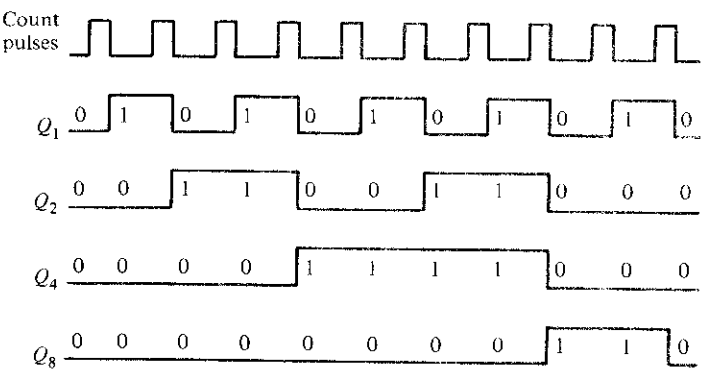


FIGURE 7-15
Timing diagram for the decimal counter of Fig. 7-14

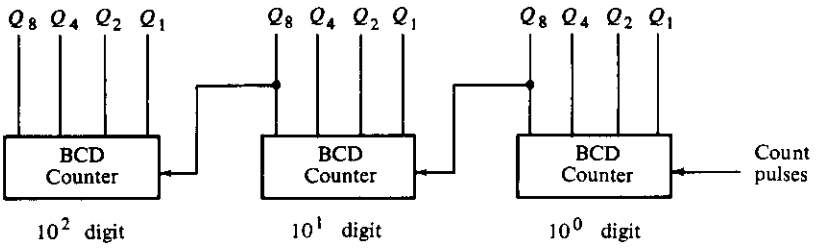


FIGURE 7-16
Block diagram of a three-decade decimal BCD counter

we need a three-decade counter. Multiple-decade counters can be constructed by connecting BCD counters in cascade, one for each decade. A three-decade counter is shown in Fig. 7-16. The inputs to the second and third decades come from Q_8 of the previous decade. When Q_8 in one decade goes from 1 to 0, it triggers the count for the next higher-order decade while its own decade goes from 9 to 0. For instance, the count after 399 will be 400.

7-5 SYNCHRONOUS COUNTERS

Synchronous counters are distinguished from ripple counters in that clock pulses are applied to the *CP* inputs of *all* flip-flops. The common pulse triggers all the flip-flops simultaneously, rather than one at a time in succession as in a ripple counter. The decision whether a flip-flop is to be complemented or not is determined from the values of the J and K inputs at the time of the pulse. If $J = K = 0$, the flip-flop remains unchanged. If $J = K = 1$, the flip-flop complements.

A design procedure for any type of synchronous counter was presented in Section 6-8. The design of a 3-bit binary counter was carried out in detail and is illustrated in Fig. 6-34. In this section, we present some typical MSI synchronous counters and explain their operation. It must be realized that there is no need to design a counter if it is already available commercially in IC form.

Binary Counter

The design of synchronous binary counters is so simple that there is no need to go through a rigorous sequential-logic design process. In a synchronous binary counter, the flip-flop in the lowest-order position is complemented with every pulse. This means that its J and K inputs must be maintained at logic-1. A flip-flop in any other position is complemented with a pulse provided all the bits in the lower-order positions are equal to 1, because the lower-order bits (when all 1's) will change to 0's on the next count pulse. The binary count dictates that the next higher-order bit be complemented. For example, if the present state of a 4-bit counter is $A_4A_3A_2A_1 = 0011$, the next count will be 0100. A_1 is always complemented. A_2 is complemented because the present state

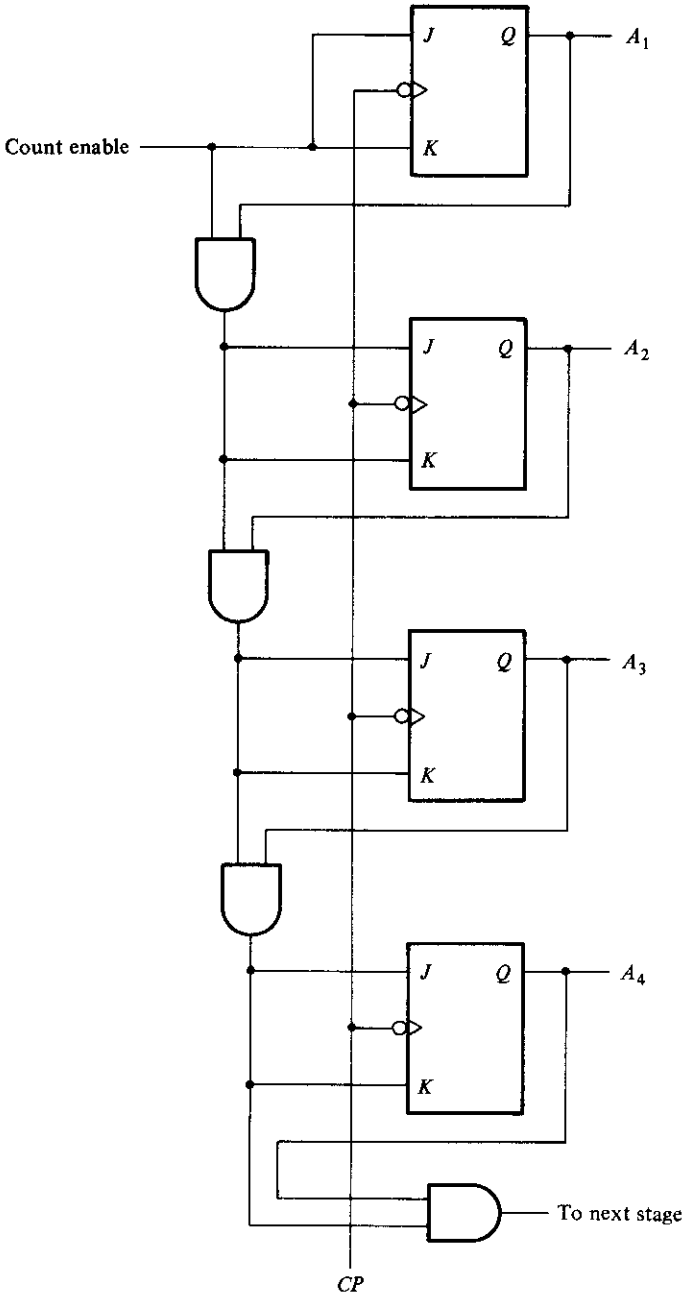


FIGURE 7-17
4-bit synchronous binary counter

of $A_1 = 1$. A_3 is complemented because the present state of $A_2A_1 = 11$. But A_4 is not complemented because the present state of $A_3A_2A_1 = 011$, which does not give an all-1's condition.

Synchronous binary counters have a regular pattern and can easily be constructed with complementing flip-flops and gates. The regular pattern can be clearly seen from the 4-bit counter depicted in Fig. 7-17. The *CP* terminals of all flip-flops are connected to a common clock-pulse source. The first stage A_1 has its J and K equal to 1 if the counter is enabled. The other J and K inputs are equal to 1 if all previous low-order bits are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the J and K inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1's.

Note that the flip-flops trigger on the negative edge of the pulse. This is not essential here as it was with the ripple counter. The counter could also be triggered on the positive edge of the pulse.

Binary Up–Down Counter

In a synchronous count-down binary counter, the flip-flop in the lowest-order position is complemented with every pulse. A flip-flop in any other position is complemented with a pulse provided all the lower-order bits are equal to 0. For example, if the present state of a 4-bit count-down binary counter is $A_4A_3A_2A_1 = 1100$, the next count will be 1011. A_1 is always complemented. A_2 is complemented because the present state of $A_1 = 0$. A_3 is complemented because the present state of $A_2A_1 = 00$. But A_4 is not complemented because the present state of $A_3A_2A_1 = 100$, which is not an all-0's condition.

A count-down binary counter can be constructed as shown in Fig. 7-17, except that the inputs to the AND gates must come from the complement outputs Q' and not from the normal outputs Q of the previous flip-flops. The two operations can be combined in one circuit. A binary counter capable of counting either up or down is shown in Fig. 7-18. The T flip-flops employed in this circuit may be considered as JK flip-flops with the J and K terminals tied together. When the up input control is 1, the circuit counts up, since the T inputs receive their signals from the values of the previous normal outputs of the flip-flops. When the down input control is 1 and the up input is 0, the circuit counts down, since the complemented outputs of the previous flip-flops are applied to the T inputs. When the up and down inputs are both 0, the circuit does not change state but remains in the same count. When the up and down inputs are both 1, the circuit counts up. This ensures that only one operation is performed at any given time.

BCD Counter

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular

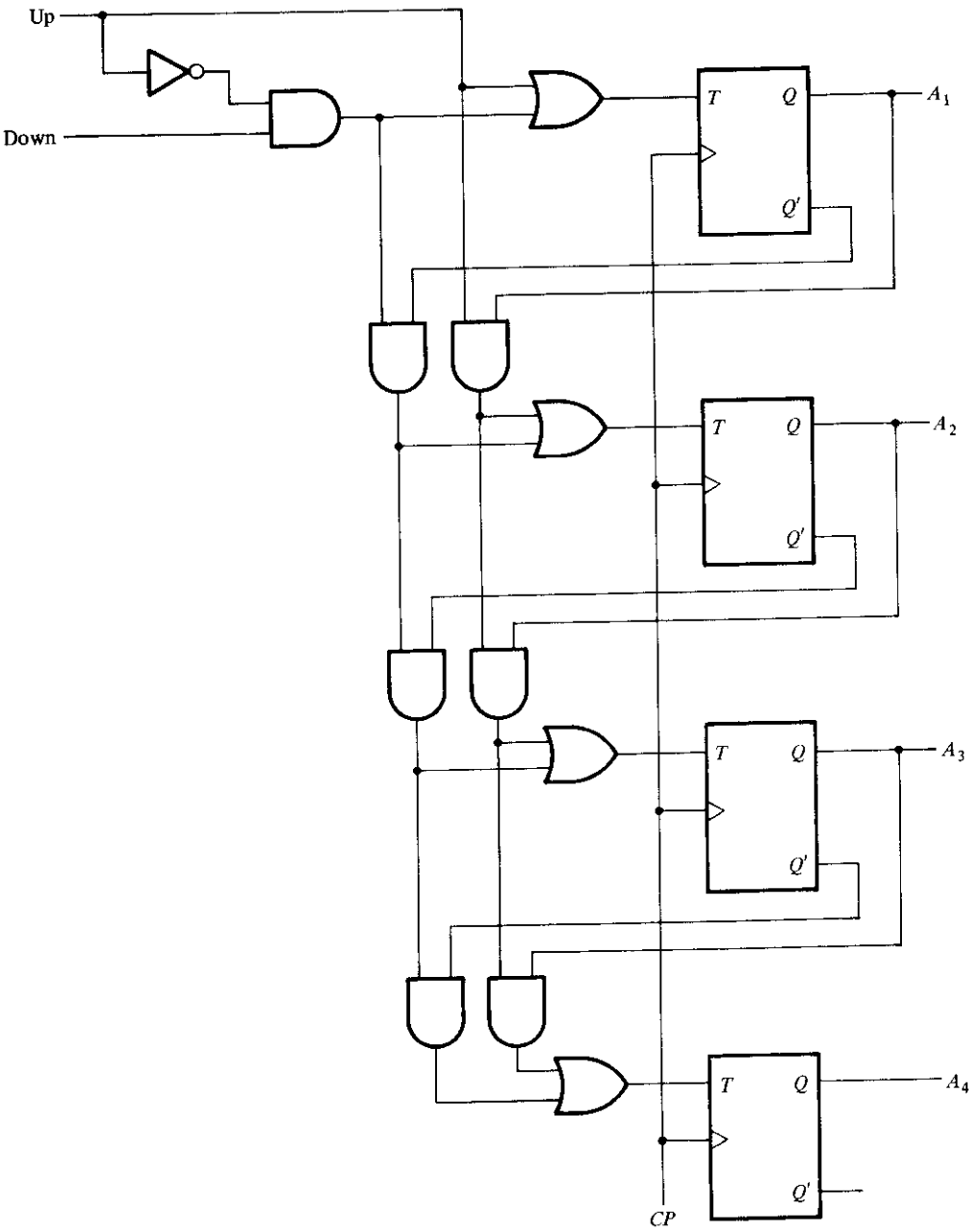


FIGURE 7-18
4-bit up–down counter

TABLE 7-5
Excitation Table for BCD Counter

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

pattern as in a straight binary count. To derive the circuit of a BCD synchronous counter, it is necessary to go through a design procedure as discussed in Section 6-8.

The excitation table of a BCD counter is given in Table 7-5. The excitation for the T flip-flops is obtained from the present and next state conditions. An output y is also shown in the table. This output is equal to 1 when the counter present state is 1001. In this way, y can enable the count of the next-higher-order decade while the same pulse switches the present decade from 1001 to 0000.

The flip-flop input functions from the excitation table can be simplified by means of maps. The unused states for minterms 10 to 15 are taken as don't-care terms. The simplified functions are

$$TQ_1 = 1$$
$$TQ_2 = Q_8' Q_1$$
$$TQ_4 = Q_2 Q_1$$
$$TQ_8 = Q_8 Q_1 + Q_4 Q_2 Q_1$$
$$y = Q_8 Q_1$$

The circuit can be easily drawn with four T flip-flops, five AND gates, and one OR gate.

Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length. The cascading is done as in Fig. 7-16, except that output y must be connected to the count input of the next-higher-order decade.

Binary Counter with Parallel Load

Counters employed in digital systems quite often require a parallel-load capability for transferring an initial binary number prior to the count operation. Figure 7-19 shows the logic diagram of a register that has a parallel-load capability and can also operate as

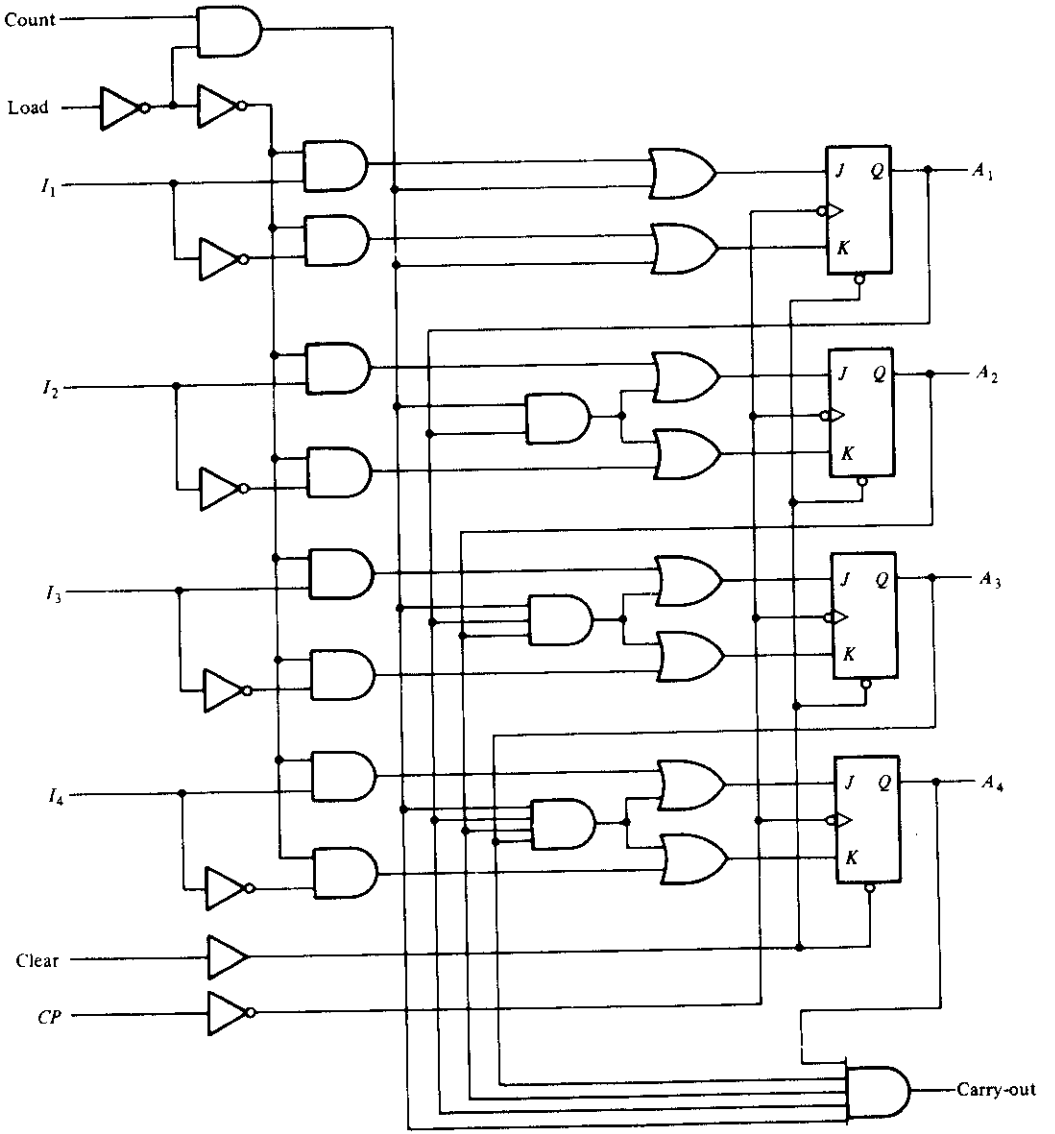


FIGURE 7-19
4-bit binary counter with parallel load

a counter. The input load control when equal to 1 disables the count sequence and causes a transfer of data from inputs I_1 through I_4 into flip-flops A_1 through A_4, respectively. If the load input is 0 and the count input control is 1, the circuit operates as a counter. The clock pulses then cause the state of the flip-flops to change according to the binary count sequence. If both control inputs are 0, clock pulses do not change the state of the register.

The carry-out terminal becomes a 1 if all flip-flops are equal to 1 while the count input is enabled. This is the condition for complementing the flip-flop holding the next-higher-order bit. This output is useful for expanding the counter to more than four bits. The speed of the counter is increased if this carry is generated directly from the outputs of all four flip-flops instead of going through a chain of AND gates. Similarly, each flip-flop is associated with an AND gate that receives all previous flip-flop outputs directly to determine when the flip-flop should be complemented.

The operation of the counter is summarized in Table 7-6. The four control inputs: clear, *CP*, load, and count determine the next output state. The clear input is asynchronous and, when equal to 0, causes the counter to be cleared to all 0's, regardless of the presence of clock pulses or other inputs. This is indicated in the table by the *X* entries, which symbolize don't-care conditions for the other inputs, so their value can be either 0 or 1. The clear input must go to the 1 state for the clocked operations listed in the next three entries in the table. With the load and count inputs both at 0, the outputs do not change, whether a pulse is applied in the *CP* terminal or not. A load input of 1 causes a transfer from inputs I_1–I_4 into the register during the positive edge of an input pulse. The input information is loaded into the register regardless of the value of the count input, because the count input is inhibited when the load input is 1. If the load input is maintained at 0, the count input controls the operation of the counter. The outputs change to the next binary count on the positive-edge transition of every clock pulse, but no change of state occurs if the count input is 0.

The 4-bit counter shown in Fig. 7-19 can be enclosed in one IC package. Two ICs are necessary for the construction of an 8-bit counter; four ICs for a 16-bit counter; and so on. The carry output of one IC must be connected to the count input of the IC holding the four next-higher-order bits of the counter.

Counters with parallel-load capability having a specified number of bits are very useful in the design of digital systems. Later, we will refer to them as registers with load and increment capabilities. The *increment* function is an operation that adds 1 to the present content of a register. By enabling the count control during one clock pulse period, the content of the register can be incremented by 1.

A counter with parallel load can be used to generate any desired number of count sequences. A modulo-*N* (abbreviated mod-*N*) counter is a counter that goes through a repeated sequence of *N* counts. For example, a 4-bit binary counter is a mod-16 counter. A BCD counter is a mod-10 counter. In some applications, one may not be concerned with the particular *N* states that a mod-*N* counter uses. If this is the case, then a coun-

TABLE 7-6
Function Table for the Counter of Fig. 7-19

Clear	*CP*	Load	Count	Function
0	*X*	*X*	*X*	Clear to 0
1	*X*	0	0	No change
1	↑	1	*X*	Load inputs
1	↑	0	1	Count next binary state

ter with parallel load can be used to construct any mod-N counter, with N being any value desired. This is shown in the following example.

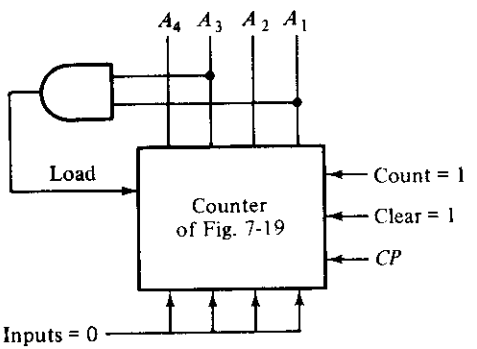
Example 7-4

Construct a mod-6 counter using the MSI circuit specified in Fig. 7-19.

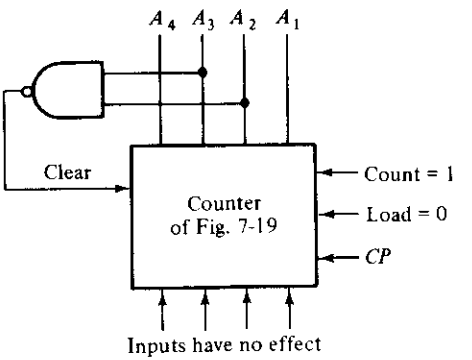
Figure 7-20 shows four ways in which a counter with parallel load can be used to generate a sequence of six counts. In each case, the count control is set to 1 to enable the count through the pulses in the *CP* input. We also use the facts that the load control inhibits the count and that the clear operation is independent of other control inputs.

The AND gate in Fig. 7-20(a) detects the occurrence of state 0101 in the output. When the counter is in this state, the load input is enabled and an all-0's input is loaded into the register. Thus, the counter goes through binary states 0, 1, 2, 3, 4, and 5 and then returns to 0. This produces a sequence of six counts.

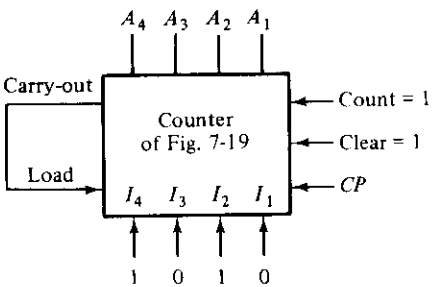
The clear input of the register is asynchronous, i.e., it does not depend on the clock. In Fig. 7-20(b), the NAND gate detects the count of 0110, but as soon as this count occurs, the register is cleared. The count 0110 has no chance of staying on for any ap-



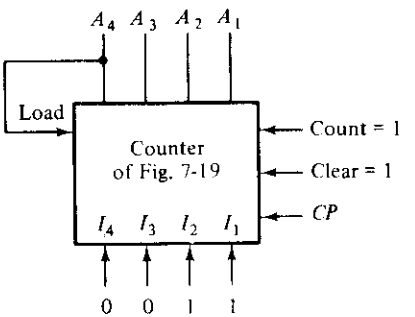
(a) Binary states 0, 1, 2, 3, 4, 5.



(b) Binary states 0, 1, 2, 3, 4, 5.



(c) Binary states 10, 11, 12, 13, 14, 15.



(d) Binary states 3, 4, 5, 6, 7, 8.

FIGURE 7-20
Four ways to achieve a mod-6 counter using a counter with parallel load

preciable time because the register goes immediately to 0. A momentary spike occurs in output A_2 as the count goes from 0101 to 0110 and immediately to 0000. This momentary spike may be undesirable, and for this reason, this configuration is not recommended. If the counter has a synchronous clear input, it would be possible to clear the counter with the clock after an occurrence of the 0101 count.

Instead of using the first six counts, we may want to choose the last six counts from 10 to 15. In this case, it is possible to take advantage of the output carry to load a number in the register. In Fig. 7-20(c), the counter starts with count 1010 and continues to 1111. The output carry generated during the last state enables the load control, which then loads the input, which is set at 1010.

It is also possible to choose any intermediate count of six states. The mod-6 counter of Fig. 7-20(d) goes through the count sequence 3, 4, 5, 6, 7, and 8. When the last count 1000 is reached, output A_4 goes to 1 and the load control is enabled. This loads the value of 0011 into the register, and the binary count continues from this state. ■

7-6 TIMING SEQUENCES

The sequence of operations in a digital system are specified by a control unit. The control unit that supervises the operations in a digital system would normally consist of timing signals that determine the time sequence in which the operations are executed. The timing sequences in the control unit can be easily generated by means of counters or shift registers. This section demonstrates the use of these MSI functions in the generation of timing signals for a control unit.

Word-Time Generation

First, we demonstrate a circuit that generates the required timing signal for serial mode of operation. Serial transfer of information was discussed in Section 7-3, with an example depicted in Fig. 7-8. The control unit in a serial computer must generate a *word-time* signal that stays on for a number of pulses equal to the number of bits in the shift registers. The word-time signal can be generated by means of a counter that counts the required number of pulses.

Assume that the word-time signal to be generated must stay on for a period of eight clock pulses. Figure 7-21(a) shows a counter circuit that accomplishes this task. Initially, the 3-bit counter is cleared to 0. A start signal will set flip-flop Q. The output of this flip-flop supplies the word-time control and also enables the counter. After the count of eight pulses, the flip-flop is reset and Q goes to 0. The timing diagram of Fig. 7-21(b) demonstrates the operation of the circuit. The start signal is synchronized with the clock and stays on for one clock-pulse period. After Q is set to 1, the counter starts counting the clock pulses. When the counter reaches the count of 7 (binary 111), it sends a stop signal to the reset input of the flip-flop. The stop signal becomes a 1 after the negative-edge transition of pulse 7. The next clock pulse switches the counter to the 000 state and also clears Q. Now the counter is disabled and the word-time signal stays

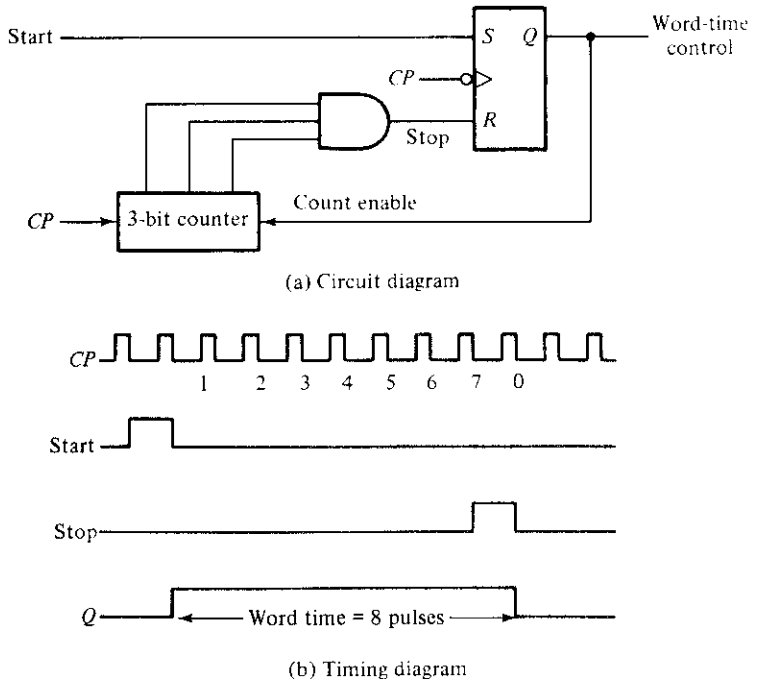


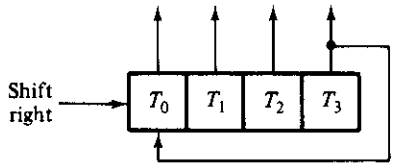
FIGURE 7-21
Generation of a word-time control for serial operations

at 0. Note that the word-time control stays on for a period of eight pulses. Note also that the stop signal in this circuit can be used to start another word-count control in another circuit just as the start signal is used in this circuit.

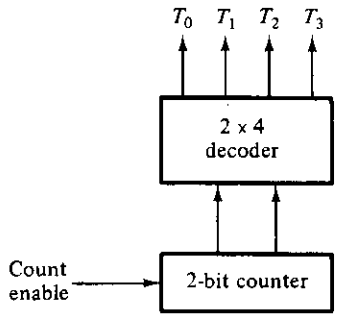
Timing Signals

In a parallel mode of operation, a single clock pulse can specify the time at which an operation should be executed. The control unit in a digital system that operates in the parallel mode must generate timing signals that stay on for only one clock pulse period, but these timing signals must be distinguished from each other.

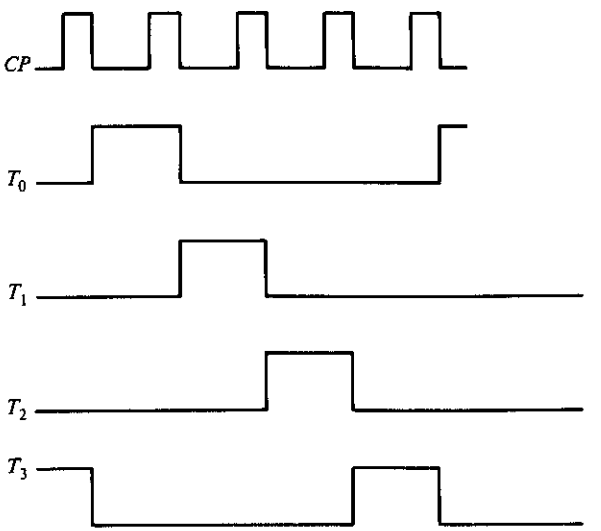
Timing signals that control the sequence of operations in a digital system can be generated with a shift register or a counter with a decoder. A *ring counter* is a circular shift register with only one flip-flop being set at any particular time; all others are cleared. The single bit is shifted from one flip-flop to the other to produce the sequence of timing signals. Figure 7-22(a) shows a 4-bit shift register connected as a ring counter. The initial value of the register is 1000, which produces the variable T_0. The single bit is shifted right with every clock pulse and circulates back from T_3 to T_0. Each flip-flop is in the 1 state once every four clock pulses and produces one of the four timing signals shown in Fig. 7-22(c). Each output becomes a 1 after the negative-edge transition of a clock pulse and remains 1 during the next clock pulse.



(a) Ringcounter (initial value = 1000)



(b) Counter and decoder



(c) Sequence of four timing signals

FIGURE 7-22
Generation of timing signals

The timing signals can be generated also by continuously enabling a 2-bit counter that goes through four distinct states. The decoder shown in Fig. 7-22(b) decodes the four states of the counter and generates the required sequence of timing signals.

The timing signals, when enabled by the clock pulses, will provide multiple-phase clock pulses. For example, if T_0 is ANDed with CP, the output of the AND gate will generate clock pulses at one-fourth the frequency of the master-clock pulses. Multiple-phase clock pulses can be used for controlling different registers with different time scales.

To generate 2^n timing signals, we need either a shift register with 2^n flip-flops or an n-bit counter together with an n-to-2^n-line decoder. For example, 16 timing signals can be generated with a 16-bit shift register connected as a ring counter or with a 4-bit counter and a 4-to-16-line decoder. In the first case, we need 16 flip-flops. In the second case, we need four flip-flops and 16 4-input AND gates for the decoder. It is also possible to generate the timing signals with a combination of a shift register and a decoder. In this way, the number of flip-flops is less than a ring counter, and the decoder requires only 2-input gates. This combination is sometimes called a *Johnson counter*.

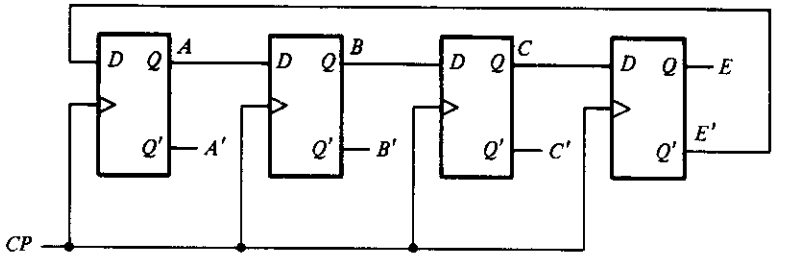
Johnson Counter

A k-bit ring counter circulates a single bit among the flip-flops to provide k distinguishable states. The number of states can be doubled if the shift register is connected as a *switch-tail* ring counter. A switch-tail ring counter is a circular shift register with the complement output of the last flip-flop connected to the input of the first flip-flop. Figure 7-23(a) shows such a shift register. The circular connection is made from the complement output of the rightmost flip-flop to the input of the leftmost flip-flop. The register shifts its contents once to the right with every clock pulse, and at the same time, the complement value of the E flip-flop is transferred into the A flip-flop. Starting from a cleared state, the switch-tail ring counter goes through a sequence of eight states, as listed in Fig. 7-23(b). In general, a k-bit switch-tail ring counter will go through a sequence of $2k$ states. Starting from all 0's, each shift operation inserts 1's from the left until the register is filled with all 1's. In the following sequences, 0's are inserted from the left until the register is again filled with all 0's.

A Johnson counter is a k-bit switch-tail ring counter with $2k$ decoding gates to provide outputs for $2k$ timing signals. The decoding gates are not shown in Fig. 7-23, but are specified in the last column of the table. The eight AND gates listed in the table, when connected to the circuit, will complete the construction of the Johnson counter. Since each gate is enabled during one particular state sequence, the outputs of the gates generate eight timing sequences in succession.

The decoding of a k-bit switch-tail ring counter to obtain $2k$ timing sequences follows a regular pattern. The all-0's state is decoded by taking the complement of the two extreme flip-flop outputs. The all-1's state is decoded by taking the normal outputs of the two extreme flip-flops. All other states are decoded from an adjacent 1, 0 or 0, 1 pattern in the sequence. For example, sequence 7 has an adjacent 0, 1 pattern in flip-flops B and C. The decoded output is then obtained by taking the complement of B and the normal output of C, or $B'C$.

One disadvantage of the circuit in Fig. 7-23(a) is that if it finds itself in an unused state, it will persist in moving from one invalid state to another and never find its way



(a) Four-stage switch-tail ring counter

Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

(b) Count sequence and required decoding.

FIGURE 7-23
Construction of a Johnson counter

to a valid state. This difficulty can be corrected by modifying the circuit to avoid this undesirable condition. One correcting procedure is to disconnect the output from flip-flop B that goes to the D input of flip-flop C, and instead enable the input of flip-flop C by the function:

$$DC = (A + C)B$$

where DC is the flip-flop input function for the D input of flip-flop C.

Johnson counters can be constructed for any number of timing sequences. The number of flip-flops needed is one-half the number of timing signals. The number of decoding gates is equal to the number of timing signals and only 2-input gates are employed.

7-7 RANDOM-ACCESS MEMORY (RAM)

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the device. Memory cells can be accessed for information transfer to or from any desired random location and hence the name *random-access memory*, abbreviated RAM.

A memory unit stores binary information in groups of bits called *words*. A word in

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

The communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer. A block diagram of the memory unit is shown in Fig. 7-24. The n data input lines provide the information to be stored in memory and the n data output lines supply the information coming out of memory. The k address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: The write input causes binary data to be transferred into the memory, and the read input causes binary data to be transferred out of memory.

The memory unit is specified by the number of words it contains and the number of bits in each word. The address lines select one particular word. Each word in memory is assigned an identification number, called an address, starting from 0 and continuing with 1, 2, 3, up to $2^k - 1$, where k is the number of address lines. The selection of a specific word inside the memory is done by applying the k-bit binary address to the address lines. A decoder inside the memory accepts this address and opens the paths needed to select the word specified. Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits. It is customary to refer to the number of words (or bytes) in a memory with one of the letters K (kilo), M (mega), or G (giga). K is equal to 2^{10}, M is equal to 2^{20}, and G is equal to 2^{30}. Thus, $64K = 2^{16}$, $2M = 2^{21}$, and $4G = 2^{32}$.

Consider, for example, the memory unit with a capacity of 1K words of 16 bits each. Since $1K = 1024 = 2^{10}$ and 16 bits constitute two bytes, we can say that the memory can accommodate 2048 = 2K bytes. Figure 7-25 shows the possible content of the first three and the last three words of this memory. Each word contains 16 bits,

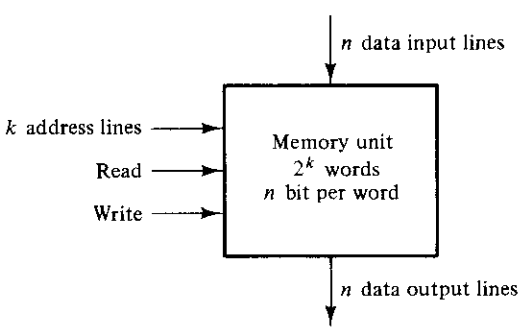


FIGURE 7-24
Block diagram of a memory unit

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

FIGURE 7-25
Content of a 1024 × 16 memory

which can be divided into two bytes. The words are recognized by their decimal address from 0 to 1023. The equivalent binary address consists of 10 bits. The first address is specified with ten 0's, and the last address is specified with ten 1's. This is because 1023 in binary is equal to 1111111111. A word in memory is selected by its binary address. When a word is read or written, the memory operates on all 16 bits as a single unit.

The 1K × 16 memory of Fig. 7-25 has 10 bits in the address and 16 bits in each word. As another example, a 64K × 10 memory will have 16 bits in the address (since $64K = 2^{16}$) and each word will consist of 10 bits. The number of address bits needed in a memory is dependent on the total number of words that can be stored in the memory and is independent of the number of bits in each word. The number of bits in the address is determined from the relationship $2^k = m$, where m is the total number of words, and k is the number of address bits.

Write and Read Operations

The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function. The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

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

TABLE 7-7
Control Inputs to Memory Chip

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

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

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

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

The memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The content of the selected word does not change after reading.

Commercial memory components available in integrated-circuit chips sometimes provide the two control inputs for reading and writing in a somewhat different configuration. Instead of having separate read and write inputs to control the two operations, some integrated circuits provide two other control inputs: one input selects the unit and the other determines the operation. The memory operations that result from these control inputs are specified in Table 7-7.

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

Types of Memories

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

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

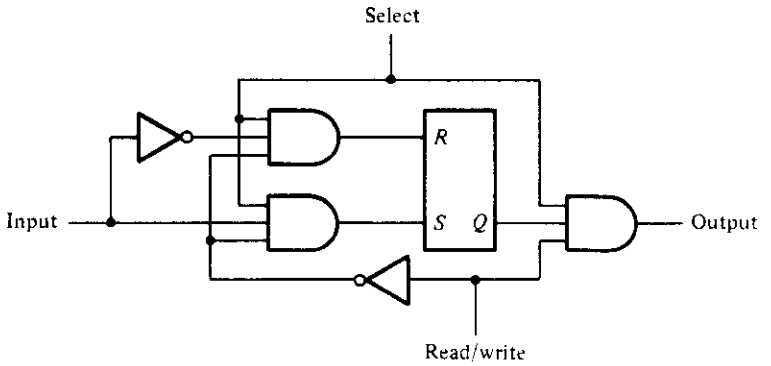
Memory units that lose the stored information when power is turned off are said to be *volatile*. Integrated-circuit RAMs, both static and dynamic, are of this category since the binary cells need external power to maintain the stored information. In contrast, a nonvolatile memory, such as magnetic disk, retains its stored information after removal of power. This is because the data stored on magnetic components is manifested by the direction of magnetization, which is retained after power is turned off. Another nonvolatile memory is the read-only memory (ROM) discussed in Section 5-7. A nonvolatile property is desirable in digital computers to store programs that are needed while the computer is in operation. Programs and data that cannot be altered are stored in ROM. Other large programs are maintained on magnetic disks. When power is turned on, the computer can use the programs from ROM. The other programs residing on disks can be transferred into the computer RAM as needed. Before turning the power off, the user transfers the binary information from the computer RAM into a disk if this information must be retained.

7-8 MEMORY DECODING

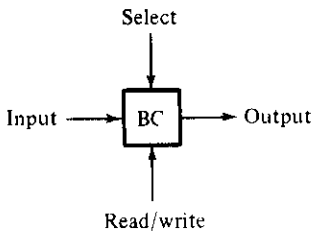
In addition to the storage components in a memory unit, there is a need for decoding circuits to select the memory word specified by the input address. In this section, we present the internal construction of a random-access memory and demonstrate the operation of the decoder. To be able to include the entire memory in one diagram, the memory unit presented here has a small capacity of 12 bits arranged in 4 words of 3 bits each. In addition to internal decoders, a memory unit may also need external decoders. This happens when integrated-circuit RAM chips are connected in a multichip memory configuration. The use of an external decoder to provide a large capacity memory will be demonstrated by means of an example.

Internal Construction

The internal construction of a random-access memory of m words with n bits per word consists of $m \times n$ binary storage cells and associated decoding circuits for selecting individual words. The binary storage cell is the basic building block of a memory unit.



(a) Logic diagram



(b) Block diagram

FIGURE 7-26
Memory cell

The equivalent logic of a binary cell that stores one bit of information is shown in Fig. 7-26. Although the cell is shown to include gates and a flip-flop, internally, it is constructed with two transistors having multiple inputs. A binary storage cell must be very small in order to be able to pack as many cells as possible in the area available in the integrated-circuit chip. The binary cell stores one bit in its internal flip-flop. It has three inputs and one output. The select input enables the cell for reading or writing and the read/write input determines the cell operation when it is selected. A 1 in the read/write input provides the read operation by forming a path from the flip-flop to the output terminal. A 0 in the read/write input provides the write operation by forming a path from the input terminal to the flip-flop. Note that the flip-flop operates without a clock and is similar to an *SR* latch (see Fig. 6-2).

The logical construction of a small RAM is shown in Fig. 7-27. It consists of 4 words of 3 bits each and has a total of 12 binary cells. Each block labeled *BC* represents the binary cell with its three inputs and one output, as specified in Fig. 7-26(b). A memory with four words needs two address lines. The two address inputs go through a 2×4 decoder to select one of the four words. The decoder is enabled with the memory-enable input. When the memory enable is 0, all outputs of the decoder are 0

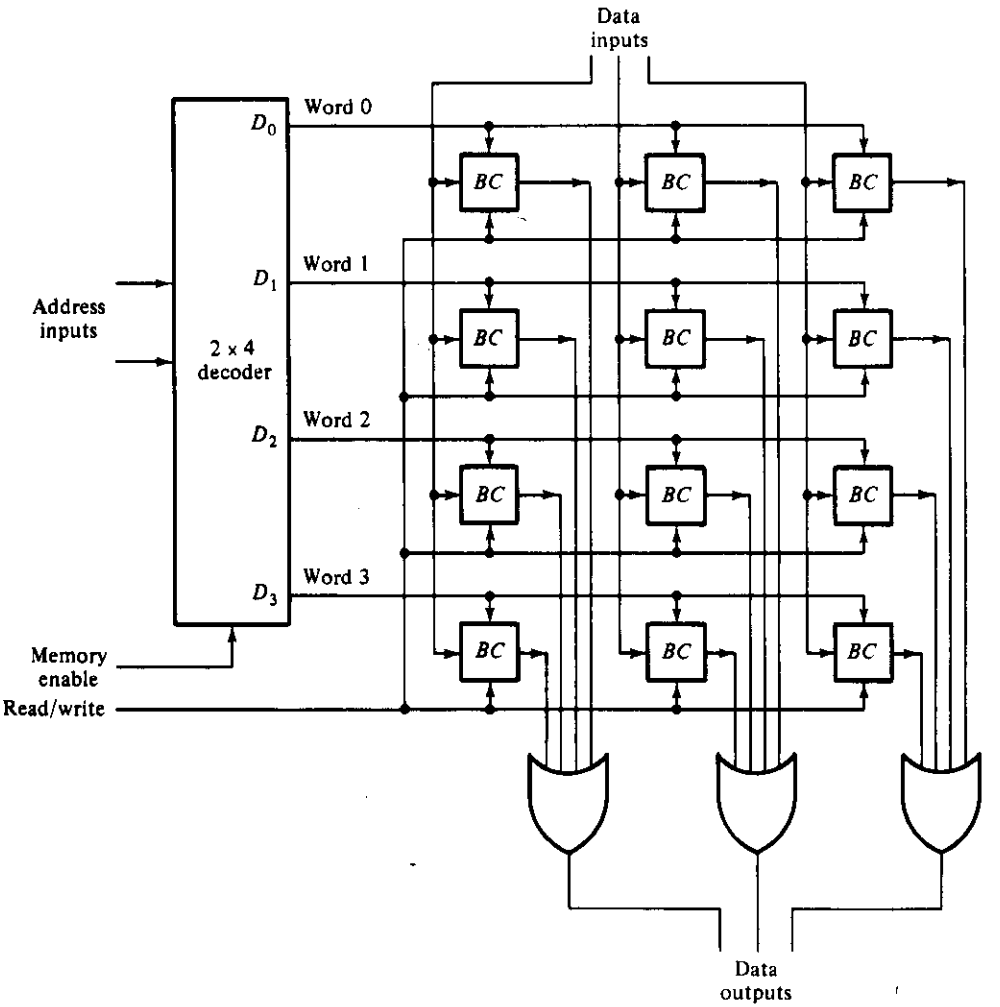


FIGURE 7-27
Logical construction of a 4 × 3 RAM

and none of the memory words are selected. With the memory enable at 1, one of the four words is selected, dictated by the value in the two address lines. Once a word has been selected, the read/write input determines the operation. During the read operation, the four bits of the selected word go through OR gates to the output terminals. During the write operation, the data available in the input lines are transferred into the four binary cells of the selected word. The binary cells that are not selected are disabled and their previous binary values remain unchanged. When the memory-enable input that goes into the decoder is equal to 0, none of the words are selected and the contents of all cells remain unchanged regardless of the value of the read/write input.

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

Array of RAM Chips

Integrated-circuit RAM chips are available in a variety of sizes. If the memory unit needed for an application is larger than the capacity of one chip, it is necessary to combine a number of chips in an array to form the required memory size. The capacity of the memory depends on two parameters: the number of words and the number of bits per word. An increase in the number of words requires that we increase the address length. Every bit added to the length of the address doubles the number of words in memory. The increase in the number of bits per word requires that we increase the length of the data input and output lines, but the address length remains the same.

To demonstrate with an example, let us first introduce a typical RAM chip, as shown in Fig. 7-28. The capacity of the RAM is 1024 words of 8 bits each. It requires a 10-bit address and 8 input and output lines. These are shown in the block diagram by a single line and a number indicating the total number of inputs or outputs. The chip-select (*CS*) input selects the particular RAM chip and the read/write (RW) input specifies the read or write operation when the chip is selected.

Suppose that we want to increase the number of words in the memory by using two or more RAM chips. Since every bit added to the address doubles the binary number that can be formed, it is natural to increase the number of words in factors of 2. For example, two RAM chips will double the number of words and add one bit to the composite address. Four RAM chips multiply the number of words by 4 and add two bits to the composite address.

Consider the possiblity of constructing a 4K × 8 RAM with four 1K × 8 RAM chips. This is shown in Fig. 7-29. The 8 input data lines go to all the chips. The outputs must be ORed together to form the common 8 output data lines. (The OR gates are not shown in the diagram.) The 4K word memory requires a 12-bit address. The 10 least significant bits of the address are applied to the address inputs of all four chips.

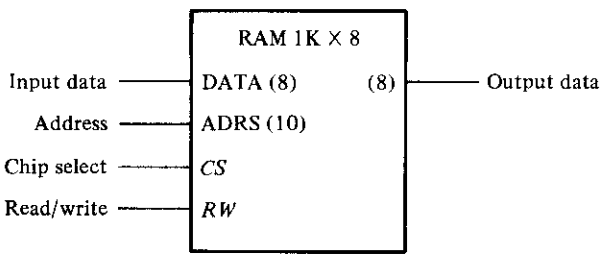


FIGURE 7-28
Block diagram of a 1K × 8 RAM chip.

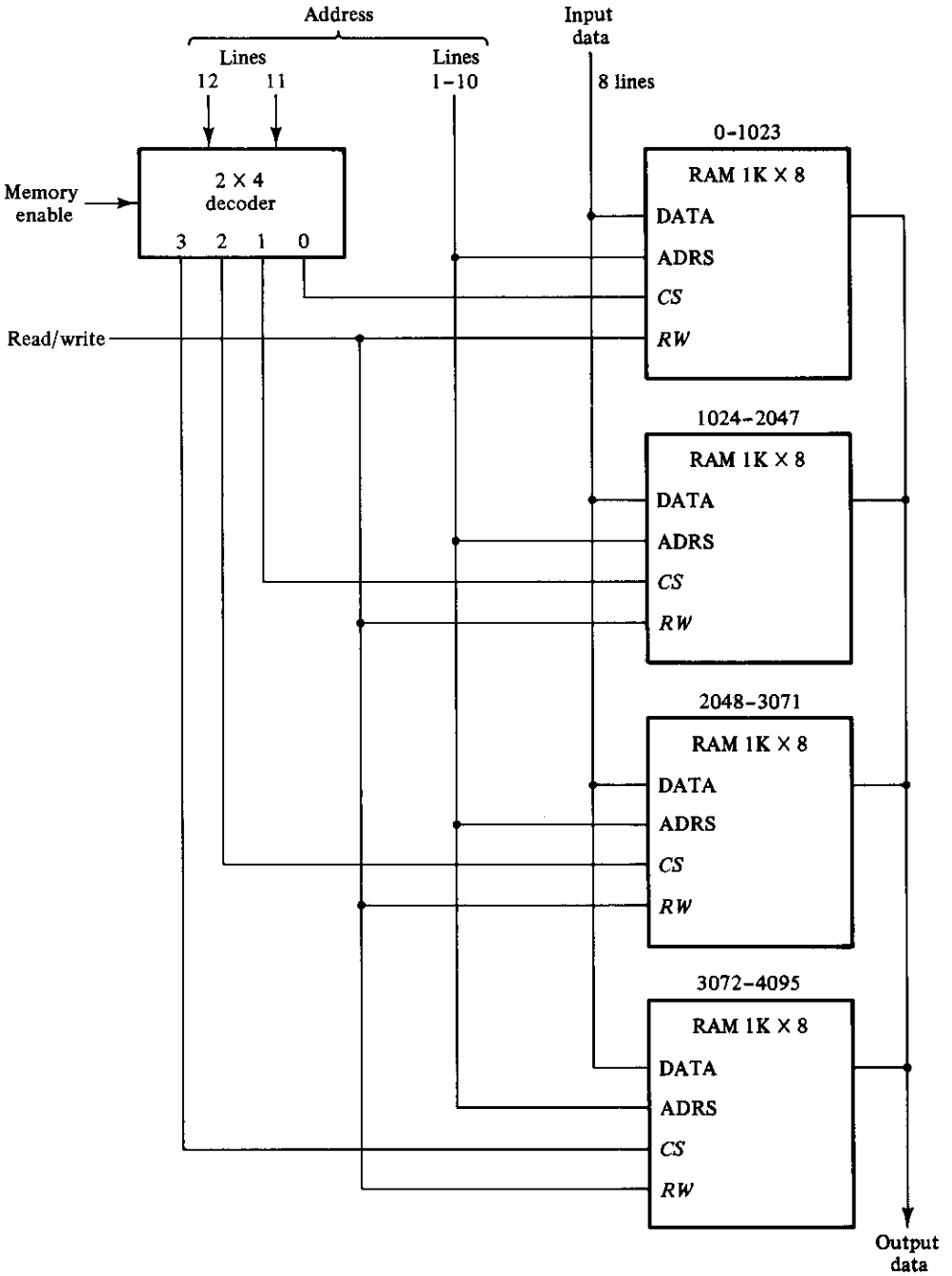


FIGURE 7-29
Block diagram of a 4K × 8 RAM.

The other two most significant bits are applied to a 2 × 4 decoder. The four outputs of the decoder are applied to the *CS* inputs of each chip. The memory is disabled when the memory-enable input of the decoder is equal to 0. This causes all four outputs of the decoder to be in the 0 state and none of the chips are selected. When the decoder is enabled, address bits 12 and 11 determine the particular chip that is selected. If bits 12 and 11 are equal to 00, the first RAM chip is selected. The remaining ten address bits select a word within the chip in the range from 0 to 1023. The next 1024 words are selected from the second RAM chip with a 12-bit address that starts with 01 and follows by the ten bits from the common address lines. The address range for each chip is listed in decimal over its block diagram in Fig. 7-29.

It is also possible to combine two chips to form a composite memory containing the same number of words but with twice as many bits in each word. Figure 7-30 shows the interconnection of two 1K × 8 chips to form a 1K × 16 memory. The 16 input and output data lines are split between the two chips. Both receive the same 10-bit address and the common *CS* and *RW* control inputs.

The two techniques just described may be combined to assemble an array of identical chips into a large-capacity memory. The composite memory will have a number of bits per word that is a multiple of that for one chip. The total number of words will increase in factors of 2 times the word capacity of one chip. An external decoder is needed to

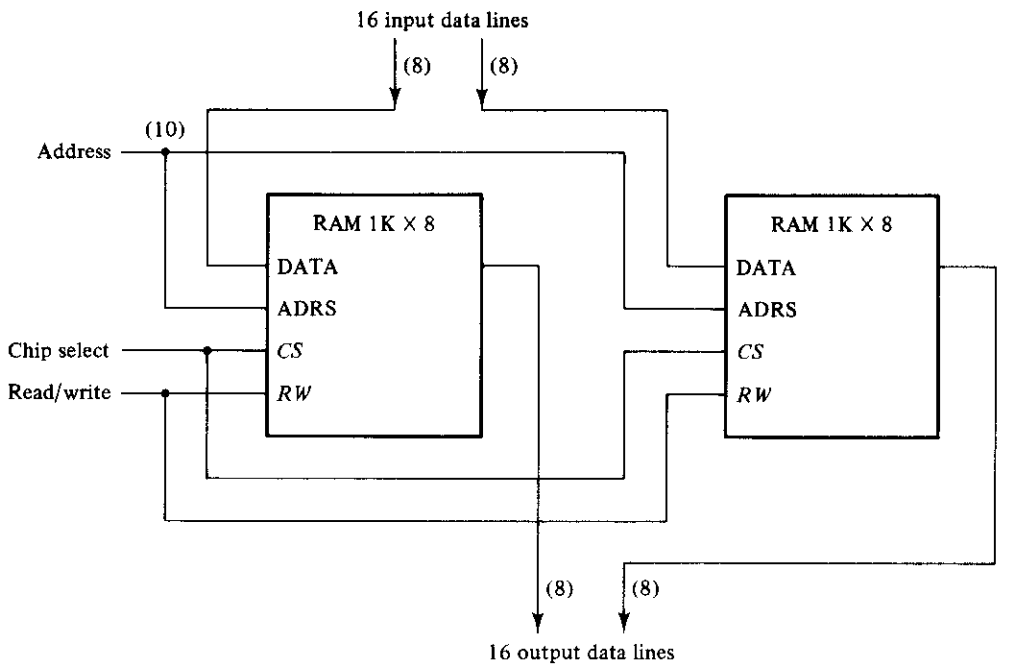


FIGURE 7-30
Block diagram of a 1K × 16 RAM

select the individual chips from the additional address bits of the composite memory.

To reduce the number of pins in the package, many RAM integrated circuits provide common terminals for the input data and output data. The common terminals are said to be *bidirectional,* which means that for the read operation, they act as outputs, and for the write operation, they act as inputs.

7-9 ERROR-CORRECTING CODE

The complexity level of a memory array may cause occasional errors in storing and retrieving the binary information. The reliability of a memory unit may be improved by employing error-detecting and correcting codes. The most common error-detection scheme is the parity bit. (See Section 4-9.) A parity bit is generated and stored along with the data word in memory. The parity of the word is checked after reading it from memory. The data word is accepted if the parity sense is correct. If the parity checked results in an inversion, an error is detected, but it cannot be corrected.

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

Hamming Code

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

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

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

The four parity bits, P_1, P_2, P_4, and P_8, are in positions 1, 2, 4, and 8, respectively. The eight bits of the data word are in the remaining positions. Each parity bit is calculated as follows:

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

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

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

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	1	1	1	0	0	1	0	1	0	0

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

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

A 0 check bit designates an even parity over the checked bits and a 1 designates an odd parity. Since the bits were stored with even parity, the result, $C = C_8C_4C_2C_1 = 0000$, indicates that no error has occurred. However, if $C \neq 0$, then the 4-bit binary number formed by the check bits gives the position of the erroneous bit. For example, consider the following three cases:

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	1	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	0	0	0	1	0	1	0	0	Error in bit 5

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

	C_8	C_4	C_2	C_1
For no error:	0	0	0	0
With error in bit 1:	0	0	0	1
With error in bit 5:	0	1	0	1

Thus, for no error, we have $C = 0000$; with an error in bit 1, we obtain $C = 0001$; and with an error in bit 5, we get $C = 0101$. The binary number of C, when it is not equal to 0000, gives the position of the bit in error. The error can be corrected by complementing the corresponding bit. Note that an error can occur in the data word or in one of the parity bits.

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

$$2^k - 1 \geq n + k$$

Solving for n in terms of k, we obtain

$$2^k - 1 - k \geq n$$

This relationship gives a formula for evaluating the number of data bits that can be used in conjunction with k check bits. For example, when $k = 3$, the number of data bits that can be used is $n \leq (2^3 - 1 - 3) = 4$. For $k = 4$, we have $2^4 - 1 - 4 = 11$, giving $n \leq 11$. The data word may be less than 11 bits, but must have at least 5 bits, otherwise, only 3 check bits will be needed. This justifies the use of 4 check bits for the 8 data bits in the previous example. Ranges of n for various values of k are listed in Table 7-8.

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

TABLE 7-8
Range of Data Bits for *k* Check Bits

Number of Check Bits, *k*	Range of Data Bits, *n*
3	2–4
4	5–11
5	12–26
6	27–57
7	58–120

Single-Error Correction, Double-Error Detection

The Hamming code can detect and correct only a single error. Multiple errors are not detected. By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors. If we include this additional parity bit, then the previous 12-bit coded word becomes $001110010100P_{13}$, where P_{13} is evaluated from the exclusive-OR of the other 12 bits. This produces the 13-bit word 0011100101001 (even parity). When the 13-bit word is read from memory, the check bits are evaluated and also the parity P over the entire 13 bits. If $P = 0$, the parity is correct (even parity), but if $P = 1$, then the parity over the 13 bits is incorrect (odd parity). The following four cases can occur:

If $C = 0$ and $P = 0$	No error occurred
If $C \neq 0$ and $P = 1$	A single error occurred, which can be corrected
If $C \neq 0$ and $P = 0$	A double error occurred, which is detected but cannot be corrected
If $C = 0$ and $P = 1$	An error occurred in the P_{13} bit

Note that this scheme cannot detect more than two errors.

Integrated circuits that use a modified Hamming code to generate and check parity bits for a single-error correction, double-error detection scheme are available commercially. One that uses an 8-bit data word and a 5-bit check word is IC type 74637. Other integrated circuits are available for data words of 16 and 32 bits. These circuits can be used in conjunction with a memory unit to correct a single error or detect double errors during the write and read operations.

REFERENCES

1. MANO. M. M., *Computer System Architecture*, 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1982.
2. PEATMAN, J. B., *Digital Hardware Design*. New York: McGraw-Hill, 1980.
3. BLAKESLEE, T. R., *Digital Design With Standard MSI And LSI*, 2nd Ed. New York: John Wiley, 1979.
4. FLETCHER, W. I., *An Engineering Approach to Digital Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
5. SANDIGE, R. S., *Digital Concepts Using Standard Integrated Circuits*. New York: McGraw-Hill, 1978.
6. SHIVA, S. G., *Introduction to Logic Design*. Glenview, IL: Scott, Foresman, 1988.
7. ROTH, C. H., *Fundamentals of Logic Design*, 3rd Ed. New York: West, 1985.
8. BOOTH, T. L., *Introduction to Computer Engineering*, 3rd Ed. New York: John Wiley, 1984
9. *The TTL Logic Data Book*. Dallas: Texas Instruments, 1988.
10. *LSI Logic Data Book*. Dallas: Texas Instruments, 1986.
11. *Memory Components Handbook*. Santa Clara, CA: Intel, 1986.

12. HAMMING, R. W., "Error Detecting and Error Correcting Codes." *Bell Syst. Tech. J.*, **29** (1950) 147–160.

13. LIN, S., and D. J. COSTELLO, JR., *Error Control Coding*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

PROBLEMS

7-1 Include a 2-input NAND gate with the register of Fig. 7-1 and connect the gate output to the *CP* inputs of all the flip-flops. One input of the NAND gate receives the clock pulses from the clock-pulse generator. The other input of the NAND gate provides a parallel-load control. Explain the operation of the modified register.

7-2 Change the asynchronous-clear circuit to a synchronous-clear circuit in the register of Fig. 7-2. The modified register will have a parallel-load capability and a synchronous-clear capability, but no asynchronous-clear circuit. The register is cleared synchronously when the clock pulse in the *CP* input goes through a negative transition provided $R = 1$ and $S = 0$ in all the flip-flops.

7-3 Repeat Problem 7-2 for the register of Fig. 7-3. Here the circuit will be cleared synchronously when the *CP* input goes through a negative transition while the *D* inputs of all flip-flops are equal to 0.

7-4 Design a sequential circuit whose state diagram is given in Fig. 6-31 using a 3-bit register and a 16 × 4 ROM.

7-5 The content of a 4-bit register is initially 1101. The register is shifted six times to the right with the serial input being 101101. What is the content of the register after each shift?

7-6 What is the difference between a serial and parallel transfer? Explain how to convert serial data to parallel and parallel data to serial. What type of register is needed?

7-7 The 4-bit bidirectional shift register with parallel load shown in Fig. 7-9 is enclosed within one IC package.

(a) Draw a block diagram of the IC showing all inputs and outputs. Include two pins for the power supply.

(b) Draw a block diagram using two ICs to produce an 8-bit bidirectional shift register with parallel load.

7-8 Design a shift register with parallel load that operates according to the following function table:

Shift	Load	Register Operation
0	0	No change
0	1	Load parallel data
1	*X*	Shift right

7-9 Draw the logic diagram of a 4-bit register with four *D* flip-flops and four 4 × 1 multiplexers with mode-selection inputs s_1 and s_0. The register operates according to the following function table:

s_1	s_0	Register Operation
0	0	No change
0	1	Complement the four outputs
1	0	Clear register to 0 (synchronous with the clock)
1	1	Load parallel data

7-10 The serial adder of Fig. 7-10 uses two 4-bit registers. Register *A* holds the binary number 0101 and register *B* holds 0111. The carry flip-flop is initially reset to 0. List the binary values in register *A* and the carry flip-flop after each shift.

7-11 What changes are needed in Fig. 7-11 to convert it to a serial subtractor that subtracts the content of register *B* from the content of register *A*?

7-12 It was stated in Section 1-5 that the 2's complement of a binary number can be formed by leaving all least significant 0's and the first 1 unchanged and complementing all other higher significant bits. Design a serial 2's complementer using this procedure. The circuit needs a shift register to store the binary number and an *RS* flip-flop to be set when the first least significant 1 occurs. An exclusive-OR gate can be used to transfer the unchanged bits ($x \oplus 0 = x$) or complement the bits ($x \oplus 1 = x'$).

7-13 Draw the logic diagram of a 4-bit binary ripple counter using flip-flops that trigger on the positive-edge transition.

7-14 Draw the logic diagram of a 4-bit binary ripple down-counter using the following:
(a) Flip-flops that trigger on the positive-edge transition of the clock.
(b) Flip-flops that trigger on the negative-edge transition of the clock.

7-15 Construct a BCD ripple counter using a 4-bit binary ripple counter that can be cleared asynchronously (similar to the clear input in Fig. 7-2) and an external NAND gate.

7-16 A flip-flop has a 10-nanosecond delay from the time its *CP* input goes from 1 to 0 to the time the output is complemented. What is the maximum delay in a 10-bit binary ripple counter that uses these flip-flops? What is the maximum frequency the counter can operate reliably?

7-17 How many flip-flops will be complemented in a 10-bit binary ripple counter to reach the next count after the following count:
(a) 1001100111;
(b) 0011111111.

7-18 Determine the next state for each of the six unused states in the BCD ripple counter shown in Fig. 7-14. Determine whether the counter is self-correcting.

7-19 Design a 4-bit binary ripple counter with *D* flip-flops.

7-20 Design a 4-bit binary synchronous counter with *D* flip-flops.

7-21 Modify the counter of Fig. 7-18 so that when both the up and down control inputs are equal to 1, the counter does not change state, but remains in the same count.

7-22 Verify the flip-flop input functions of the synchronous BCD counter specified in Table 7-5. Draw the logic diagram of the BCD counter and include a count-enable control input.

7-23 Design a synchronous BCD counter with *JK* flip-flops.

7-24 Show the connections between four IC binary counters with parallel load (Fig. 7-19) to produce a 16-bit binary counter with parallel load. Use a block diagram for each IC.

7-25 Construct a BCD counter using the circuit specified in Fig. 7-19 and an AND gate.

7-26 Construct a mod-12 counter using the circuit of Fig. 7-19. Give four alternatives.

7-27 Using two circuits of the type shown in Fig. 7-19, construct a binary counter that counts from 0 through binary 64.

7-28 Using a start signal as in Fig. 7-21, construct a word-time control that stays on for a period of 16 clock pulses.

7-29 Add four 2-input AND gates to the circuit of Fig. 7-22(b). One input in each gate is connected to one output of the decoder. The other input in each gate is connected to the clock. Label the outputs of the AND gate as P_0, P_1, P_2, and P_3. Show the timing diagram of the four P outputs.

7-30 Show the circuit and the timing diagram for generating six repeated timing signals, T_0 through T_5.

7-31 Complete the design of the Johnson counter of Fig. 7-23 showing the outputs of the eight timing signals using eight AND gates.

7-32 Construct a Johnson counter for ten timing signals.

7-33 The following memory units are specified by the number of words times the number of bits per word. How many address lines and input–output data lines are needed in each case?
(a) 2K × 16;
(b) 64K × 8;
(c) 16M × 32;
(d) 96K × 12.

7-34 Word number 535 in the memory shown in Fig. 7-25 contains the binary equivalent of 2209. List the 10-bit address and the 16-bit memory content of the word.

7-35 (a) How many 128 × 8 RAM chips are needed to provide a memory capacity of 2048 bytes?
(b) How many lines of the address must be used to access 2048 bytes? How many of these lines are connected to the address inputs of all chips?
(c) How many lines must be decoded for the chip-select inputs? Specify the size of the decoder.

7-36 A computer uses RAM chips of 1024 × 1 capacity.
(a) How many chips are needed and how should their address lines be connected to provide a memory capacity of 1024 bytes?
(b) How many chips are needed to provide a memory capacity of 16K bytes? Explain in words how the chips are to be connected.

7-37 An integrated-circuit RAM chip has a capacity of 1024 words of 8 bits each (1K × 8).
(a) How many address and data lines are there in the chip?
(b) How many chips are needed to construct a 16K × 16 RAM?
(c) How many address and data lines are there in the 16K × 16 RAM?
(d) What size decoder is needed to construct the 16K × 16 memory from the 1K × 8 chips? What are the inputs to the decoder and where are its outputs connected?

7-38 Given the 8-bit data word 01011011, generate the 13-bit composite word for the Hamming code that corrects single errors and detects double errors.

7-39 Given the 11-bit data word 11001001010, generate the 15-bit Hamming-code word.

7-40 A 12-bit Hamming-code word containing 8 bits of data and 4 parity bits is read from memory. What was the original 8-bit data word that was written into memory if the 12-bit word read out is as follows:
(a) 000011101010
(b) 101110000110
(c) 101111110100

7-41 How many parity check bits must be included with the data word to achieve single error-correction and double-error detection when the data word contains: (a) 16 bits; (b) 32 bits; (c) 48 bits.

7-42 It is necessary to formulate the Hamming code for four data bits, D_3, D_5, D_6, and D_7, together with three parity bits, P_1, P_2, and P_4.
(a) Evaluate the 7-bit composite code word for the data word 0010.
(b) Evaluate three check bits, C_4, C_2, and C_1, assuming no error.
(c) Assume an error in bit D_5 during writing into memory. Show how the error in the bit is detected and corrected.
(d) Add parity bit P_8 to include a double-error detection in the code. Assume that errors occurred in bits P_2 and D_5. Show how the double error is detected.