symbolizes a read operation from a memory register whose address is given by $A3$. The binary information coming out of memory is transferred to register $B0$. Again, this statement implies the required selection inputs for the address multiplexer and the selection variables for the destination decoder.

8.3 Arithmetic, Logic, and Shift Microoperations

The interregister-transfer microoperations do not change the information content when the binary information moves from the source register to the destination register. All other microoperations change the information content during the transfer. Among all possible operations that can exist in a digital system, there is a basic set from which all other operations can be obtained. In this section, we define a set of basic microoperations, their symbolic notation, and the digital hardware that implements them. Other microoperations with appropriate symbols can be defined if necessary to suit a particular application.

8.3.1 Arithmetic Microoperations

The basic arithmetic microoperations are add, subtract, complement, and shift. Arithmetic shifts are explained in Section 8-7 in conjunction with the type of binary data representation. All other arithmetic operations can be obtained from a variation or a sequence of these basic microoperations.

The arithmetic microoperation defined by the statement:

$$F \leftarrow A + B$$

specifies an *add* operation. It states that the contents of register A are to be added to the contents of register B, and the sum transferred to register F. To implement this statement, we require three registers, A, B, and F, and the digital function that performs the addition operation, such as a parallel adder. The other basic arithmetic operations are listed in Table 8-2. Arithmetic subtraction implies the availability of a binary parallel subtractor composed of full-subtractor circuits connected in cascade. Subtraction is most often implemented through complementation and addition as specified by the statement:

$$F \leftarrow A + \overline{B} + 1$$

Table 8-2 Arithmetic microoperations

Symbolic designation	Description
$F \leftarrow A + B$	Contents of A plus B transferred to F
$F \leftarrow A - B$	Contents of A minus B transferred to F
$B \leftarrow \overline{B}$	Complement register B (1's complement)
$B \leftarrow \overline{B} + 1$	Form the 2's complement of the contents of register B
$F \leftarrow A + \overline{B} + 1$	A plus the 2's complement of B transferred to F
$A \leftarrow A + 1$	Increment the contents of A by 1 (count up)
$A \leftarrow A - 1$	Decrement the contents of A by 1 (count down)

8.3.2 Logic Microoperations

Logic microoperations specify binary operations for a string of bits stored in registers. These operations consider each bit in the registers separately and treat it as a binary variable. As an illustration, the exclusive-OR microoperation is symbolized by the statement:

$$F \leftarrow A \oplus B$$

It specifies a logic operation that considers each pair of bits in the registers as binary variables. If the content of register A is 1010 and that of register B 1100, the information transferred to register F is 0110:

1010	content of A
1100	content of B
0110	content of $F \leftarrow A \oplus B$

There are 16 different possible logic operations that can be performed with two binary variables. These logic operations are listed in Table 2-6. All 16 logic operations can be expressed ill terms of the AND, OR, and complement operations. Special symbols will be adopted for these three microoperations to distinguish them from the corresponding symbols used to express Boolean functions. The symbol $\vee$ will be used to denote an OR microoperation and the symbol $\wedge$ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the letter (or letters) that denotes the register. By using these symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. The symbols for the four logic microoperations are summarized in Table 8-3. The last two symbols are for the shift microoperations discussed below.

A more important reason for adopting a special symbol for the OR microoperation is to differentiate the symbol +, when used as an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When this symbol occurs in a microoperation, it denotes an arithmetic plus. When it occurs in a control (or Boolean) function, it denotes a logic OR operation. For example, in the statement:

$$T_1 + T_2: A \leftarrow A + B,\ C \leftarrow D \vee F$$

Table 8-3 Logic and shift microoperations

Symbolic designation	Description
$A \leftarrow \overline{A}$	Complement all bits of register A
$F \leftarrow A \vee B$	Logic OR microoperation
$F \leftarrow A \wedge B$	Logic AND microoperation
$F \leftarrow A \oplus B$	Logic exclusive-OR microoperation
$A \leftarrow \text{shl}\ A$	Shift-left register A
$A \leftarrow \text{shr}\ A$	Shift-right register A

may be obtained from the 1's complement plus 1 (i.e., 1 111111 + 1) provided the end carry is discarded.

The range of binary integer numbers that can be accommodated in a register of $n = k + 1$ bits is $\pm (2^k - 1)$, where k bits are reserved for the number and one bit for the sign. A register with 8 bits can store binary numbers in the range of $\pm (2^7 - 1) = \pm 127$. However, since the sign-2's-complement representation has only one zero, it should accommodate one more number than the other two representations. Consider the representation of the largest and smallest numbers:

	sign-1's complement	sign-2's complement
+ 126 = 0 1111110	− 126 = 1 0000001	1 0000010
+ 127 = 0 1111111	− 127 = 1 0000000	1 0000001
+ 128 (impossible)	− 128 (impossible)	1 0000000

In the sign-2's-complement representation, it is possible to represent −128 with eight bits. In general, the sign-2's-complement representation can accommodate numbers in the range $+ (2^k - 1)$ to -2^k, where $k = n - 1$ and n is the number of bits in the register.

8.5.4 Arithmetic Subtraction

Subtraction of two signed binary numbers when negative numbers are in the 2's-complement form is very simple and can be stated as follows: *Take the* 2's *complement of the subtrahend* (*including the sign bit*) *and add it to the minuend* (*including sign bit*). This procedure uses the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationships (B is the subtrahend):

$$(\pm A) - (-B) = (\pm A) + (+B)$$
$$(\pm A) - (+B) = (\pm A) + (-B)$$

But changing a positive number to a negative number is easily done by taking its 2's complement (including the sign bit). The reverse is also true because the complement of the complement restores the number to its original value.

The subtraction with 1's-complement numbers is similar except for the end-around carry. Subtraction with sign-magnitude requires that only the sign bit of the subtrahend be complemented. Addition and subtraction of binary numbers in sign-magnitude representation is demonstrated in Section 10-3.

Because of the simple procedure for adding and subtracting binary numbers when negative numbers are in sign-2's-complement form, most computers adopt this representation over the more familiar sign-magnitude form. The reason 2's complement is usually chosen over the 1's complement is to avoid the end-around carry and the occurrence of a negative zero.

8.6 Overflow

When two numbers of n digits each are added and the sum occupies $n + 1$digits, we say that an *overflow* occurs. This is true for binary numbers or decimal numbers whether signed or unsigned. When one performs the addition with paper and pencil, an overflow is not a problem, since we

are not limited by the width of the page to write down the sum. An overflow is a problem in a digital computer because the lengths of all registers, including memory registers, are of finite length. A result of $n + 1$ bits cannot be accommodated in a register of standard length n. For this reason, many computers check for the occurrence of an overflow, and when it occurs, they set an overflow flip-flop for the user to check.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result (positive or negative) which is smaller than the larger of the two original numbers. An overflow may occur if the two numbers are added and both are positive or both are negative. When two numbers in sign-magnitude representation are added, an overflow can be easily detected from the carry out of the number bits. When two numbers in sign-2's-complement representation are added, the sign bit is treated as part of the number and the end carry does not necessarily indicate an overflow.

The algorithm for adding two numbers in sign-2's-complement representation, as previously stated, gives an incorrect result when an overflow occurs. This arises because an overflow of the number bits always changes the sign of the result and gives an erroneous n-bit answer. To see how this happens, consider the following example. Two signed binary numbers, 35 and 40, are stored in two 7-bit registers. The maximum capacity of the register is $(2^6 - 1) = 63$ and the minimum capacity is $-2^6 = -64$. Since the sum of the numbers is 75, it exceeds the capacity of the register. This is true if the numbers are both positive or both negative. The operations in binary are shown below together with the last two carries of the addition:

carries: 0 1		carries: 1 0	
+ 35	0 100011	− 35	1 011101
+ 40	0 101000	− 40	1 011000
+ 75	1 001011	− 75	0 110101

In either case, we see that the 7-bit result that should have been positive is negative, and vice versa. Obviously, the binary answer is incorrect and the algorithm for adding binary numbers represented in 2's complement as stated previously fails to give correct results when an overflow occurs. Note that if the carry out of the sign-bit position is taken as the sign for the result, then the 8-bit answer so obtained will be correct.

An overflow condition can be detected by observing the carry *into* the sign-bit position and the carry *out* of the sign-bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the example above where the two carries are explicitly shown. The reader can try a few examples of numbers that do not produce an overflow to see that these two carries will always come out to be either both 0's or both 1's. If the two carries are applied to an exclusive-OR gate, an overflow would be detected when the output of the gate is 1.

The addition of two signed binary numbers when negative numbers are represented in sign-2's-complement form is implemented with digital functions as shown in Fig. 8-10. Register A holds the augend, with the sign bit in position A_n. Register B holds the addend, with the sign bit in B_n. The two numbers are added by means of an n-bit parallel adder. The full-adder (FA) circuit in stage n (the sign bits) is shown explicitly. The carry going into this full-adder is C_n. The carry

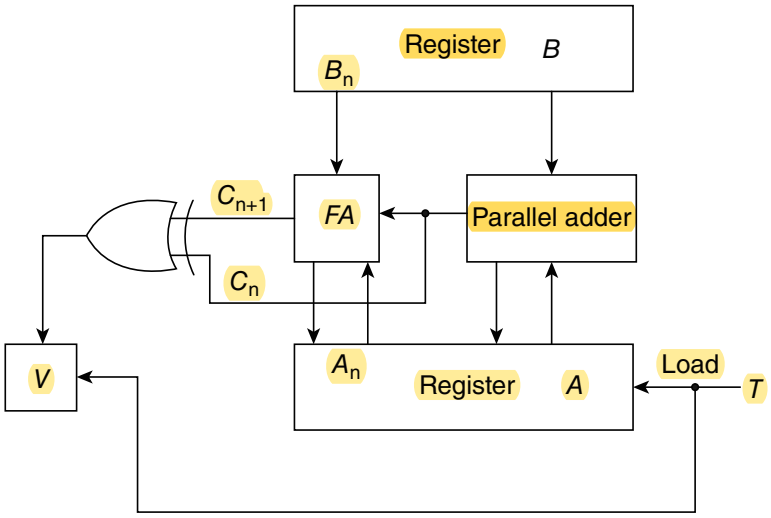

Figure 8.10 Addition of sign-2's-complement numbers

out of the full-adder is C_{n+1}. The exclusive-OR of these two carries is applied to an overflow flip-flop *V*. If, after the addition, $V = 0$, then the sum loaded into *A* is correct. If $V = 1$, there is an overflow and the *n*-bit sum is incorrect. The circuit shown in Fig. 8-10 can be specified by the following statement:

$$T: A \leftarrow A + B, \qquad V \leftarrow C_n + C_{n+1}$$

The variables in the statement are defined in Fig. 8-10. Note that variables C_n and C_{n+1} do not represent registers; they represent output carries from a parallel adder.

8.7 Arithmetic Shifts

An arithmetic shift is a microoperation that shifts a *signed* binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign unchanged because the sign of the number remains the same when it is multiplied or divided by 2.

The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. Figure 8-11 shows a register of *n* bits. Bit A_n in the leftmost position holds the sign bit and is designated by $A(S)$. The number bits are stored in the portion of the register designated by $A(N)$. A_1 refers to the least significant bit, A_{n-1} refers to the most significant position in the number bits, and *A* refers to the entire register.

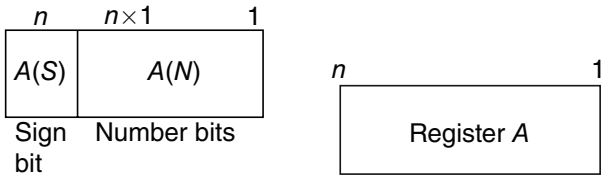

Figure 8.11 Defining register *A* for arithmetic shifts

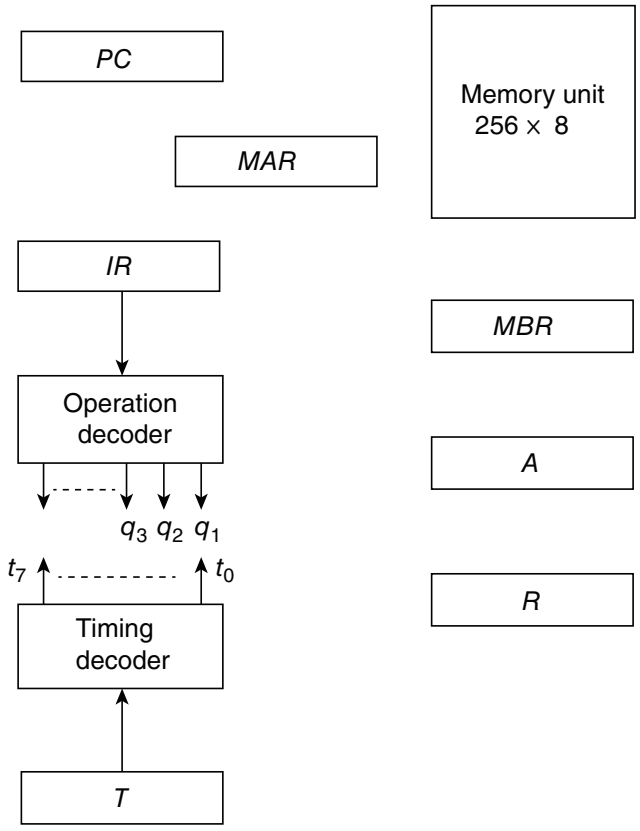

Figure 8.14 Block diagram of a simple computer

to supply eight timing variables, t_0 through t_7 (see Section 7-6). This counter is incremented with every clock pulse, but it can be cleared at any time to start a new sequence from t_0.

The *PC* goes through a step-by-step counting sequence and causes the computer to read successive instructions previously stored in memory. The *PC* always holds the address of the next instruction in memory. To read an instruction, the contents of *PC* are transferred into *MAR* and a memory-read cycle is initiated. The *PC* is then incremented by 1 so it holds the next address in the sequence of instructions. An operation code read from memory into *MBR* is then transferred into *IR*. If the memory-address part of an instruction is read into *MBR,* this address is

Table 8-4 List of registers for the simple computer

Symbol	Number of bits	Name of register	Function
MAR	8	Memory address register	Holds address for memory
MBR	8	Memory buffer register	Holds contents of memory word
A	8	*A* register	Processor register
R	8	*R* register	Processor register
PC	8	Program counter	Holds address of instruction
IR	8	Instruction register	Holds current operation code
T	3	Timing counter	Sequence generator

9.2.3 Accumulator Register

Some processor units separate one register from all others and call it an *accumulator* register, abbreviated *AC* or *A* register. The name of this register is derived from the arithmetic addition process encountered in digital computers. The process of adding many numbers is carried out by initially storing these numbers in other processor registers or in the memory unit of the computer and clearing the accumulator to 0. The numbers are then added to the accumulator one at a time, in consecutive order. The first number is added to 0, and the sum transferred to the accumulator. The second number is added to the contents of the accumulator, and the newly formed sum replaces its previous value. This process is continued until all numbers are added and the total sum is formed. Thus, the register "accumulates" the sum in a step-by-step manner by performing sequential additions between a new number and the previously accumulated sum.

The accumulator register in a processor unit is a multipurpose register capable of performing not only the add microoperation, but many other microoperations as well. In fact, the gates associated with an accumulator register provide all the digital functions found in an ALU.

Figure 9-4 shows the block diagram of a processor unit that employs an accumulator register. The *A* register is distinguished from all other processor registers. In some cases the entire processor unit is just the accumulator register and its associated ALU. The register itself can function as a shift register to provide the shift microoperations. Input *B* supplies one external source information. This information may come from other processor registers or directly from the main memory of the computer. The *A* register supplies the other source information to the

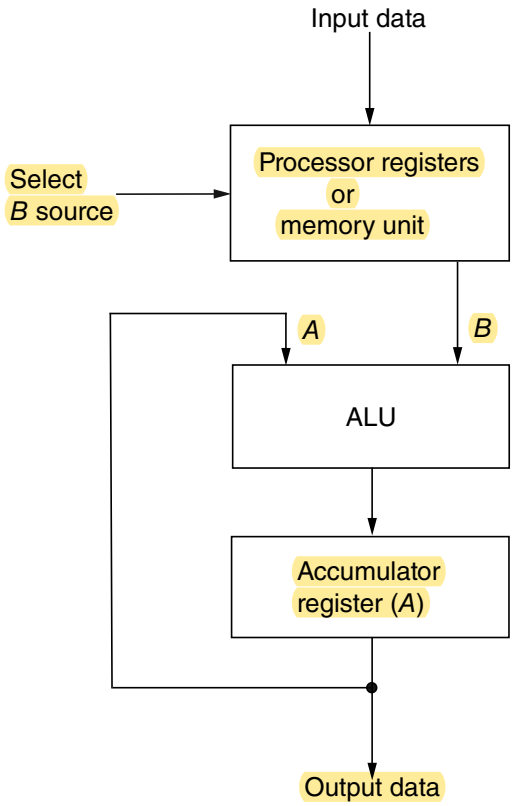

Figure 9-4 Processor with an accumulator register

ALU at input *A*. The result of an operation is transferred back to the *A* register and replaces its previous content. The output from the *A* register may go to an external destination or into the input terminals of other processor registers or memory unit.

To form the sum of two numbers stored in processor registers, it is necessary to add them in the *A* register using the following sequence of microoperations:

T_1:	$A \leftarrow 0$	clear *A*
T_2:	$A \leftarrow A + R1$	transfer *R*1 to *A*
T_3:	$A \leftarrow A + R2$	add *R*2 to *A*

Register *A* is first cleared. The first number in *R*1 is transferred into the *A* register by adding it to the present zero content of *A*. The second number in *R*2 is then added to the present value of *A*. The sum formed in *A* may be used for other computations or may be transferred to a required destination.

9.3 Arithmetic Logic Unit

An arithmetic logic unit (ALU) is a multioperation, combinational-logic digital function. It can perform a set of basic arithmetic operations and a set of logic operations. The ALU has a number of selection lines to select a particular operation in the unit. The selection lines are decoded within the ALU so that *k* selection variables can specify up to 2^k distinct operations.

Figure 9-5 shows the block diagram of a 4-bit ALU. The four data inputs from *A* are combined with the four inputs from *B* to generate an operation at the *F* outputs. The mode-select input s_2 distinguishes between arithmetic and logic operations. The two function-select inputs s_1 and s_0 specify the particular arithmetic or logic operation to be generated. With three selection variables, it is possible to specify four arithmetic operations (with s_2 in one state) and four logic operations (with s_2 in the other state). The input and output carries have meaning only during an arithmetic operation.

The input carry in the least significant position of an ALU is quite often used as a fourth selection variable that can double the number of arithmetic operations. In this way, it is possible to generate four more operations, for a total of eight arithmetic operations.

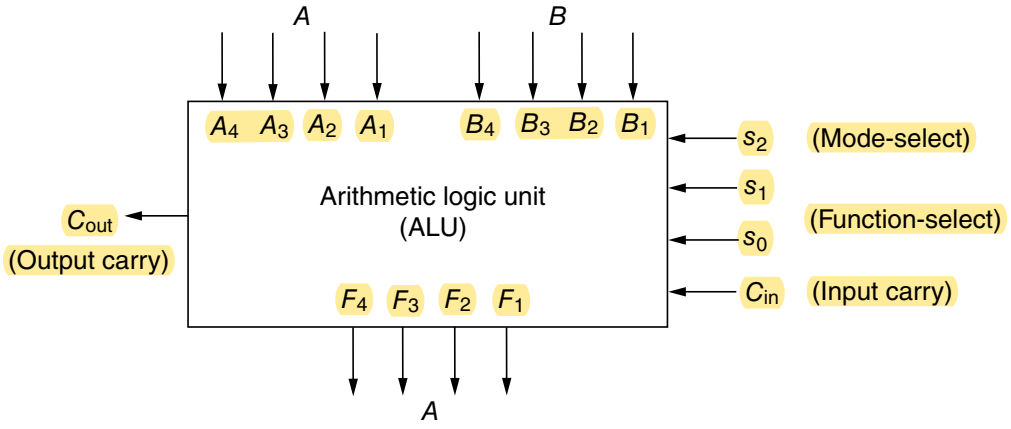

Figure 9-5 Block diagram of a 4-bit ALU

A flowchart is a diagram that consists of blocks connected by directed lines. Within the blocks, we specify the procedural steps for implementing the algorithm. The directed lines between blocks designate the path to be taken from one procedural step to the next. Two major types of blocks are used: A rectangular block designates a *function* block within which the microoperations are listed. A diamond-shaped block is a *decision* block within which is listed a given status condition. A decision block has two or more alternate paths, and the path that is taken depends on the value of the status condition specified within the block.

A flowchart is very similar to a state diagram. Each function block in the flowchart is equivalent to a state in the state diagram. The decision block in the flowchart is equivalent to the binary information written along the directed lines that connect two states in a state diagram. As a consequence, it is sometimes convenient to express an algorithm by means of a flowchart from which the control state diagram may be readily derived.

In this chapter, we first present four possible configurations for a control unit. The various configurations are presented in block diagram form to emphasize the differences in organization. We then demonstrate the various procedures available for control logic design by going through specific examples.

The design of control logic cannot be separated from the algorithmic development necessary for solving a design problem. Moreover, the control logic is directly related to the data processor part of the system that it controls. As a consequence, the examples presented in this chapter start with the development of an algorithm for implementing the given problem. The data-processing part of the system is then derived from the stated algorithm. Only after this is done can we proceed to show the design of the control that sequences the data processor according to the steps specified by the algorithm.

10.2 Control Organization

Short note

Once a control sequence has been established, the sequential system that implements the control operations must be designed. Since the control is a sequential circuit, it can be designed by a sequential logic procedure as outlined in Chapter 6.

However, in most cases this method is impractical because of the large number of states that the control circuit may have. Design methods that use state and excitation tables can be used in theory, but in practice they are cumbersome and difficult to manage. Moreover, the control circuit obtained by this method usually requires an excessive number of flip-flops and gates, which implies the use of SSI circuits. This type of implementation is inefficient with respect to the number of IC packages used and the number of wires that must be interconnected. One major goal of control logic design should be the development of a circuit that implements the desired control sequence in a logical and straightforward manner. The attempt to minimize the number of circuits would tend to produce an irregular network which would make it difficult for anyone but the designer to recognize the sequence of events that the control undergoes. As a consequence, it may be difficult to service and maintain the equipment when it is in operation.

Because of the reasons cited above, experienced logic designers use specialized methods for control logic design which may be considered an extension of the classical sequential-logic method combined with the register-transfer method. In this section, we consider four methods of control organization:

1. One flip-flop per state method.
2. Sequence register and decoder method.

Only brief no need to study diagram

3. PLA control.
4. Microprogram control.

The first two methods result in a circuit that must use SSI and MSI circuits for the implementation. The various circuits are interconnected by wires to form the control network. A control unit implemented with SSI and MSI devices is said to be a hard-wired control. If any alterations or modifications are needed, the circuits must be rewired to fulfill the new requirements. This is in contrast to the PLA or microprogram control which uses an LSI device such as a programmable logic array or a read-only memory. Any alterations or modifications in a microprogram control can be easily achieved without wiring changes by removing the ROM from its socket and inserting another ROM programmed to fulfill the new specifications.

We shall now explain each method in general terms. The subsequent sections of this chapter deal with specific examples that demonstrate the detailed design of control units by each of the four methods.

10.2.1 One Flip-Flop per State Method

This method uses one flip-flop per state in the control sequential circuit. Only one flip-flop is set at any particular time: all others are cleared. A single bit is made to propagate from one flip-flop to the other under the control of decision logic. In such an array, each flip-flop represents a state and is activated only when the control bit is transferred to it.

It is obvious that this method does not use a minimum number of flip-flops for the sequential circuit. In fact, it uses a maximum number of flip-flops. For example, a sequential circuit with 12 states requires a minimum of four flip-flops because $2^3 < 12 < 2^4$. Yet by this method, the control circuit uses 12 flip-flops, one for each state.

The advantage of the one flip-flop per state method is the simplicity with which it can be designed. This type of controller can be designed by inspection from the state diagram that describes the control sequence. At first glance, it may seem that this method would increase system cost since more flip-flops are used. But the method offers other advantages which may not be apparent at first. For example, it offers a savings in design effort, an increase in operational simplicity, and a potential decrease in the combinational circuits required to implement the complete sequential circuit.

Figure 10-2 shows the configuration of a four-state sequential control logic that uses four *D-type* flip-flops: one flip-flop per state T_i, $i = 0, 1,2, 3$. At any given time interval between two clock pulses, only one flip-flop is equal to 1; all others are equal to 0. The transition from the present state to the next is a function of the present T_i that is a 1 and certain input conditions. The next state is manifested when the previous flip-flop is cleared and a new one is set. Each of the flip-flop outputs is connected to the data-processing section of the digital system to initiate certain microoperations. The other control outputs shown in the diagram are a function of the *T*'s and external inputs. These outputs may also initiate microoperations.

If the control circuit does not need external inputs for its sequencing, the circuit reduces to a straight shift register with a single bit shifted from one position to the next. If the control sequence must be repeated over and over again, the control reduces to a ring counter. A *ring counter* is a shift register with the output of the last flip-flop connected to the input of the first flip-flop. In a ring counter, the single bit continuously shifts from one position to the next in a circular manner. For this reason, the one flip-flop per state method is sometimes called a *ring-counter controller.*

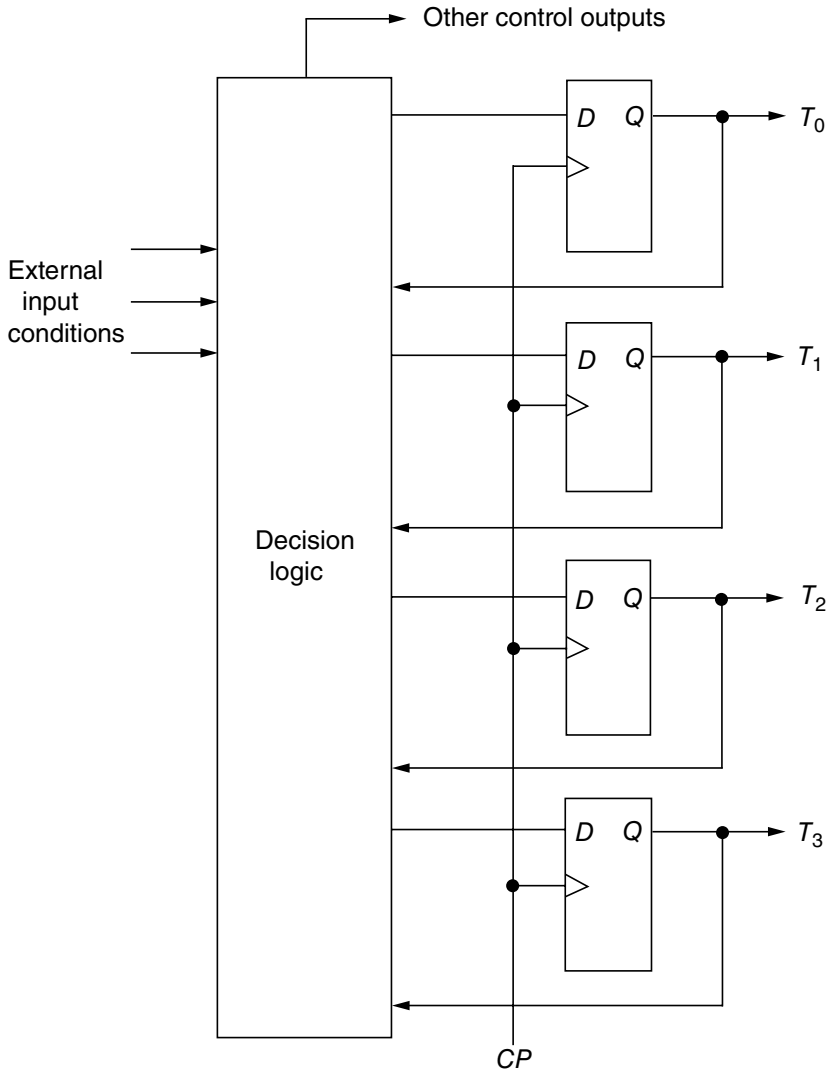

Figure 10-2 Control logic with one flip-flop per state

10.2.2 Sequence Register and Decoder Method

This method uses a register to sequence the control states. The register is decoded to provide one output for each state. For n flip-flops in the sequence register, the circuit will have 2^n states and the decoder will have 2^n outputs. For example, a 4-bit register can be in any one of 16 states. A 4×16 decoder will have 16 outputs, one for each state of the register. Both the sequence register and decoder are MSI devices.

Figure 10-3 shows the configuration of a four-state sequential control logic. The sequence register has two flip-flops and the decoder establishes separate outputs for each state in the register. The transition to the next state in the sequence register is a function of the present state and the external input conditions. Since the outputs of the decoder are available anyway, it is convenient to use them as present state variables rather than use the direct flip-flop outputs. Other outputs which are a function of the present-state and external inputs may initiate microoperations in addition to the decoder outputs.

If the control circuit of Fig. 10-3 does not need external inputs, the sequence register reduces to a counter that continuously sequences through the four states. For this reason, this method is

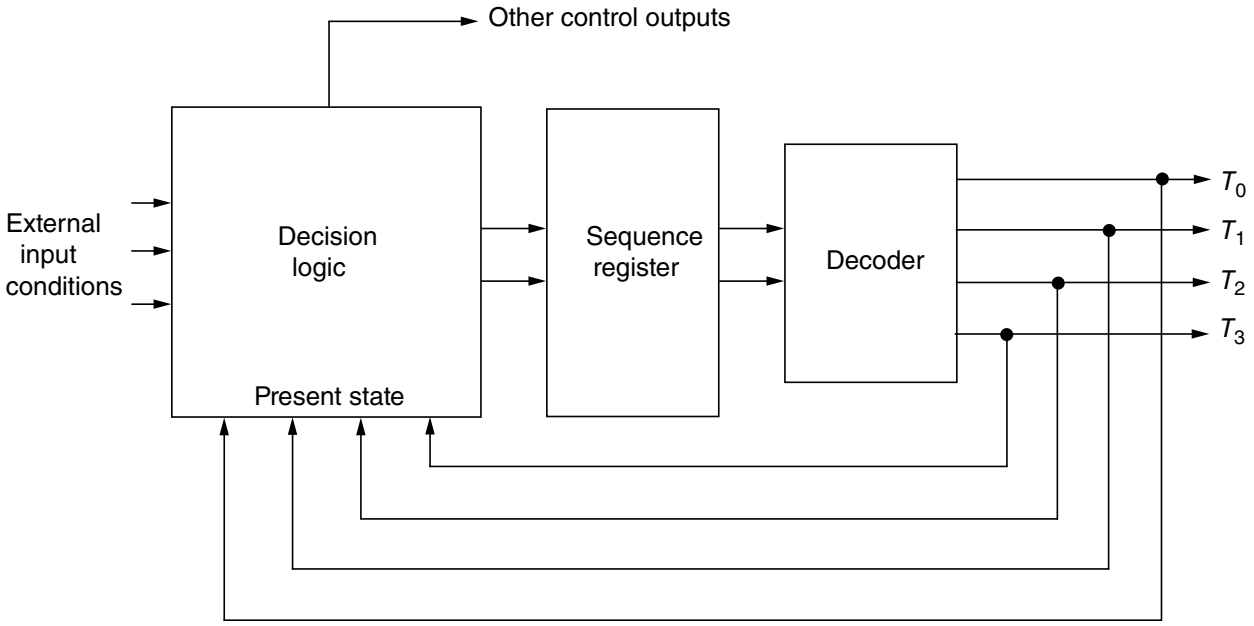

Figure 10-3 Control logic with sequence register and decoder

sometimes called a *counter-decoder* method. The counter-decoder method and the ring-counter method were explained in Chapter 7 in conjunction with Fig. 7-22.

10.2.3 PLA Control

The programmable logic array was introduced in Section 5-8. It was shown there that the PLA is an LSI device that can implement any complex combinational circuit. The PLA control is essentially similar to the sequence register and decoder method except that all combinational circuits are implemented with a PLA, including the decoder and the decision logic. By using a PLA for the combinational circuit, it is possible to reduce the number of ICs and the number of interconnection wires.

Figure 10-4 shows the configuration of a PLA controller. An external sequence register establishes the present state of the control circuit. The PLA outputs determine which microop-

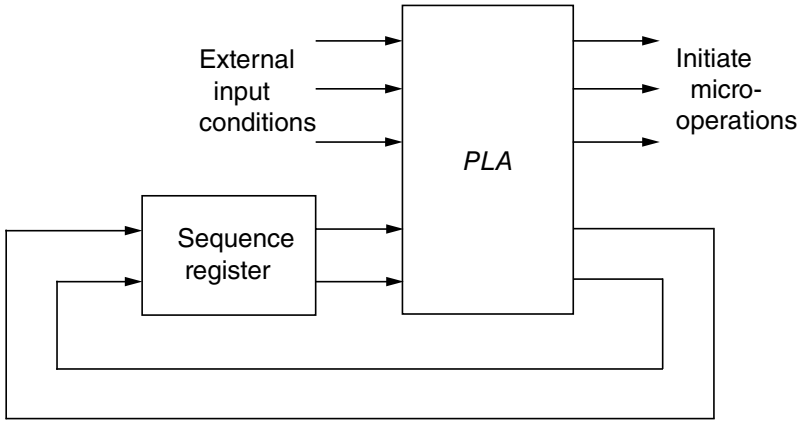

Figure 10-4 PLA control logic

erations should be initiated, depending on external input conditions and the present state of the sequence register. At the same time, other PLA outputs determine the next state of the sequence register.

The sequence register is external to the PLA if the unit implements only combinational circuits. However, some PLAs are available which include not only gates, but also flip-flops within the unit. This type of PLA can implement a sequential circuit by specifying the links that must be connected to the flip-flops in the same manner that the gate links are specified.

10.2.4 Microprogram Control

The purpose of the control unit is to initiate a series of sequential steps of microoperations. During any given time, certain operations are to be initiated while all others remain idle. Thus, the control variables at any given time can be represented by a string of 1's and 0's called a *control word.* As such, control words can be programmed to initiate the various components in the system in an organized manner. A control unit whose control variables are stored in a memory is called a *microprogrammed control unit.* Each control word of memory is called a *microinstruction,* and a sequence of microinstructions is called a *microprogram.* Since alteration of the microprogram is seldom needed, the control memory can be a ROM. The use of a microprogram involves placing all control variables in words of the ROM for use by the control unit through successive read operations. The content of the word in the ROM at a given address specifies the microoperations for the system.

A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from the computer console or from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a *writable control memory* (WCM). This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A ROM, PLA, or WCM, when used in a control unit, is referred to as a *control memory.*

Figure 10-5 illustrates the general configuration of the microprogram control unit. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the control word read from control memory. It must be realized that a ROM operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word remains on the output wires as long as the address value remains in the address register. No read signal is needed as in a random-access memory. The word out of the ROM should be transferred to a

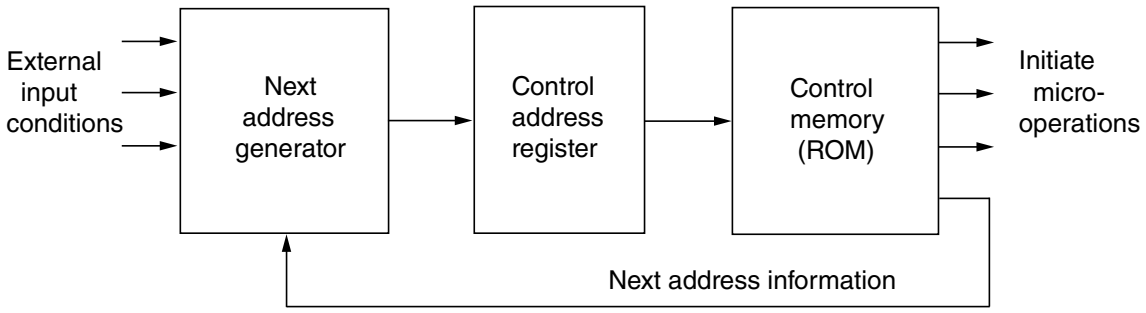

Figure 10-5 Microprogram control logic