CHAPTER FOUR

Register Transfer and Microoperations

IN THIS CHAPTER

4-1 Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

microoperation

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. Some of the digital components introduced in Chap. 2 are registers that implement microoperations. For example, a counter with parallel load is capable of performing the micro-

operations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

register transfer language

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

4-2 Register Transfer

registers

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name *MAR*. Other designations for registers are *PC* (for program counter), *IR* (for instruction register, and *R*1 (for processor register). The individual flip-flops in an n-bit register are numbered in sequence from 0 through $n - 1$, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 4-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is *PC*. The symbol *PC*(0–7) or *PC*(*L*) refers to the low-order byte and *PC*(8–15) or *PC*(*H*) to the high-order byte.

register transfer

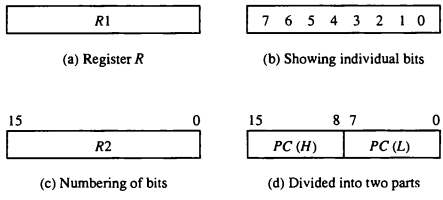
Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the content of register *R*1 into register *R*2. It designates a replacement of the content of *R*2 by the content of *R*1. By definition, the content of the source register *R*1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Nor-

Figure 4-1 Block diagram of register.



mally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an *if-then* statement.

$$If\ (P = 1)\ then\ (R2 \leftarrow R1)$$

where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a *control function*. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

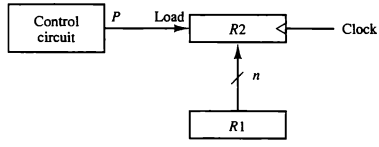
control function

$$P:\ \ R2 \leftarrow R1$$

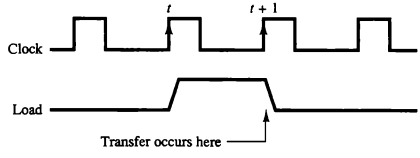
The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if $P = 1$.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4-2 shows the block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register $R2$ has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



(b) Timing diagram

in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time $t + 1$ finds the load input active and the data inputs of $R2$ are then loaded into the register in parallel. P may go back to 0 at time $t + 1$; otherwise, the transfer will occur with every clock pulse transition while P remains active.

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

The basic symbols of the register transfer notation are listed in Table 4-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T: \quad R2 \leftarrow R1, \quad R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that $T = 1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

4-3 Bus and Memory Transfers

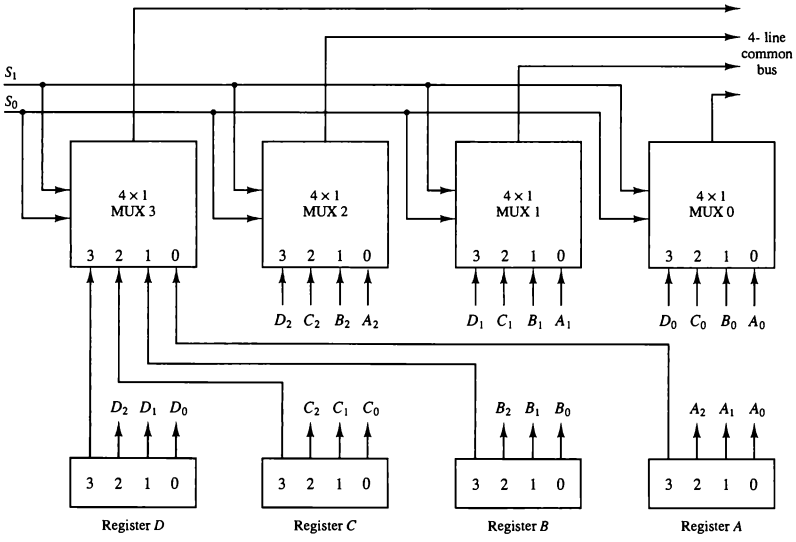
A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals

common bus

determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig. 4-3. Each register has four bits, numbered 0 through 3. The bus consists of four 4 × 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0. In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled A_1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

Figure 4-3 Bus system for four registers.



bus selection

The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if $S_1S_0 = 01$, and so on. Table 4-2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

TABLE 4-2 Function Table for Bus of Fig. 4-3

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

In general, a bus system will multiplex k registers of n bits each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be $k \times 1$ since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

$$BUS \leftarrow C, \qquad R1 \leftarrow BUS$$

The content of register C is placed on the bus, and the content of the bus is loaded into register $R1$ by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

Three-State Bus Buffers

three-state gate

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a *high-impedance state*. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

high-impedance

buffer

The graphic symbol of a three-state buffer gate is shown in Fig. 4-4. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

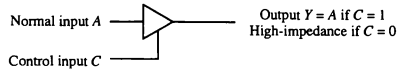
bus system

The construction of a bus system with three-state buffers is demonstrated in Fig. 4-5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Fig. 4-5 is another way of constructing a 4×1 multiplexer since the circuit can replace the multiplexer in Fig. 4-3.

To construct a common bus for four registers of n bits each using three-

Figure 4-4 Graphic symbols for three-state buffer.



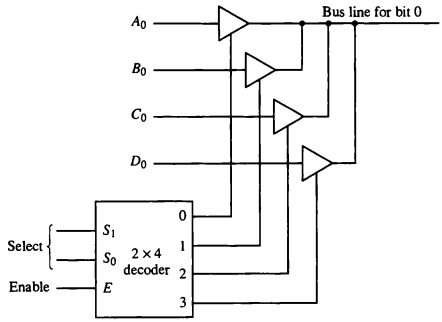


Figure 4-5 Bus line with three state-buffers.

state buffers, we need n circuits with four buffers in each as shown in Fig. 4-5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of n lines. Only one decoder is necessary to select between the four registers.

Memory Transfer

The operation of a memory unit was described in Sec. 2-7. The transfer of information from a memory word to the outside environment is called a *read* operation. The transfer of new information to be stored into the memory is called a *write* operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

memory read

$$\text{Read:} \quad DR \leftarrow M[AR]$$

This causes a transfer of information into DR from the memory word M selected by the address in AR.

memory write

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register $R1$

$$R3 \leftarrow R1 + \overline{R2} + 1$$

$\overline{R2}$ is the symbol for the 1's complement of $R2$. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of $R1$ to the 2's complement of $R2$ is equivalent to $R1 - R2$.

TABLE 4-3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

The arithmetic operations of multiply and divide are not listed in Table 4-3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift (see Chapter 10).

Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 1-17). The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full-adder circuits con-

binary adder

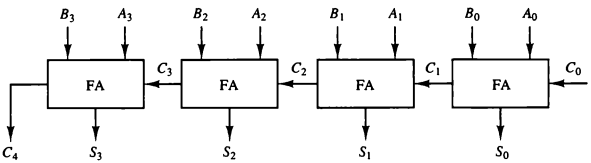


Figure 4-6 4-bit binary adder.

nected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4. The S outputs of the full-adders generate the required sum bits.

full-adder

An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from one register (such as $R1$), and the n data bits for the B inputs come from another register (such as $R2$). The sum can be transferred to a third register or to one of the source registers ($R1$ or $R2$), replacing its previous content.

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-2. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 4-7. The mode input M controls the operation. When $M = 0$ the circuit is an adder and when $M = 1$ the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When $M = 1$, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the

adder-subtractor

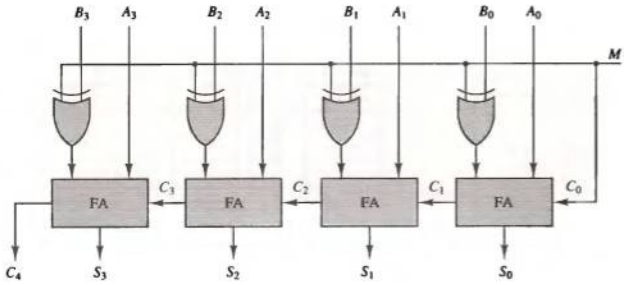


Figure 4-7 4-bit adder-subtractor.

2's complement of B. For unsigned numbers, this gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is $A - B$ provided that there is no overflow.

Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter (see Fig. 2-10). Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders (see Fig. 1-16) connected in cascade.

incrementer

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 4-8. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3. The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to 0.

The circuit of Fig. 4-8 can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

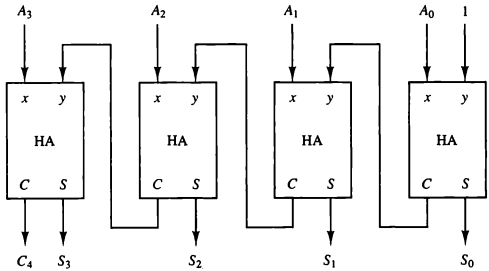


Figure 4-8 4-bit binary incrementer.

Arithmetic Circuit

arithmetic circuit

The arithmetic microoperations listed in Table 4-3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in Fig. 4-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B are connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, S_1 and S_0. The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

input carry

The output of the binary adder is calculated from the following arithmetic sum:

$$D = A + Y + C_{in}$$

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4-4.

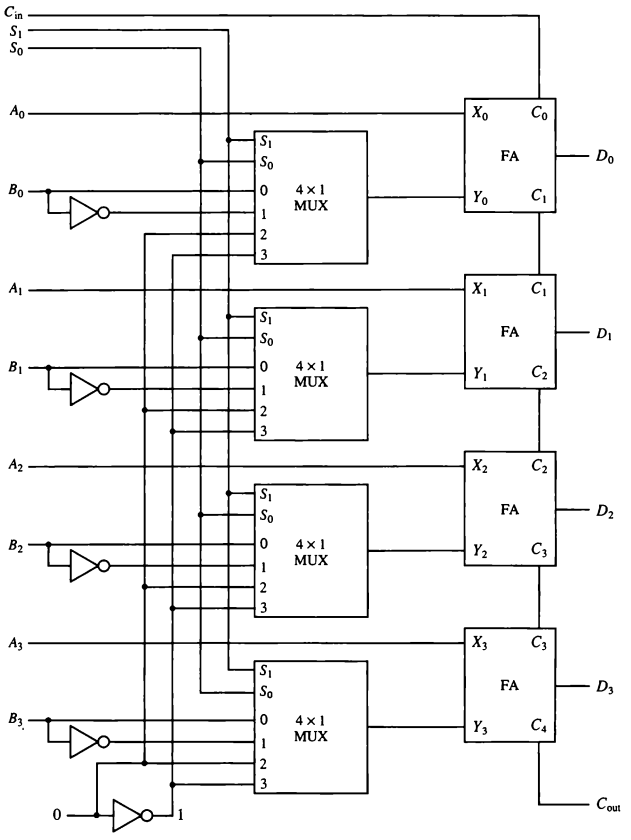


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

When $S_1S_0 = 00$, the value of B is applied to the Y inputs of the adder. If $C_{in} = 0$, the output $D = A + B$. If $C_{in} = 1$, output $D = A + B + 1$. Both cases perform the add microoperation with or without adding the input carry.

addition

When $S_1S_0 = 01$, the complement of B is applied to the Y inputs of the adder. If $C_{in} = 1$, then $D = A + \overline{B} + 1$. This produces A plus the 2's complement of B, which is equivalent to a subtraction of $A - B$. When $C_{in} = 0$, then $D = A + \overline{B}$. This is equivalent to a subtract with borrow, that is, $A - B - 1$.

subtraction

When $S_1S_0 = 10$, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes $D = A + 0 + C_{in}$. This gives $D = A$ when $C_{in} = 0$ and $D = A + 1$ when $C_{in} = 1$. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

increment

When $S_1S_0 = 11$, all 1's are inserted into the Y inputs of the adder to produce the decrement operation $D = A - 1$ when $C_{in} = 0$. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces $F = A$ + 2's complement of $1 = A - 1$. When $C_{in} = 1$, then $D = A - 1 + 1 = A$, which causes a direct transfer from input A to output D. Note that the microoperation $D = A$ is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

decrement

4-5 Logic Microoperations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers $R1$ and $R2$ is symbolized by the statement

$$P: \quad R1 \leftarrow R1 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable $P = 1$. As a numerical example, assume that each register has four bits. Let the content of $R1$ be 1010 and the content of $R2$ be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

$$\begin{array}{ll} 1010 & \text{Content of } R1 \\ \underline{1100} & \text{Content of } R2 \\ 0110 & \text{Content of } R1 \text{ after } P = 1 \end{array}$$

The content of $R1$, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in $R2$ and previous values of $R1$. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

special symbols

Special symbols will be adopted for the logic microoperations OR, AND, and complement, 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 symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol + , when used to symbolize 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 the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example, in the statement

$$P + Q: \quad R1 \leftarrow R2 + R3, \quad R4 \leftarrow R5 \vee R6$$

the + between P and Q is an OR operation between two binary variables of a control function. The + between $R2$ and $R3$ specifies an add microoperation. The OR microoperation is designated by the symbol $\vee$ between registers $R5$ and $R6$.

List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4-5. In this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4-6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B. It is important to realize that the Boolean functions listed in the first column of Table 4-6 represent a relationship between two binary variables x and y. The logic microoperations listed in the second column represent a relationship between the binary content of two registers A and B. Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Hardware Implementation

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four—AND, OR, XOR (exclusive-OR), and complement—from which all others can be derived.

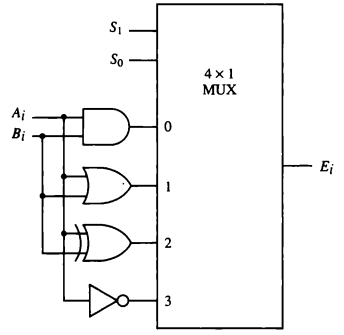
logic circuit

Figure 4-10 shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for $i = 0, 1, 2, \ldots, n - 1$. The selection variables are applied to all stages. The function table in Fig. 4-10(b) lists the logic microoperations obtained for each combination of the selection variables.

Some Applications

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated

Figure 4-10 One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

selective-set

The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

```
1010    A before
1100    B (logic operand)
1110    A after
```

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register.

selective-complement

The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

```
1010    A before
1100    B (logic operand)
0110    A after
```

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

selective-clear

The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

```
1010    A before
1100    B (logic operand)
0010    A after
```

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'. The corresponding logic microoperation is

$$A \leftarrow A \wedge \overline{B}$$

The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

1010	A before
1100	B (logic operand)
1000	A after masking

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. The two leftmost bits are left unchanged because the corresponding bits of B are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0110 1010	A before
0000 1111	B (mask)
0000 1010	A after masking

and then insert the new value:

0000 1010	A before
1001 0000	B (insert)
1001 1010	A after insertion

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

1010	A
1010	B
0000	$A \leftarrow A \oplus B$

When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

4-6 Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

logical shift

A *logical* shift is one that transfers 0 through the serial input. We will adopt the symbols *shl* and *shr* for logical shift-left and shift-right microoperations. For example:

$$R1 \leftarrow \text{shl } R1$$

$$R2 \leftarrow \text{shr } R2$$

are two microoperations that specify a 1-bit shift to the left of the content of register $R1$ and a 1-bit shift to the right of the content of register $R2$. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

circular shift

The *circular* shift (also known as a *rotate* operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table 4-7.

TABLE 4-7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

arithmetic shift

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 bit unchanged because the sign of the number remains the same

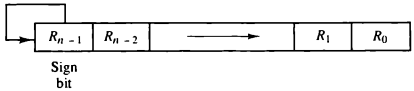


Figure 4-11 Arithmetic shift right.

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. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 4-11 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1}, and so on for the other bits in the register. The bit in R_0 is lost.

The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2}. An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

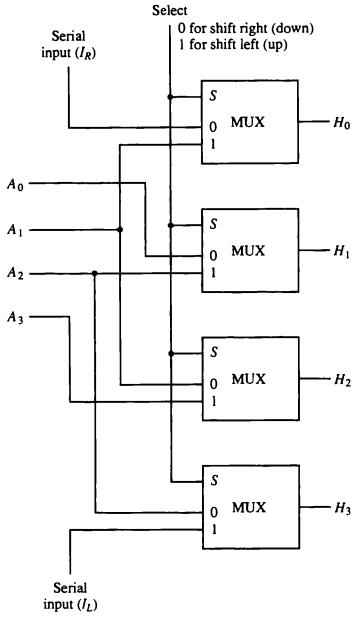
If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

Hardware Implementation

A possible choice for a shift unit would be a bidirectional shift register with parallel load (see Fig. 2-9). Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register.

shifter

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12. The 4-bit shifter has four data inputs, A_0 through A_3, and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-12 4-bit combinational circuit shifter.

(I_L) and the other for shift right (I_L). When the selection input $S = 0$, the input data are shifted right (down in the diagram). When $S = 1$, the input data are shifted left (up in the diagram). The function table in Fig. 4-12 shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

4-7 Arithmetic Logic Shift Unit

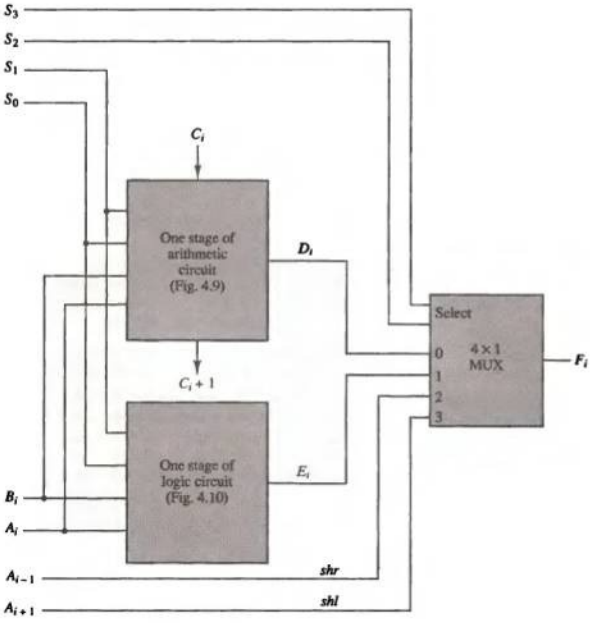
Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To

ALU

perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 4-13. The subscript i designates a typical stage. Inputs A_i and B_i are applied to both the arithmetic and logic

Figure 4-13 One stage of arithmetic logic shift unit.



units. A particular microoperation is selected with inputs S_1 and S_0. A 4×1 multiplexer at the output chooses between an arithmetic output in E_i and a logic output in H_i. The data in the multiplexer are selected with inputs S_3 and S_2. The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig. 4-13 must be repeated n times for an n-bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence. The input carry to the first stage is the input carry C_{in}, which provides a selection variable for the arithmetic operations.

The circuit whose one stage is specified in Fig. 4-13 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S_3, S_2, S_1, S_0, and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.

Table 4-8 lists the 14 operations of the ALU. The first eight are arithmetic operations (see Table 4-4) and are selected with $S_3S_2 = 00$. The next four are logic operations (see Fig. 4-10) and are selected with $S_3S_2 = 01$. The input carry has no effect during the logic operations and is marked with don't-care $\times$'s. The last two operations are shift operations and are selected with $S_3S_2 = 10$ and 11. The other three selection inputs have no effect on the shift.

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	$\times$	$F = A \wedge B$	AND
0	1	0	1	$\times$	$F = A \vee B$	OR
0	1	1	0	$\times$	$F = A \oplus B$	XOR
0	1	1	1	$\times$	$F = \overline{A}$	Complement A
1	0	$\times$	$\times$	$\times$	$F =$ shr A	Shift right A into F
1	1	$\times$	$\times$	$\times$	$F =$ shl A	Shift left A into F

PROBLEMS

4-1. Show the block diagram of the hardware (similar to Fig. 4-2a) that implements the following register transfer statement:

$$yT_2: \quad R2 \leftarrow R1, \quad R1 \leftarrow R2$$

4-2. The outputs of four registers, $R0$, $R1$, $R2$, and $R3$, are connected through 4-to-1-line multiplexers to the inputs of a fifth register, $R5$. Each register is eight bits long. The required transfers are dictated by four timing variables T_0 through T_3 as follows:

$$\begin{aligned} T_0&: \quad R5 \leftarrow R0 \\ T_1&: \quad R5 \leftarrow R1 \\ T_2&: \quad R5 \leftarrow R2 \\ T_3&: \quad R5 \leftarrow R3 \end{aligned}$$

The timing variables are mutually exclusive, which means that only one variable is equal to 1 at any given time, while the other three are equal to 0. Draw a block diagram showing the hardware implementation of the register transfers. Include the connections necessary from the four timing variables to the selection inputs of the multiplexers and to the load input of register $R5$.

4-3. Represent the following conditional control statement by two register transfer statements with control functions.

$$\text{If } (P = 1) \text{ then } (R1 \leftarrow R2) \text{ else if } (Q = 1) \text{ then } (R1 \leftarrow R3)$$

4-4. What has to be done to the bus system of Fig. 4-3 to be able to transfer information from any register to any other register? Specifically, show the connections that must be included to provide a path from the outputs of register C to the inputs of register A.

4-5. Draw a diagram of a bus system similar to the one shown in Fig. 4-3, but use three-state buffers and a decoder instead of the multiplexers.

4-6. A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with multiplexers.

a. How many selection inputs are there in each multiplexer?
b. What size of multiplexers are needed?
c. How many multiplexers are there in the bus?

4-7. The following transfer statements specify a memory. Explain the memory operation in each case.

a. $R2 \leftarrow M[AR]$
b. $M[AR] \leftarrow R3$
c. $R5 \leftarrow M[R5]$

4-8. Draw the block diagram for the hardware that implements the following statements:

$$x + yz: \quad AR \leftarrow AR + BR$$

where AR and BR are two n-bit registers and x, y, and z are control variables. Include the logic gates for the control function. (Remember that the symbol + designates an OR operation in a control or Boolean function but that it represents an arithmetic plus in a microoperation.)

4-9. Show the hardware that implements the following statement. Include the logic gates for the control function and a block diagram for the binary counter with a count enable input.

$$xyT_0 + T_1 + y'T_2: \quad AR \leftarrow AR + 1$$

4-10. Consider the following register transfer statements for two 4-bit registers $R1$ and $R2$.

$$\begin{aligned} xT: &\quad R1 \leftarrow R1 + R2 \\ x'T: &\quad R1 \leftarrow R2 \end{aligned}$$

Every time that variable $T = 1$, either the content of $R2$ is added to the content of $R1$ if $x = 1$, or the content of $R2$ is transferred to $R1$ if $x = 0$. Draw a diagram showing the hardware implementation of the two statements. Use block diagrams for the two 4-bit registers, a 4-bit adder, and a quadruple 2-to-1-line multiplexer that selects the inputs to $R1$. In the diagram, show how the control variables x and T select the inputs of the multiplexer and the load input of register $R1$.

4-11. Using a 4-bit counter with parallel load as in Fig. 2-11 and a 4-bit adder as in Fig. 4-6, draw a block diagram that shows how to implement the following statements:

$$\begin{aligned} x: &\quad R1 \leftarrow R1 + R2 \quad \text{Add } R2 \text{ to } R1 \\ x'y: &\quad R1 \leftarrow R1 + 1 \quad \text{Increment } R1 \end{aligned}$$

where $R1$ is a counter with parallel load and $R2$ is a 4-bit register.

4-12. The adder-subtractor circuit of Fig. 4-7 has the following values for input mode M and data inputs A and B. In each case, determine the values of the outputs: S_3, S_2, S_1, S_0, and C_4.

	M	A	B
a.	0	0111	0110
b.	0	1000	1001
c.	1	1100	1000
d.	1	0101	1010
e.	1	0000	0001

4-13. Design a 4-bit combinational circuit decrementer using four full-adder circuits.

4-14. Assume that the 4-bit arithmetic circuit of Fig. 4-9 is enclosed in one IC package. Show the connections among two such ICs to form an 8-bit arithmetic circuit.

4-15. Design an arithmetic circuit with one selection variable S and two n-bit data inputs A and B. The circuit generates the following four arithmetic operations in conjunction with the input carry C_{in}. Draw the logic diagram for the first two stages.

S	$C_{in} = 0$	$C_{in} = 1$
0	$D = A + B$ (add)	$D = A + 1$ (increment)
1	$D = A - 1$ (decrement)	$D = A + \overline{B} + 1$ (subtract)

4-16. Derive a combinational circuit that selects and generates any of the 16 logic functions listed in Table 4-5.

4-17. Design a digital circuit that performs the four logic operations of exclusive-OR, exclusive-NOR, NOR, and NAND. Use two selection variables. Show the logic diagram of one typical stage.

4-18. Register A holds the 8-bit binary 11011001. Determine the B operand and the logic microoperation to be performed in order to change the value in A to:
a. 01101101
b. 11111101

4-19. The 8-bit registers AR, BR, CR, and DR initially have the following values:

$$\begin{aligned} AR &= 11110010 \\ BR &= 11111111 \\ CR &= 10111001 \\ DR &= 11101010 \end{aligned}$$

Determine the 8-bit values in each register after the execution of the following sequence of microoperations.

$AR \leftarrow AR + BR$	Add BR to AR
$CR \leftarrow CR \wedge DR$, $BR \leftarrow BR + 1$	AND DR to CR, increment BR
$AR \leftarrow AR - CR$	Subtract CR from AR

4-20. An 8-bit register contains the binary value 10011100. What is the register value after an arithmetic shift right? Starting from the initial number 10011100, determine the register value after an arithmetic shift left, and state whether there is an overflow.

4-21. Starting from an initial value of R = 11011101, determine the sequence of binary values in R after a logical shift-left, followed by a circular shift-right, followed by a logical shift-right and a circular shift-left.

4-22. What is the value of output H in Fig. 4-12 if input A is 1001, $S = 1$, $I_R = 1$, and $I_L = 0$?

4-23. What is wrong with the following register transfer statements?

a. xT: $AR \leftarrow \overline{AR}$, $AR \leftarrow 0$

b. yT: $R1 \leftarrow R2$, $R1 \leftarrow R3$

c. zT: $PC \leftarrow AR$, $PC \leftarrow PC + 1$

REFERENCES

1. Bell, C. G., J. C. Mudge, and J. E. McNamara, *Computer Engineering*. Bedford, MA: Digital Press, 1980.
2. Booth, T. L., *Introduction to Computer Engineering*, 3rd ed. New York: John Wiley, 1984.
3. Hays, J. F., *Computer Architecture and Organization*, 2nd ed. New York: McGraw-Hill, 1988.
4. Hill, F. J., and G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 3rd ed. New York: John Wiley, 1987.
5. Mano, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.
6. Patterson, D. A., and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1990.
7. Prosser, F. P., and D. E. Winkel, *The Art of Digital Design*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1987.
8. Sandige, R. S., *Modern Digital Design*. New York: McGraw-Hill, 1990.
9. Shiva, S. G., *Computer Design and Architecture*, 2nd ed. New York: HarperCollins Publishers, 1991.
10. Tomek, I., *Introduction to Computer Organization*. Rockville, MD: Computer Science Press, 1981.
11. Ward, S. A., and R.H. Halstead, Jr., *Computation Structures*. Cambridge, MA: MIT Press, 1990.

CHAPTER FIVE

Basic Computer Organization and Design

IN THIS CHAPTER

5-1 Instruction Codes

In this chapter we introduce a basic computer and show how its operation can be specified with register transfer statements. The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses. The design of the computer is then carried out in detail. Although the basic computer presented in this chapter is very small compared to commercial computers, it has the advantage of being simple enough so we can demonstrate the design process without too many complications.

The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers. The general-purpose digital computer is capable of executing various microoperations and, in addition, can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations,

operands, and the sequence by which processing has to occur. The data-processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set. The ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer.

instruction code

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its

operation code

operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations. As an illustration, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configuration 110010 assigned to the ADD operation. When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

At this point we must recognize the relationship between a computer operation and a microoperation. An operation is part of an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers. For every operation code, the control issues a sequence of microoperations needed for the hardware implementation of the specified operation. For this reason, an operation code is sometimes called a macrooperation because it specifies a set of microoperations.

The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory. An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored. Memory words can be specified in instruction codes by their address. Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^k registers. There are many variations for arranging the binary code of instructions, and each computer has its own particular instruction code format. Instruction code formats are con-

ceived by computer designers who specify the architecture of the computer. In this chapter we choose a particular instruction code to explain the basic organization and design of digital computers.

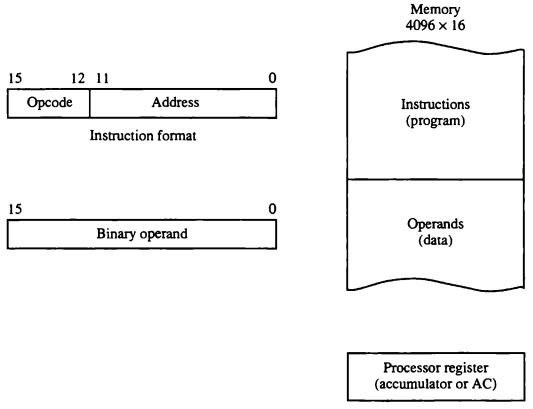
Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

Figure 5-1 depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

opcode

Figure 5-1 Stored program organization.



accumulator (AC)

Computers that have a single-processor register usually assign to it the name accumulator and label it *AC*. The operation is performed with the memory operand and the content of *AC*.

If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear *AC*, complement *AC*, and increment *AC* operate on data stored in the *AC* register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

Indirect Address

immediate instruction

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

As an illustration of this configuration, consider the instruction code format shown in Fig. 5-2(a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by *I*. The mode bit is 0 for a direct address and 1 for an indirect address. A direct address instruction is shown in Fig. 5-2(b). It is placed in address 22 in memory. The *I* bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of *AC*. The instruction in address 35 shown in Fig. 5-2(c) has a mode bit $I = 1$. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of *AC*. The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself. We define the *effective address* to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction. Thus the effective address in the instruction of Fig. 5-2(b) is 457 and in the instruction of Fig 5-2(c) is 1350.

effective address

The direct and indirect addressing modes are used in the computer presented in this chapter. The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of

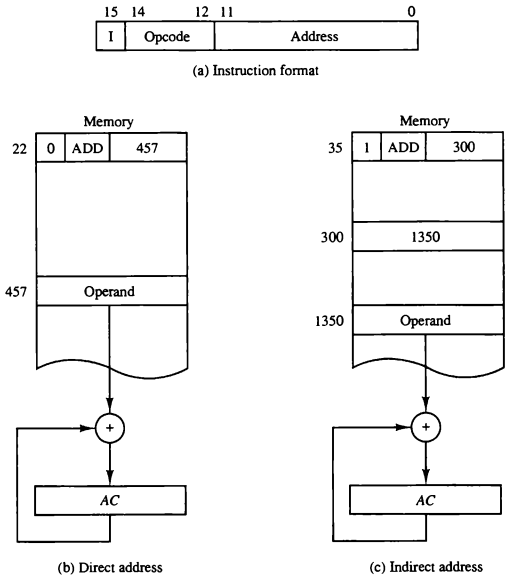


Figure 5-2 Demonstration of direct and indirect address.

data. The pointer could be placed in a processor register instead of memory as done in commercial computers.

5-2 Computer Registers

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instruction

code after it is read from memory. The computer needs processor registers for manipulating data and a register for holding a memory address. These requirements dictate the register configuration shown in Fig. 5-3. The registers are also listed in Table 5-1 together with a brief description of their function and the number of bits that they contain.

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation part of the instruction and a bit to specify a direct or indirect address. The data register (*DR*) holds the operand read from memory. The accumulator (*AC*) register is a general-purpose processing register. The instruction read from memory is placed in the instruction register (*IR*). The temporary register (*TR*) is used for holding temporary data during the processing.

TABLE 5-1 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

program counter (PC)

The memory address register (*AR*) has 12 bits since this is the width of a memory address. The program counter (*PC*) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The *PC* goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to *PC* to become the address of the next instruction. To read an instruction, the content of *PC* is taken as the address for memory and a memory read cycle is initiated. *PC* is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register (*INPR*) receives an 8-bit character from an input device. The output register (*OUTR*) holds an 8-bit character for an output device.

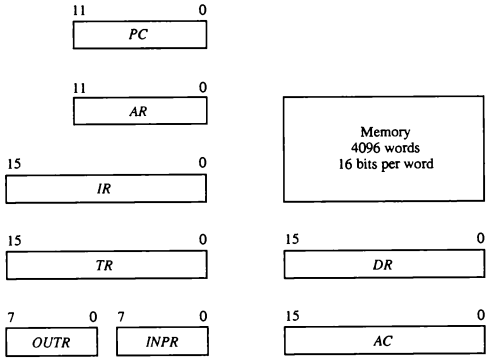


Figure 5-3 Basic computer registers and memory.

Common Bus System

The basic computer has eight registers, a memory unit, and a control unit (to be presented in Sec. 5-4). Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. We have shown in Sec. 4-3 how to construct a bus system using multiplexers or three-state buffer gates. The connection of the registers and memory of the basic computer to a common bus system is shown in Fig. 5-4.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2, S_1, and S_0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of *DR* is 3. The 16-bit outputs of *DR* are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

load (LD)

Four registers, *DR*, *AC*, *IR*, and *TR*, have 16 bits each. Two registers, *AR*

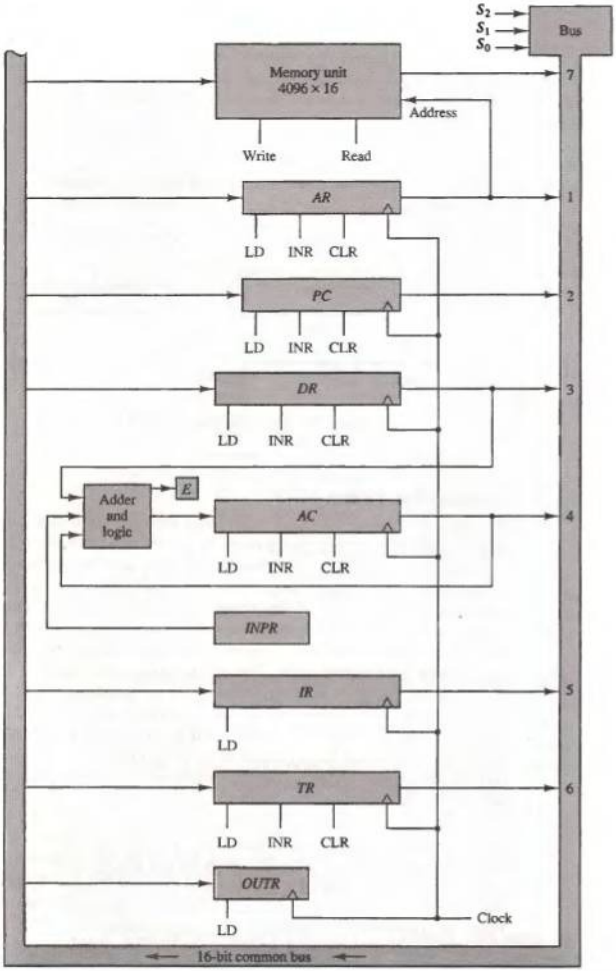


Figure 5-4 Basic computer registers connected to a common bus.

and *PC*, have 12 bits each since they hold a memory address. When the contents of *AR* or *PC* are applied to the 16-bit common bus, the four most significant bits are set to 0's. When *AR* or *PC* receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register *INPR* and the output register *OUTR* have 8 bits each and communicate with the eight least significant bits in the bus. *INPR* is connected to provide information to the bus but *OUTR* can only receive information from the bus. This is because *INPR* receives a character from an input device which is then transferred to *AC*. *OUTR* receives a character from *AC* and delivers it to an output device. There is no transfer from *OUTR* to any of the other registers.

The 16 lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). This type of register is equivalent to a binary counter with parallel load and synchronous clear similar to the one shown in Fig. 2-11. The increment operation is achieved by enabling the count input of the counter. Two registers have only a LD input. This type of register is shown in Fig. 2-7.

The input data and output data of the memory are connected to the common bus, but the memory address is connected to *AR*. Therefore, *AR* must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except *AC*.

memory address

The 16 inputs of *AC* come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of *AC*. They are used to implement register microoperations such as complement *AC* and shift *AC*. Another set of 16-bit inputs come from the data register *DR*. The inputs from *DR* and *AC* are used for arithmetic and logic microoperations, such as add *DR* to *AC* or AND *DR* to *AC*. The result of an addition is transferred to *AC* and the end carry-out of the addition is transferred to flip-flop *E* (extended *AC* bit). A third set of 8-bit inputs come from the input register *INPR*. The operation of *INPR* and *OUTR* is explained in Sec. 5-7.

Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into *AC*. For example, the two microoperations

$$DR \leftarrow AC \quad \text{and} \quad AC \leftarrow DR$$

can be executed at the same time. This can be done by placing the content of *AC* on the bus (with $S_2S_1S_0 = 100$), enabling the LD (load) input of *DR*, trans-

ferring the content of *DR* through the adder and logic circuit into *AC*, and enabling the LD (load) input of *AC*, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

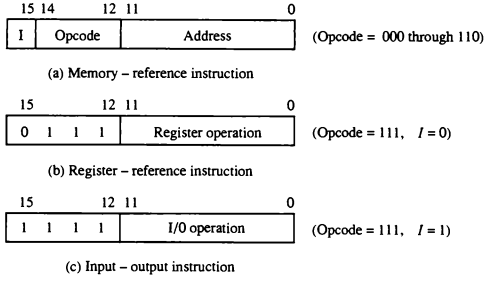
5-3 Computer Instructions

Instruction format

The basic computer has three instruction code formats, as shown in Fig. 5-5. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode *I*. *I* is equal to 0 for direct address and to 1 for indirect address (see Fig. 5-2). The register-reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the *AC* register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, an input–output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input–output operation or test performed.

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode *I*. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the

Figure 5-5 Basic computer instruction formats.



instruction is a register-reference type. If the bit is 1, the instruction is an input–output type. Note that the bit in position 15 of the instruction code is designated by the symbol *I* but is not used as a mode bit when the operation code is equal to 111.

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. However, since register-reference and input–output instructions use the remaining 12 bits as part of the operation code, the total number of instructions can exceed eight. In fact, the total number of instructions chosen for the basic computer is equal to 25.

The instructions for the computer are listed in Table 5-2. The symbol designation is a three-letter word and represents an abbreviation intended for

TABLE 5-2 Basic Computer Instructions

	Hexadecimal code		
Symbol	$I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
HLT	7001		Halt computer
INP	F800		Input character to *AC*
OUT	F400		Output character from *AC*
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

hexadecimal code

programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I. When $I = 0$, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When $I = 1$, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1.

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits. The input–output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Instruction Set Completeness

Before investigating the operations performed by the instructions, let us discuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision-making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the

user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

The instructions listed in Table 5-2 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction, ADD, and two related instructions, complement *AC*(CMA) and increment *AC*(INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement *AC*(CMA), and clear *AC*(CLA). The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables (listed in Table 4-6). Moving information from memory to *AC* is accomplished with the load *AC*(LDA) instruction. Storing information from *AC* into memory is done with the store *AC*(STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer. The programs are presented in Chap. 6 together with other programming examples for the basic computer. By using a limited number of instructions it is possible to show the detailed logic design of the computer. A more complete set of instructions would have made the design too complex. In this way we can demonstrate the basic principles of computer organization and design without going into excessive complex details. In Chap. 8 we present a complete list of computer instructions that are included in most commercial computers.

The function of each instruction listed in Table 5-2 and the microoperations needed for their execution are presented in Secs. 5-5 through 5-7. We delay this discussion because we must first consider the control unit and understand its internal organization.

5-4 Timing and Control

clock pulses

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by

a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

hardwired control

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section. A microprogrammed control unit for a similar computer is presented in Chap. 7.

microprogrammed control

control unit

The block diagram of the control unit is shown in Fig. 5-6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (*IR*). The position of this register in the common bus system is indicated in Fig. 5-4. The instruction register is shown again in Fig. 5-6, where it is divided into three parts: the *I* bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 × 8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol *I*. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}. The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

timing signals

The sequence counter *SC* can be incremented or cleared synchronously (see the counter of Fig. 2-11). Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 × 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T_0. As an example, consider the case where *SC* is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, *SC* is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3T_4: \quad SC \leftarrow 0$$

The timing diagram of Fig. 5-7 shows the time relationship of the control signals. The sequence counter *SC* responds to the positive transition of the clock. Initially, the CLR input of *SC* is active. The first positive transition of the

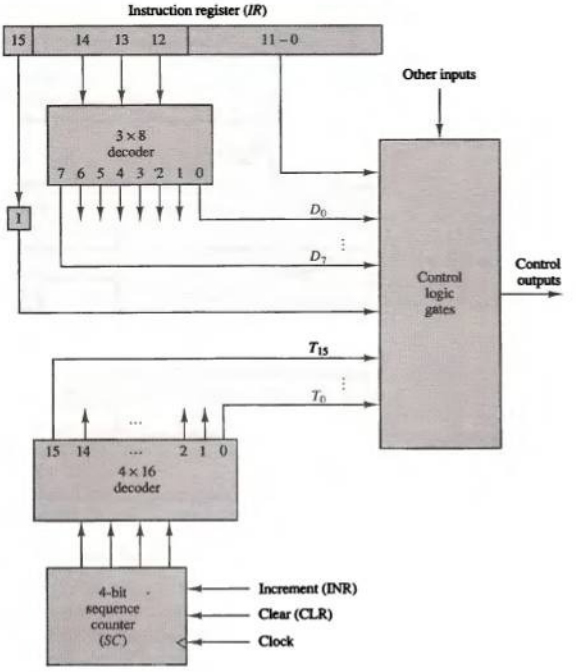


Figure 5-6 Control unit of basic computer.

clock clears *SC* to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0. *SC* is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4, and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition.) If *SC* is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

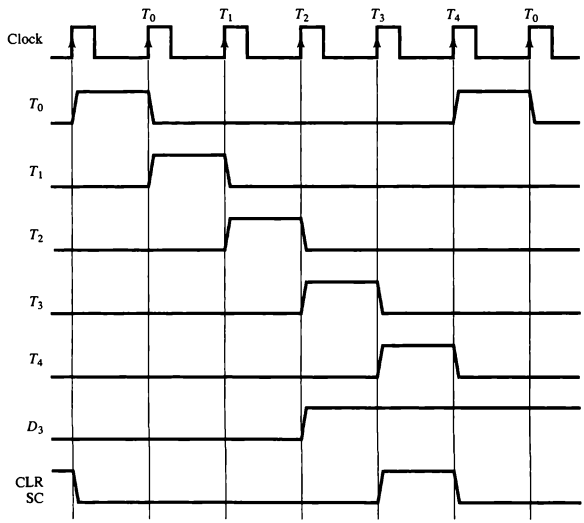


Figure 5-7 Example of control timing signals.

The last three waveforms in Fig. 5-7 show how *SC* is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of *SC*. On the next positive clock transition (the one marked T_4 in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if *SC* were incremented instead of cleared.

A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor

until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$T_0: \quad AR \leftarrow PC$$

specifies a transfer of the content of *PC* into *AR* if timing signal T_0 is active. T_0 is active during an entire clock cycle interval. During this time the content of *PC* is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of *AR* is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter *SC* from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive.

5-5 Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode

Initially, the program counter *PC* is loaded with the address of the first instruction in the program. The sequence counter *SC* is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, *SC* is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T_0: $AR \leftarrow PC$
T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$

Since only *AR* is connected to the address inputs of memory, it is necessary to transfer the address from *PC* to *AR* during the clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register *IR* with the clock transition associated with timing

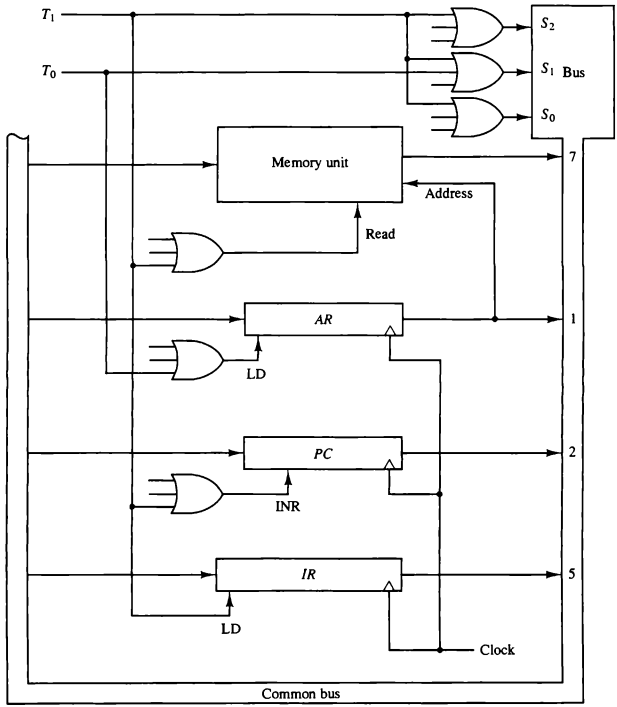


Figure 5-8 Register transfers for the fetch phase.

signal T_1. At the same time, *PC* is incremented by one to prepare it for the address of the next instruction in the program. At time T_2, the operation code in *IR* is decoded, the indirect bit is transferred to flip-flop *I*, and the address part of the instruction is transferred to *AR*. Note that *SC* is incremented after each clock pulse to produce the sequence T_0, T_1, and T_2.

Figure 5-8 shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of *PC* to *AR* we must apply timing signal T_0 to achieve the following connection:

1. Place the content of *PC* onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
2. Transfer the content of the bus to *AR* by enabling the LD input of *AR*.

The next clock transition initiates the transfer from *PC* to *AR* since $T_0 = 1$. In order to implement the second statement

$$T_1: \quad IR \leftarrow M[AR], \quad PC \leftarrow PC + 1$$

it is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
3. Transfer the content of the bus to *IR* by enabling the LD input of *IR*.
4. Increment *PC* by enabling the INR input of *PC*.

The next clock transition initiates the read and increment operations since $T_1 = 1$.

Figure 5-8 duplicates a portion of the bus system and shows how T_0 and T_1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory. The flowchart of Fig. 5-9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. The three possible instruction types available in the basic computer are specified in Fig. 5-5.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. From Fig. 5-5 we determine that if $D_7 = 1$, the instruction must be a

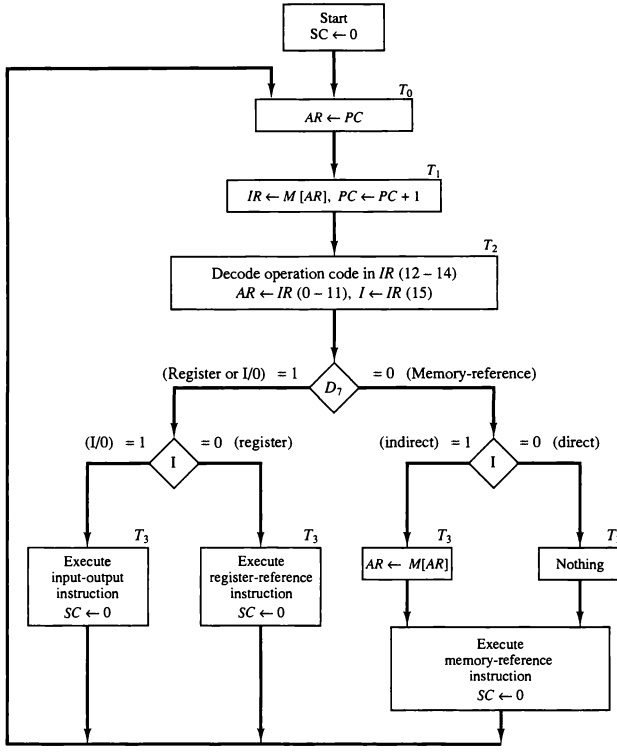


Figure 5-9 Flowchart for instruction cycle (initial configuration).

register-reference or input–output type. If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If $D_7 = 0$ and $I = 1$, we have a memory-reference instruction with an indirect address. It is then necessary to read the

indirect address

effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement

$$AR \leftarrow M[AR]$$

Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

$D_7' IT_3$: $AR \leftarrow M[AR]$
$D_7' I'T_3$: Nothing
$D_7 I'T_3$: Execute a register-reference instruction
$D_7 IT_3$: Execute an input–output instruction

When a memory-reference instruction with $I = 0$ is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when $D_7'T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4. A register-reference or input–output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$.

Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement $SC \leftarrow SC + 1$, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement $SC \leftarrow 0$.

The register transfers needed for the execution of the register-reference instructions are presented in this section. The memory-reference instructions are explained in the next section. The input–output instructions are included in Sec. 5-7.

Register-Reference Instructions

Register-reference instructions are recognized by the control when $D_7 = 1$ and $I = 0$. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in $IR(0–11)$. They were also transferred to AR during time T_2.

The control functions and microoperations for the register-reference in-

structions are listed in Table 5-3. These instructions are executed with the clock transition associated with timing variable T_3. Each control function needs the Boolean relation $D_7I'T_3$, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in $IR(0–11)$. By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i. For example, the instruction CLA has the hexadecimal code 7800 (see Table 5-2), which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I'. The next three bits constitute the operation code and are recognized from decoder output D_7. Bit 11 in IR is 1 and is recognized from B_{11}. The control function that initiates the microoperation for this instruction is $D_7I'T_3B_{11} = rB_{11}$. The execution of a register-reference instruction is completed at time T_3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1). The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in $AC(15) = 0$; it is negative when $AC(15) = 1$. The content of AC is zero ($AC = 0$) if all the flip-flops of the register are zero. The HLT instruction clears a start–stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start–stop flip-flop must be set manually.

TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

5-6 Memory-Reference Instructions

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Looking back to Table 5-2, where the instructions are listed, we find that some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation. We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

Table 5-4 lists the seven memory-reference instructions. The decoded output D_i for $i = 0, 1, 2, 3, 4, 5$, and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when $I = 0$, or during timing signal T_3 when $I = 1$. The execution of the memory-reference instructions starts with timing signal T_4. The symbolic description of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control functions and microoperations needed for their execution. A flowchart that summarizes all the microoperations is presented at the end of this section.

effective address

TABLE 5-4 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to *AC*

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of

the operation is transferred to AC. The microoperations that execute this instruction are:

$$D_0T_4: \quad DR \leftarrow M[AR]$$
$$D_0T_5: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

The control function for this instruction uses the operation decoder D_0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T_4 transfers the operand from memory into DR. The clock transition associated with the next timing signal T_5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T_0 to start a new instruction cycle.

ADD to *AC*

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are

$$D_1T_4: \quad DR \leftarrow M[AR]$$
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

The same two timing signals, T_4 and T_5, are used again but with operation decoder D_1 instead of D_0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-reference instruction.

LDA: Load to *AC*

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are

$$D_2T_4: \quad DR \leftarrow M[AR]$$
$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

Looking back at the bus system shown in Fig. 5-4 we note that there is no direct path from the bus into AC. The adder and logic circuit receive information from DR which can be transferred into AC. Therefore, it is necessary to read the

memory word into *DR* first and then transfer the content of *DR* into *AC*. The reason for not connecting the bus to the inputs of *AC* is the delay encountered in the adder and logic circuit. It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle. By not connecting the bus to the inputs of *AC* we can maintain one clock cycle per microoperation.

STA: Store *AC*

This instruction stores the content of *AC* into the memory word specified by the effective address. Since the output of *AC* is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. Remember that *PC* holds the address of the instruction to be read from memory in the next instruction cycle. *PC* is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

$$D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

The effective address from *AR* is transferred through the common bus to *PC*. Resetting *SC* to 0 transfers control to T_0. The next instruction is then fetched and executed from the memory address given by the new value in *PC*.

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in *PC*) into a memory location specified by the effective address. The effective address plus one is then transferred to *PC* to serve as the address of the first instruction in the subroutine. This operation was specified in Table 5-4 with the following register transfer:

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 5-10. The BSA instruction is assumed to be in memory at address 20. The *I* bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, *PC* contains 21, which is the address of the next instruction in the program (referred to as the *return address*). *AR* holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

return address

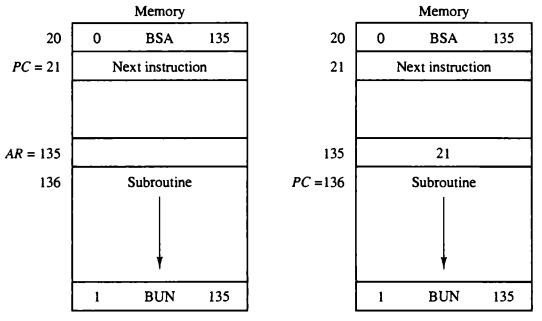
$$M[135] \leftarrow 21, \quad PC \leftarrow 135 + 1 = 136$$

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to *PC*. The next instruction cycle finds *PC* with the value 21, so control continues to execute the instruction at the return address.

subroutine call

The BSA instruction performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor

Figure 5-10 Example of BSA instruction execution.



(a) Memory, *PC*, and *AR* at time T_4 (b) Memory and *PC* after execution

register or in a portion of memory called a stack. This is discussed in more detail in Sec. 8-7.

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

Timing signal T_4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC.

ISZ: Increment and Skip if Zero

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

$$D_6T_4: \quad DR \leftarrow M[AR]$$
$$D_6T_5: \quad DR \leftarrow DR + 1$$
$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

Control Flowchart

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Fig. 5-11. The control functions are indicated on top of each box. The microoperations that are performed during time T_4, T_5, or T_6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T_0 to start the next instruction cycle.

Note that we need only seven timing signals to execute the longest instruction (ISZ). The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.

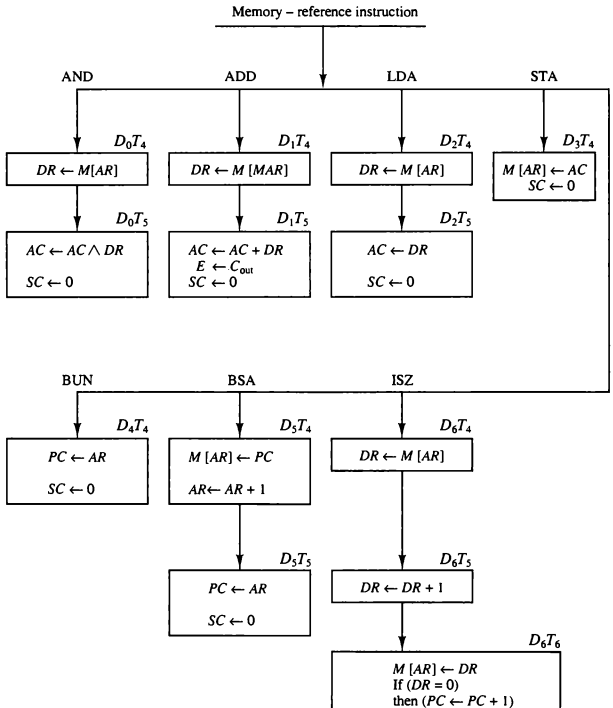


Figure 5-11 Flowchart for memory-reference instructions.

5-7 Input–Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of

input and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer. Input–output organization is dicsussed further in Chap. 11.

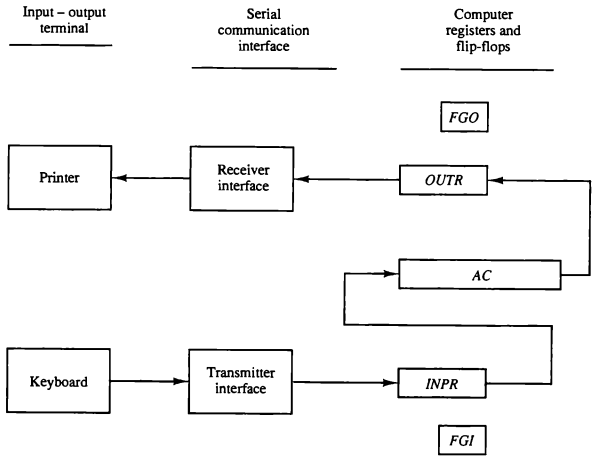
Input–Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register *INPR*. The serial information for the printer is stored in the output register *OUTR*. These two registers communicate with a communication interface serially and with the *AC* in parallel. The input–output configuration is shown in Fig. 5-12. The transmitter interface receives serial information from the keyboard and transmits it to *INPR*. The receiver interface receives information from *OUTR* and sends it to the printer serially. The operation of the serial communication interface is explained in Sec. 11-3.

input register

The input register *INPR* consists of eight bits and holds an alphanumeric input information. The 1-bit input flag *FGI* is a control flip-flop. The flag bit is

Figure 5-12 Input-output configuration.



set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag *FGI* is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into *INPR* and the input flag *FGI* is set to 1. As long as the flag is set, the information in *INPR* cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from *INPR* is transferred in parallel into *AC* and *FGI* is cleared to 0. Once the flag is cleared, new information can be shifted into *INPR* by striking another key.

output register

The output register *OUTR* works similarly but the direction of information flow is reversed. Initially, the output flag *FGO* is set to 1. The computer checks the flag bit; if it is 1, the information from *AC* is transferred in parallel to *OUTR* and *FGO* is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets *FGO* to 1. The computer does not load a new character into *OUTR* when *FGO* is 0 because this condition indicates that the output device is in the process of printing the character.

Input–Output Instructions

Input and output instructions are needed for transferring information to and from *AC* register, for checking the flag bits, and for controlling the interrupt facility. Input–output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input–output instructions are listed in Table 5-5. These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in $IR(6–11)$. By assigning the symbol B_i to bit i of IR, all control functions

TABLE 5-5 Input-Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

can be denoted by pB_i for i = 6 though 11. The sequence counter *SC* is cleared to 0 when $p = D_7IT_3 = 1$.

The INP instruction transfers the input information from *INPR* into the eight low-order bits of *AC* and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of *AC* into the output register *OUTR* and clears the output flag to 0. The next two instructions in Table 5-5 check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. (Examples of input and output programs are given in Sec. 6-8.) The last two instructions set and clear an interrupt enable flip-flop *IEN*. The purpose of *IEN* is explained in conjunction with the interrupt operation.

Program Interrupt

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input–output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs. Assume that the input–output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop *IEN* can be set and cleared with two instructions. When *IEN* is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When *IEN* is set to 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

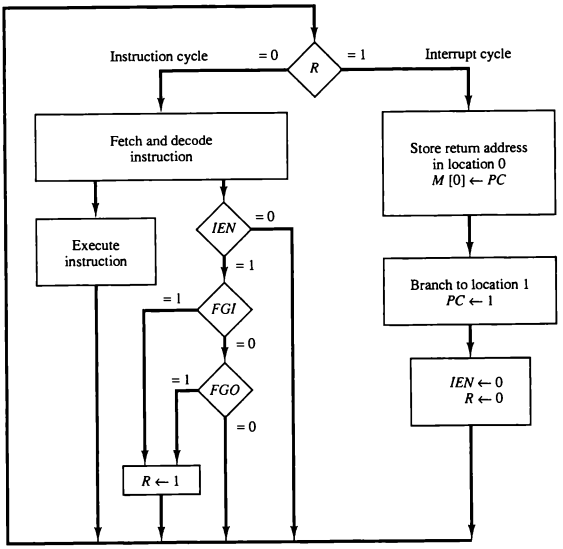


Figure 5-13 Flowchart for interrupt cycle.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 5-13. An interrupt flip-flop R is included in the computer. When $R = 0$, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while $IEN = 1$, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

interrupt cycle

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor

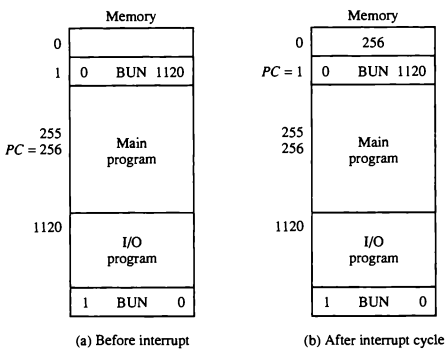
register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into *PC* and clears *IEN* and *R* so that no more interruptions can occur until the interrupt request from the flag has been serviced.

An example that shows what happens during the interrupt cycle is shown in Fig. 5-14. Suppose that an interrupt occurs and *R* is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in *PC*. The programmer has previously placed an input–output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 5-14(a).

When control reaches timing signal T_0 and finds that $R = 1$, it proceeds with the interrupt cycle. The content of *PC* (256) is stored in memory location 0, *PC* is set to 1, and *R* is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of *PC*. The branch instruction at address 1 causes the program to transfer to the input–output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set *IEN* to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. 5-14(b).

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction is

Figure 5-14 Demonstration of the interrupt cycle.



read from memory during the fetch phase, control goes to the indirect phase (because $I = 1$) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into *PC* the return address from location 0.

Interrupt Cycle

We are now ready to list the register transfer statements for the interrupt cycle. The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop *R* is equal to 1. This flip-flop is set to 1 if *IEN* = 1 and either *FGI* or *FGO* are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1, or T_2 are active. The condition for setting flip-flop *R* to 1 can be expressed with the following register transfer statement:

$$T_0'T_1'T_2'(IEN)(FGI + FGO): \quad R \leftarrow 1$$

The symbol + between *FGI* and *FGO* in the control function designates a logic OR operation. This is ANDed with *IEN* and $T_0'T_1'T_2'$.

modified fetch phase

We now modify the fetch and decode phases of the instruction cycle. Instead of using only timing signals T_0, T_1, and T_2 (as shown in Fig. 5-9) we will AND the three timing signals with R' so that the fetch and decode phases will be recognized from the three control functions $R'T_0$, $R'T_1$, and $R'T_2$. The reason for this is that after the instruction is executed and *SC* is cleared to 0, the control will go through a fetch phase only if $R = 0$. Otherwise, if $R = 1$, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in *PC*) into memory location 0, branches to memory location 1, and clears *IEN*, *R*, and *SC* to 0. This can be done with the following sequence of microoperations:

$$\begin{aligned} RT_0&: \quad AR \leftarrow 0, \quad TR \leftarrow PC \\ RT_1&: \quad M[AR] \leftarrow TR, \quad PC \leftarrow 0 \\ RT_2&: \quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0 \end{aligned}$$

During the first timing signal *AR* is cleared to 0, and the content of *PC* is transferred to the temporary register *TR*. With the second timing signal, the return address is stored in memory at location 0 and *PC* is cleared to 0. The third timing signal increments *PC* to 1, clears *IEN* and *R*, and control goes back to T_0 by clearing *SC* to 0. The beginning of the next instruction cycle has the condition $R'T_0$ and the content of *PC* is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

5-8 Complete Computer Description

flowchart for basic computer

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in Fig. 5-15. The interrupt flip-flop R may be set at any time during the indirect or execute phases. Control returns to timing signal T_0 after SC is cleared to 0. If $R = 1$, the computer goes through an interrupt cycle. If $R = 0$, the computer goes through an instruction cycle. If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction according to the flowchart of Fig. 5-11. If the instruction is one of the register-reference instructions, it is executed with one of the microoperations listed in Table 5-3. If it is an input–output instruction, it is executed with one of the microoperations listed in Table 5-5.

Instead of using a flowchart, we can describe the operation of the computer with a list of register transfer statements. This is done by accumulating all the control functions and microoperations in one table. The entries in the table are taken from Figs. 5-11 and 5-15, and Tables 5-3 and 5-5.

The control functions and microoperations for the entire computer are summarized in Table 5-6. The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer. The control functions and conditional control statements listed in the table formulate the Boolean functions for the gates in the control unit. The list of microoperations specifies the type of control inputs needed for the registers and memory. A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.

5-9 Design of Basic Computer

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each
2. Nine registers: *AR*, *PC*, *DR*, *AC*, *IR*, *TR*, *OUTR*, *INPR*, and *SC*
3. Seven flip-flops: *I*, *S*, *E*, *R*, *IEN*, *FGI*, and *FGO*
4. Two decoders: a 3 × 8 operation decoder and a 4 × 16 timing decoder
5. A 16-bit common bus
6. Control logic gates
7. Adder and logic circuit connected to the input of *AC*

The memory unit is a standard component that can be obtained readily from a commercial source. The registers are of the type shown in Fig. 2-11 and

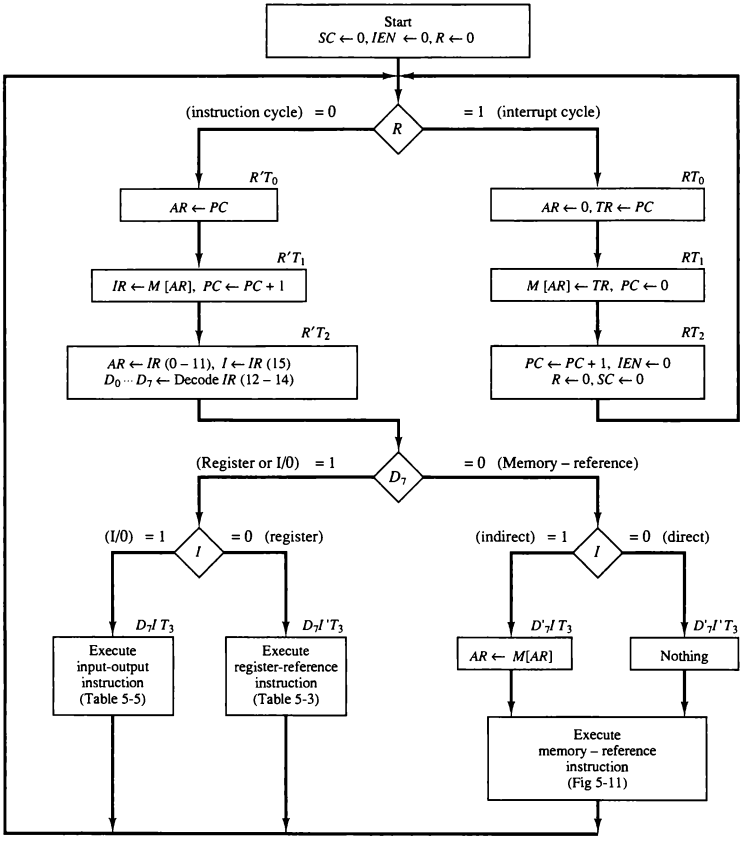


Figure 5-15 Flowchart for computer operation.

TABLE 5-6 Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
Indirect	D_7IT_3:	$AR \leftarrow M[AR]$
Interrupt:		
$T_0'T_1'T_2(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Register-reference:		
	$D_7I'T_3 = r$ (common to all register-reference instructions)	
	$IR(i) = B_i$ $(i = 0, 1, 2, \ldots, 11)$	
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		
	$D_7IT_3 = p$ (common to all input–output instructions)	
	$IR(i) = B_i$ $(i = 6, 7, 8, 9, 10, 11)$	
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

are similar to integrated circuit type 74163. The flip-flops can be either of the *D* or *JK* type, as described in Sec. 1-6. The two decoders are standard components similar to the ones presented in Sec. 2-2. The common bus system can be constructed with sixteen 8 × 1 multiplexers in a configuration similar to the one shown in Fig. 4-3. We are now going to show how to design the control logic gates. The next section deals with the design of the adder and logic circuit associated with *AC*.

Control Logic Gates

control unit

The block diagram of the control logic gates is shown in Fig. 5-6. The inputs to this circuit come from the two decoders, the *I* flip-flop, and bits 0 through 11 of *IR*. The other inputs to the control logic are: *AC* bits 0 through 15 to check if $AC = 0$ and to detect the sign bit in *AC*(15); *DR* bits 0 through 15 to check if $DR = 0$; and the values of the seven flip-flops.

The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers
2. Signals to control the read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops
4. Signals for S_2, S_1, and S_0 to select a register for the bus
5. Signals to control the *AC* adder and logic circuit

The specifications for the various control signals can be obtained directly from the list of register transfer statements in Table 5-6.

Control of Registers and Memory

The registers of the computer connected to a common bus system are shown in Fig. 5-4. The control inputs of the registers are LD (load), INR (increment), and CLR (clear). Suppose that we want to derive the gate structure associated with the control inputs of *AR*. We scan Table 5-6 to find all the statements that change the content of *AR*:

$$\begin{aligned} R'T_0&: \quad AR \leftarrow PC \\ R'T_2&: \quad AR \leftarrow IR(0\text{–}11) \\ D_7IT_3&: \quad AR \leftarrow M[AR] \\ RT_6&: \quad AR \leftarrow 0 \\ D_5T_4&: \quad AR \leftarrow AR + 1 \end{aligned}$$

The first three statements specify transfer of information from a register or memory to *AR*. The content of the source register or memory is placed on

the bus and the content of the bus is transferred into AR by enabling its LD control input. The fourth statement clears AR to 0. The last statement increments AR by 1. The control functions can be combined into three Boolean expressions as follows:

$$\begin{aligned} \text{LD}(AR) &= R'T_0 + R'T_2 + D_7'IT_3 \\ \text{CLR}(AR) &= RT_0 \\ \text{INR}(AR) &= D_5T_4 \end{aligned}$$

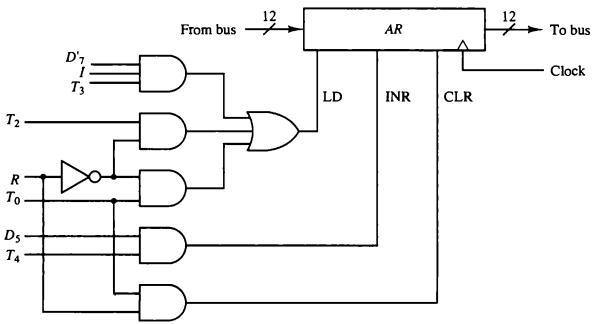
where LD(AR) is the load input of AR, CLR(AR) is the clear input of AR, and INR(AR) is the increment input of AR. The control gate logic associated with AR is shown in Fig. 5-16.

In a similar fashion we can derive the control gates for the other registers as well as the logic needed to control the read and write inputs of memory. The logic gates associated with the read input of memory is derived by scanning Table 5-6 to find the statements that specify a read operation. The read operation is recognized from the symbol $\leftarrow M[AR]$.

$$\text{Read} = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

The output of the logic gates that implement the Boolean expression above must be connected to the read input of memory.

Figure 5-16 Control gates associated with AR.



Control of Single Flip-flops

The control gates for the seven flip-flops can be determined in a similar manner. For example, Table 5-6 shows that *IEN* may change as a result of the two instructions ION and IOF.

$$pB_7: \quad IEN \leftarrow 1$$
$$pB_6: \quad IEN \leftarrow 0$$

where $p = D_7IT_3$ and B_7 and B_6 are bits 7 and 6 of *IR*, respectively. Moreover, at the end of the interrupt cycle *IEN* is cleared to 0.

$$RT_2: \quad IEN \leftarrow 0$$

If we use a *JK* flip-flip for *IEN*, the control gate logic will be as shown in Fig. 5-17.

Control of Common Bus

The 16-bit common bus shown in Fig. 5-4 is controlled by the selection inputs S_2, S_1, and S_0. The decimal number shown with each bus input specifies the equivalent binary number that must be applied to the selection inputs in order to select the corresponding register. Table 5-7 specifies the binary numbers for $S_2S_1S_0$ that select each register. Each binary number is associated with a Boolean variable x_1 through x_7, corresponding to the gate structure that must be active in order to select the register or memory for the bus. For example, when $x_1 = 1$, the value of $S_2S_1S_0$ must be 001 and the output of *AR* will be selected for the bus. Table 5-7 is recognized as the truth table of a binary encoder. The placement of the encoder at the inputs of the bus selection logic is shown in Fig. 5-18. The Boolean functions for the encoder are

$$S_0 = x_1 + x_3 + x_5 + x_7$$
$$S_1 = x_2 + x_3 + x_6 + x_7$$
$$S_2 = x_4 + x_5 + x_6 + x_7$$

Figure 5-17 Control inputs for *IEN*.

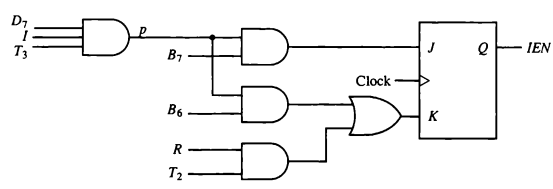


TABLE 5-7 Encoder for Bus Selection Circuit

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	*AR*
0	1	0	0	0	0	0	0	1	0	*PC*
0	0	1	0	0	0	0	0	1	1	*DR*
0	0	0	1	0	0	0	1	0	0	*AC*
0	0	0	0	1	0	0	1	0	1	*IR*
0	0	0	0	0	1	0	1	1	0	*TR*
0	0	0	0	0	0	1	1	1	1	Memory

To determine the logic for each encoder input, it is necessary to find the control functions that place the corresponding register onto the bus. For example, to find the logic that makes $x_1 = 1$, we scan all register transfer statements in Table 5-6 and extract those statements that have *AR* as a source.

$$D_4T_4: \quad PC \leftarrow AR$$
$$D_5T_5: \quad PC \leftarrow AR$$

Therefore, the Boolean function for x_1 is

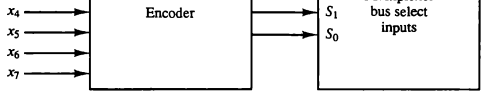
$$x_1 = D_4T_4 + D_5T_5$$

The data output from memory are selected for the bus when $x_7 = 1$ and $S_2S_1S_0 = 111$. The gate logic that generates x_7 must also be applied to the read input of memory. Therefore, the Boolean function for x_7 is the same as the one derived previously for the read operation.

$$x_7 = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

In a similar manner we can determine the gate logic for the other registers.

Figure 5-18 Encoder for bus selection inputs.



5-10 Design of Accumulator Logic

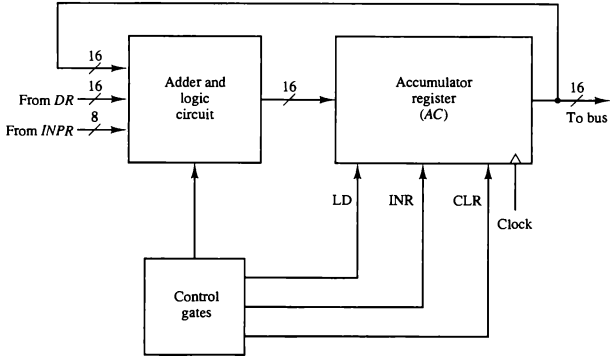
The circuits associated with the *AC* register are shown in Fig. 5-19. The adder and logic circuit has three sets of inputs. One set of 16 inputs comes from the outputs of *AC*. Another set of 16 inputs comes from the data register *DR*. A third set of eight inputs comes from the input register *INPR*. The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit.

In order to design the logic associated with *AC*, it is necessary to go over the register transfer statements in Table 5-6 and extract all the statements that change the content of *AC*.

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

From this list we can derive the control logic gates and the adder and logic circuit.

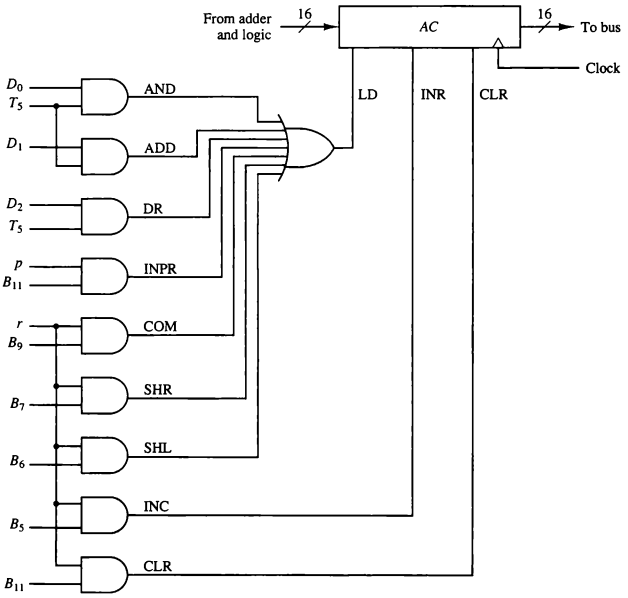
Figure 5-19 Circuits associated with AC.



Control of *AC* Register

The gate structure that controls the LD, INR, and CLR inputs of *AC* is shown in Fig. 5-20. The gate configuration is derived from the control functions in the list above. The control function for the clear microoperation is rB_{11}, where $r = D_7I'T_3$ and $B_{11} = IR(11)$. The output of the AND gate that generates this control function is connected to the CLR input of the register. Similarly, the output of the gate that implements the increment microoperation is connected to the INR input of the register. The other seven microoperations are generated in the adder and logic circuit and are loaded into *AC* at the proper time. The outputs of the gates for each control function is marked with a symbolic name. These outputs are used in the design of the adder and logic circuit.

Figure 5-20 Gate structure for controlling the LD, INR, and CLR of AC.



Adder and Logic Circuit

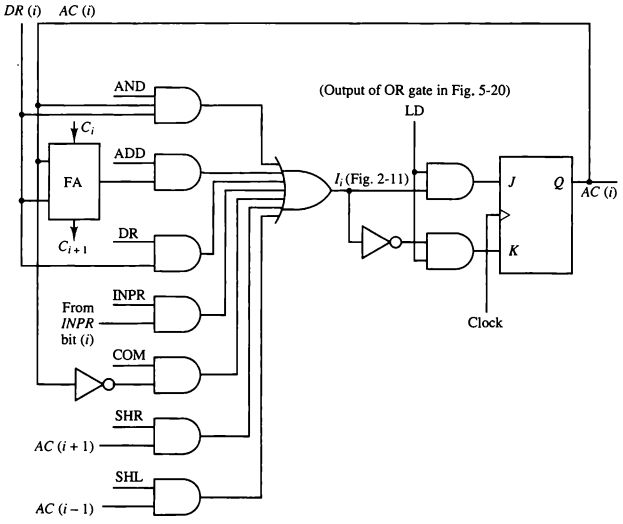
The adder and logic circuit can be subdivided into 16 stages, with each stage corresponding to one bit of *AC*. The internal construction of the register is as shown in Fig. 2-11. Looking back at that figure we note that each stage has a *JK* flip-flop, two OR gates, and two AND gates. The load (LD) input is connected to the inputs of the AND gates. Figure 5-21 shows one such *AC* register stage (with the OR gates removed). The input is labeled I_i and the output *AC*(*i*). When the LD input is enabled, the 16 inputs I_i for $i = 0, 1, 2, \ldots, 15$ are transferred to *AC* (0–15).

adder and logic circuit

One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full-adder (FA), as shown in Fig. 5-21. The inputs of the gates with symbolic names come from the outputs of gates marked with the same symbolic name in Fig. 5-20. For example, the input marked ADD in Fig. 5-21 is connected to the output marked ADD in Fig. 5-20.

The AND operation is achieved by ANDing *AC*(*i*) with the corresponding bit in the data register *DR*(*i*). The ADD operation is obtained using a binary adder similar to the one shown in Fig. 4-6. One stage of the adder uses a

Figure 5-21 One stage of adder and logic circuit.



full-adder with the corresponding input and output carries. The transfer from *INPR* to *AC* is only for bits 0 through 7. The complement microoperation is obtained by inverting the bit value in *AC*. The shift-right operation transfers the bit from $AC(i + 1)$, and the shift-left operation transfers the bit from $AC(i - 1)$. The complete adder and logic circuit consists of 16 stages connected together.

PROBLEMS

5-1. A computer uses a memory unit with 256K words of 32 bits each. A binary instruction code is stored in one word of memory. The instruction has four parts: an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part.

a. How many bits are there in the operation code, the register code part, and the address part?

b. Draw the instruction word format and indicate the number of bits in each part.

c. How many bits are there in the data and address inputs of the memory?

5-2. What is the difference between a direct and an indirect address instruction? How many references to memory are needed for each type of instruction to bring an operand into a processor register?

5-3. The following control inputs are active in the bus system shown in Fig. 5-4. For each case, specify the register transfer that will be executed during the next clock transition.

	S_2	S_1	S_0	LD of register	Memory	Adder
a.	1	1	1	*IR*	Read	—
b.	1	1	0	*PC*	—	—
c.	1	0	0	*DR*	Write	—
d.	0	0	0	*AC*	—	Add

5-4. The following register transfers are to be executed in the system of Fig. 5-4. For each transfer, specify: (1) the binary value that must be applied to bus select inputs S_2, S_1, and S_0; (2) the register whose LD control input must be active (if any); (3) a memory read or write operation (if needed); and (4) the operation in the adder and logic circuit (if any).

a. $AR \leftarrow PC$

b. $IR \leftarrow M[AR]$

c. $M[AR] \leftarrow TR$

d. $AC \leftarrow DR$, $DR \leftarrow AC$ (done simultaneously)

5-5. Explain why each of the following microoperations cannot be executed

during a single clock pulse in the system shown in Fig. 5-4. Specify a sequence of microoperations that will perform the operation.

a. $IR \leftarrow M[PC]$
b. $AC \leftarrow AC + TR$
c. $DR \leftarrow DR + AC$ (AC does not change)

5-6. Consider the instruction formats of the basic computer shown in Fig. 5-5 and the list of instructions given in Table 5-2. For each of the following 16-bit instructions, give the equivalent four-digit hexadecimal code and explain in your own words what it is that the instruction is going to perform.

a. 0001 0000 0010 0100
b. 1011 0001 0010 0100
c. 0111 0000 0010 0000

5-7. What are the two instructions needed in the basic computer in order to set the E flip-flop to 1?

5-8. Draw a timing diagram similar to Fig. 5-7 assuming that SC is cleared to 0 at time T_3 if control signal C_7 is active.

$$C_7T_3: \quad SC \leftarrow 0$$

C_7 is activated with the positive clock transition associated with T_1.

5-9. The content of AC in the basic computer is hexadecimal A937 and the initial value of E is 1. Determine the contents of AC, E, PC, AR, and IR in hexadecimal after the execution of the CLA instruction. Repeat 11 more times, starting from each one of the register-reference instructions. The initial value of PC is hexadecimal 021.

5-10. An instruction at address 021 in the basic computer has $I = 0$, an operation code of the AND instruction, and an address part equal to 083 (all numbers are in hexadecimal). The memory word at address 083 contains the operand B8F2 and the content of AC is A937. Go over the instruction cycle and determine the contents of the following registers at the end of the execute phase: PC, AR, DR, AC, and IR. Repeat the problem six more times starting with an operation code of another memory-reference instruction.

5-11. Show the contents in hexadecimal of registers PC, AR, DR, IR, and SC of the basic computer when an ISZ indirect instruction is fetched from memory and executed. The initial content of PC is 7FF. The content of memory at address 7FF is EA9F. The content of memory at address A9F is 0C35. The content of memory at address C35 is FFFF. Give the answer in a table with five columns, one for each register and a row for each timing signal. Show the contents of the registers after the positive transition of each clock pulse.

5-12. The content of PC in the basic computer is 3AF (all numbers are in hexadecimal). The content of AC is 7EC3. The content of memory at address 3AF is 932E. The content of memory at address 32E is 09AC. The content of memory at address 9AC is 8B9F.

a. What is the instruction that will be fetched and executed next?
b. Show the binary operation that will be performed in the AC when the instruction is executed.

c. Give the contents of registers *PC*, *AR*, *DR*, *AC*, and *IR* in hexadecimal and the values of *E*, *I*, and the sequence counter *SC* in binary at the end of the instruction cycle.

5-13. Assume that the first six memory-reference instructions in the basic computer listed in Table 5-4 are to be changed to the instructions specified in the following table. *EA* is the effective address that resides in *AR* during time T_4. Assume that the adder and logic circuit in Fig. 5-4 can perform the exclusive-OR operation $AC \leftarrow AC \oplus DR$. Assume further that the adder and logic circuit cannot perform subtraction directly. The subtraction must be done using the 2's complement of the subtrahend by complementing and incrementing *AC*. Give the sequence of register transfer statements needed to execute each of the listed instructions starting from timing T_4. Note that the value in *AC* should not change unless the instruction specifies a change in its content. You can use *TR* to store the content of *AC* temporary or you can exchange *DR* and *AC*.

Symbol	Opcode	Symbolic designation	Description in words
XOR	000	$AC \leftarrow AC \oplus M[EA]$	Exclusive-OR to *AC*
ADM	001	$M[EA] \leftarrow M[EA] + AC$	Add *AC* to memory
SUB	010	$AC \leftarrow AC - M[EA]$	Subtract memory from *AC*
XCH	011	$AC \leftarrow M[EA]$, $M[EA] \leftarrow AC$	Exchange *AC* and memory
SEQ	100	If $(M[EA] = AC)$ then $(PC \leftarrow PC + 1)$	Skip on equal
BPA	101	If $(AC > 0)$ then $(PC \leftarrow EA)$	Branch if *AC* positive and non-zero

5-14. Make the following changes to the basic computer.

1. Add a register to the bus system CTR (count register) to be selected with $S_2S_1S_0 = 000$.
2. Replace the ISZ instruction with an instruction that loads a number into CTR.

LDC Address $CTR \leftarrow M[Address]$

3. Add a register reference instruction ICSZ: Increment *CTR* and skip next instruction if zero. Discuss the advantage of this change.

5-15. The memory unit of the basic computer shown in Fig. 5-3 is to be changed to a 65,536 × 16 memory, requiring an address of 16 bits. The instruction format of a memory-reference instruction shown in Fig. 5-5(a) remains the same for $I = 1$ (indirect address) with the address part of the instruction residing in positions 0 through 11. But when $I = 0$ (direct address), the address of the instruction is given by the 16 bits in the next word following the instruction. Modify the microoperations during time T_2, T_3, (and T_4 if necessary) to conform with this configuration.

5-16. A computer uses a memory of 65,536 words with eight bits in each word. It has the following registers: *PC*, *AR*, *TR* (16 bits each), and *AC*, *DR*, *IR* (eight bits each). A memory-reference instruction consists of three words: an 8-bit operation-code (one word) and a 16-bit address (in the next two words). All operands are eight bits. There is no indirect bit.

a. Draw a block diagram of the computer showing the memory and registers as in Fig. 5-3. (Do not use a common bus).

b. Draw a diagram showing the placement in memory of a typical three-word instruction and the corresponding 8-bit operand.

c. List the sequence of microoperations for fetching a memory reference instruction and then placing the operand in *DR*. Start from timing signal T_0.

5-17. A digital computer has a memory unit with a capacity of 16,384 words, 40 bits per word. The instruction code format consists of six bits for the operation part and 14 bits for the address part (no indirect mode bit). Two instructions are packed in one memory word, and a 40-bit instruction register *IR* is available in the control unit. Formulate a procedure for fetching and executing instructions for this computer.

5-18. An output program resides in memory starting from address 2300. It is executed after the computer recognizes an interrupt when *FGO* becomes a 1 (while *IEN* = 1).

a. What instruction must be placed at address 1?

b. What must be the last two instructions of the output program?

5-19. The register transfer statements for a register *R* and the memory in a computer are as follows (the X's are control functions that occur at random):

$X_3' X_1$:	$R \leftarrow M[AR]$	Read memory word into *R*
$X_1' X_2$:	$R \leftarrow AC$	Transfer *AC* to *R*
$X_1' X_3$:	$M[AR] \leftarrow R$	Write *R* to memory

The memory has data inputs, data outputs, address inputs, and control inputs to read and write as in Fig. 2-12. Draw the hardware implementation of *R* and the memory in block diagram form. Show how the control functions X_1 through X_3 select the load control input of *R*, the select inputs of multiplexers that you include in the diagram, and the read and write inputs of the memory.

5-20. The operations to be performed with a flip-flop *F* (not used in the basic computer) are specified by the following register transfer statements:

xT_3:	$F \leftarrow 1$	Set *F* to 1
yT_1:	$F \leftarrow 0$	Clear *F* to 0
zT_2:	$F \leftarrow \overline{F}$	Complement *F*
wT_5:	$F \leftarrow G$	Transfer value of *G* to *F*

Otherwise, the content of *F* must not change. Draw the logic diagram showing the connections of the gates that form the control functions and the inputs of flip-flop *F*. Use a *JK* flip-flop and minimize the number of gates.

5-21. Derive the control gates associated with the program counter *PC* in the basic computer.

5-22. Derive the control gates for the write input of the memory in the basic computer.

5-23. Show the complete logic of the interrupt flip-flops *R* in the basic computer. Use a *JK* flip-flop and minimize the number of gates.

5-24. Derive the Boolean logic expression for x_2 (see Table 5-7). Show that x_2 can be generated with one AND gate and one OR gate.

5-25. Derive the Boolean expression for the gate structure that clears the sequence counter *SC* to 0. Draw the logic diagram of the gates and show how the output is connected to the INR and CLR inputs of *SC* (see Fig. 5-6). Minimize the number of gates.

REFERENCES

1. Bell, C. G., J. C. Mudge, and J. E. McNamara, Computer Engineering. Bedford, MA: Digital Press, 1980.
2. Booth, T. L., *Introduction to Computer Engineering*, 3rd ed. New York: John Wiley, 1984.
3. Gibson, G. A., *Computer Systems Concepts and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
4. Gray, N. A. B., *Introduction to Computer Systems*. Englewood Cliffs, NJ: Prentice Hall, 1987.
5. Hill, F. J., and G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 3rd ed. New York: John Wiley, 1987.
6. Lewin, M. H. *Logic Design and Computer Organization*. Reading, MA: Addison-Wesley, 1983.
7. Mano, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.
8. Patterson, D. A. and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1990.
9. Prosser, F. P., and D. E. Winkel, *The Art of Digital Design*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1987.
10. Shiva, S. G., *Computer Design and Architecture*, 2nd ed. New York: HarperCollins Publishers, 1991.

CHAPTER SIX

Programming the Basic Computer

IN THIS CHAPTER

6-1 Introduction

A total computer system includes both *hardware* and *software*. Hardware consists of the physical components and all associated equipment. Software refers to the programs that are written for the computer. It is possible to be familiar with various aspects of computer software without being concerned with details of how the computer hardware operates. It is also possible to design parts of the hardware without a knowledge of its software capabilities. However, those concerned with computer architecture should have a knowledge of both hardware and software because the two branches influence each other.

Writing a program for a computer consists of specifying, directly or indirectly, a sequence of machine instructions. Machine instructions inside the computer form a binary pattern which is difficult, if not impossible, for people to work with and understand. It is preferable to write programs with the more familiar symbols of the alphanumeric character set. As a consequence, there is a need for translating user-oriented symbolic programs into binary programs recognized by the hardware.

A program written by a user may be either dependent or independent of

the physical computer that runs his program. For example, a program written in standard Fortran is machine independent because most computers provide a translator program that converts the standard Fortran program to the binary code of the computer available in the particular installation. But the translator program itself is machine dependent because it must translate the Fortran program to the binary code recognized by the hardware of the particular computer used.

This chapter introduces some elementary programming concepts and shows their relation to the hardware representation of instructions. The first part presents the basic operation and structure of a program that translates a user's symbolic program into an equivalent binary program. The discussion emphasizes the important concepts of the translator rather than the details of actually producing the program itself. The usefulness of various machine instructions is then demonstrated by means of several basic programming examples.

instruction set

The instruction set of the basic computer, whose hardware organization was explored in Chap. 5, is used in this chapter to illustrate many of the techniques commonly used to program a computer. In this way it is possible to explore the relationship between a program and the hardware operations that execute the instructions.

The 25 instructions of the basic computer are repeated in Table 6-1 to provide an easy reference for the programming examples that follow. Each instruction is assigned a three-letter symbol to facilitate writing symbolic programs. The first seven instructions are memory-reference instructions and the other 18 are register-reference and input–output instructions. A memory-reference instruction has three parts: a mode bit, an operation code of three bits, and a 12-bit address. The first hexadecimal digit of a memory-reference instruction includes the mode bit and the operation code. The other three digits specify the address. In an indirect address instruction the mode bit is 1 and the first hexadecimal digit ranges in value from 8 to *E*. In a direct mode, the range is from 0 to 6. The other 18 instructions have a 16-bit operation code. The code for each instruction is listed as a four-digit hexadecimal number. The first digit of a register-reference instruction is always 7. The first digit of an input–output instruction is always *F*. The symbol *m* used in the description column denotes the effective address. The letter *M* refers to the memory word (operand) found at the effective address.

6-2 Machine Language

A program is a list of instructions or statements for directing the computer to perform a required data-processing task. There are various types of programming languages that one may *write* for a computer, but the computer can *execute* programs only when they are represented internally in binary form. Programs

TABLE 6-1 Computer Instructions

Symbol	Hexadecimal code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to $m + 1$
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC,
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

written in any other language must be translated to the binary representation of instructions before they can be executed by the computer. Programs written for a computer may be in one of the following categories:

1. *Binary code*. This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.
2. *Octal or hexadecimal code*. This is an equivalent translation of the binary code to octal or hexadecimal representation.
3. *Symbolic code*. The user employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code. Each symbolic instruction can be translated into one binary coded instruction. This translation is done by a special program called an *assembler*. Because an assembler translates the sym-

assembly language

bols, this type of symbolic program is referred to as an *assembly language* program.

4. *High-level programming languages*. These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior. An example of a high-level programming language is *Fortran*. It employs problem-oriented symbols and formats. The program is written in a sequence of statements in a form that people prefer to think in when solving a problem. However, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer. The program that translates a high-level language program to binary is called a *compiler*.

machine language

Strictly speaking, a *machine language* program is a binary program of category 1. Because of the simple equivalency between binary and octal or hexadecimal representation, it is customary to refer to category 2 as machine language. Because of the one-to-one relationship between a symbolic instruction and its binary equivalent, an assembly language is considered to be a machine-level language.

We now use the basic computer to illustrate the relation between binary and assembly languages. Consider the binary program listed in Table 6-2. The first column gives the memory location (in binary) of each instruction or operand. The second column lists the binary content of these memory locations. (The *location* is the address of the memory word where the instruction is stored. It is important to differentiate it from the address part of the instruction itself.) The program can be stored in the indicated portion of memory, and then executed by the computer starting from address 0. The hardware of the computer will execute these instructions and perform the intended task. However, a person looking at this program will have a difficult time understanding what is to be achieved when this program is executed. Nevertheless, the computer hardware recognizes *only* this type of instruction code.

TABLE 6-2 Binary Program to Add Two Numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

TABLE 6-3 Hexadecimal Program to Add Two Numbers

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

Writing 16 bits for each instruction is tedious because there are too many digits. We can reduce the number of digits per instruction if we write the octal equivalent of the binary code. This will require six digits per instruction. On the other hand, we can reduce each instruction to four digits if we write the equivalent hexadecimal code as shown in Table 6-3. The hexadecimal representation is convenient to use; however, one must realize that each hexadecimal digit must be converted to its equivalent 4-bit number when the program is entered into the computer. The advantage of writing binary programs in equivalent octal or hexadecimal form should be evident from this example.

hexadecimal code

The program in Table 6-4 uses the symbolic names of instructions (listed in Table 6-1) instead of their binary or hexadecimal equivalent. The address parts of memory-reference instructions, as well as operands, remain in their hexadecimal value. Note that location 005 has a negative operand because the sign bit in the leftmost position is 1. The inclusion of a column for comments provides some means for explaining the function of each instruction. Symbolic programs are easier to handle, and as a consequence, it is preferable to write programs with symbols. These symbols can be converted to their binary code equivalent to produce the binary program.

We can go one step further and replace each hexadecimal address by a

TABLE 6-4 Program with Symbolic Operation Codes

Location	Instruction	Comments
000	LDA 004	Load first operand into *AC*
001	ADD 005	Add second operand to *AC*
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	First operand
005	FFE9	Second operand (negative)
006	0000	Store sum here

TABLE 6-5 Assembly Language Program to Add Two Numbers

```
      ORG 0       /Origin of program is location 0
      LDA A       /Load operand from location A
      ADD B       /Add operand from location B
      STA C       /Store sum in location C
      HLT         /Halt computer
A,    DEC 83      /Decimal operand
B,    DEC -23     /Decimal operand
C,    DEC 0       /Sum stored in location C
      END         /End of symbolic program
```

symbolic address and each hexadecimal operand by a decimal operand. This is convenient because one usually does not know exactly the numeric memory location of operands while writing a program. If the operands are placed in memory following the instructions, and if the length of the program is not known in advance, the numerical location of operands is not known until the end of the program is reached. In addition, decimal numbers are more familiar than their hexadecimal equivalents.

The program in Table 6-5 is the assembly-language program for adding two numbers. The symbol ORG followed by a number is not a machine instruction. Its purpose is to specify an *origin*, that is, the memory location of the next instruction below it. The next three lines have symbolic addresses. Their value is specified by their being present as a label in the first column. Decimal operands are specified following the symbol DEC. The numbers may be positive or negative, but if negative, they must be converted to binary in the signed-2's complement representation. The last line has the symbol END indicating the end of the program. The symbols ORG, DEC, and END, called *pseudoinstructions*, are defined in the next section. Note that all comments are preceded by a slash.

The equivalent Fortran program for adding two integer numbers is listed in Table 6-6. The two values for *A* and *B* may be specified by an input statement or by a data statement. The arithmetic operation for the two numbers is specified by one simple statement. The translation of this Fortran program into a binary program consists of assigning three memory locations, one each for the augend, addend, and sum, and then deriving the sequence of binary

TABLE 6-6 Fortran Program to Add Two Numbers

```
INTEGER A, B, C
DATA A,83  B,-23
C = A + B
END
```

instructions that form the sum. Thus a compiler program translates the symbols of the Fortran program into the binary values listed in the program of Table 6-2.

6-3 Assembly Language

A programming language is defined by a set of rules. Users must conform with all format rules of the language if they want their programs to be translated correctly. Almost every commercial computer has its own particular assembly language. The rules for writing assembly language programs are documented and published in manuals which are usually available from the computer manufacturer.

The basic unit of an assembly language program is a line of code. The specific language is defined by a set of rules that specify the symbols that can be used and how they may be combined to form a line of code. We will now formulate the rules of an assembly language for writing symbolic programs for the basic computer.

Rules of the Language

Each line of an assembly language program is arranged in three columns called fields. The fields specify the following information.

1. The *label* field may be empty or it may specify a symbolic address.
2. The *instruction* field specifies a machine instruction or a pseudoinstruction.
3. The *comment* field may be empty or it may include a comment.

symbolic address

A symbolic address consists of one, two, or three, but not more than three alphanumeric characters. The first character must be a letter; the next two may be letters or numerals. The symbol can be chosen arbitrarily by the programmer. A symbolic address in the label field is terminated by a comma so that it will be recognized as a label by the assembler.

The instruction field in an assembly language program may specify one of the following items:

1. A memory-reference instruction (MRI)
2. A register-reference or input–output instruction (non-MRI)
3. A pseudoinstruction with or without an operand

A memory-reference instruction occupies two or three symbols separated by spaces. The first must be a three-letter symbol defining an MRI operation

code from Table 6-1. The second is a symbolic address. The third symbol, which may or may not be present, is the letter I. If I is missing, the line denotes a direct address instruction. The presence of the symbol I denotes an indirect address instruction.

A non-MRI is defined as an instruction that does not have an address part. A non-MRI is recognized in the instruction field of a program by any one of the three-letter symbols listed in Table 6-1 for the register-reference and input–output instructions.

The following is an illustration of the symbols that may be placed in the instruction field of a program.

```
CLA          non-MRI
ADD OPR      direct address MRI
ADD PTR I    indirect address MRI
```

The first three-letter symbol in each line must be one of the instruction symbols of the computer and must be listed in Table 6-1. A memory-reference instruction, such as ADD, must be followed by a symbolic address. The letter I may or may not be present.

A symbolic address in the instruction field specifies the memory location of an operand. This location must be defined somewhere in the program by appearing again as a label in the first column. To be able to translate an assembly language program to a binary program, it is absolutely necessary that each symbolic address that is mentioned in the instruction field *must* occur again in the label field.

pseudoinstruction

A pseudoinstruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation. Four pseudoinstructions that are recognized by the assembler are listed in Table 6-7. (Other assembly language programs recognize many more pseudoinstructions.) The ORG (origin) pseudoinstruction informs the assembler that the instruction or operand in the following line is to be placed in a memory location specified by the number next to ORG. It is possible to use ORG more than once in a program to specify more than one segment of memory. The END symbol

TABLE 6-7 Definition of Pseudoinstructions

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

is placed at the end of the program to inform the assembler that the program is terminated. The other two pseudoinstructions specify the radix of the operand and tell the assembler how to convert the listed number to a binary number.

The third field in a program is reserved for comments. A line of code may or may not have a comment, but if it has, it must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are useful for explaining the program and are helpful in understanding the step-by-step procedure taken by the program. Comments are inserted for explanation purposes only and are neglected during the binary translation process.

An Example

The program of Table 6-8 is an example of an assembly language program. The first line has the pseudoinstruction ORG to define the origin of the program at memory location $(100)_{16}$. The next six lines define machine instructions, and the last four have pseudoinstructions. Three symbolic addresses have been used and each is listed in column 1 as a label and in column 2 as an address of a memory-reference instruction. Three of the pseudoinstructions specify operands, and the last one signifies the END of the program.

When the program is translated into binary code and executed by the computer it will perform a subtraction between two numbers. The subtraction is performed by adding the minuend to the 2's complement of the subtrahend. The subtrahend is a negative number. It is converted into a binary number in signed-2's complement representation because we dictate that all negative numbers be in their 2's complement form. When the 2's complement of the subtrahend is taken (by complementing and incrementing the *AC*), −23 converts to +23 and the difference is 83 + (2's complement of −23) = 83 + 23 = 106.

TABLE 6-8 Assembly Language Program to Subtract Two Numbers

	ORG 100	/Origin of program is location 100
	LDA SUB	/Load subtrahend to *AC*
	CMA	/Complement *AC*
	INC	/Increment *AC*
	ADD MIN	/Add minuend to *AC*
	STA DIF	/Store difference
	HLT	/Halt computer
MIN,	DEC 83	/Minuend
SUB,	DEC −23	/Subtrahend
DIF,	HEX 0	/Difference stored here
	END	/End of symbolic program

Translation to Binary

assembler

The translation of the symbolic program into binary is done by a special program called an *assembler*. The tasks performed by the assembler will be better understood if we first perform the translation on paper. The translation of the symbolic program of Table 6-8 into an equivalent binary code may be done by scanning the program and replacing the symbols by their machine code binary equivalent. Starting from the first line, we encounter an ORG pseudoinstruction. This tells us to start the binary program from hexadecimal location 100. The second line has two symbols. It must be a memory-reference instruction to be placed in location 100. Since the letter I is missing, the first bit of the instruction code must be 0. The symbolic name of the operation is LDA. Checking Table 6-1 we find that the first hexadecimal digit of the instruction should be 2. The binary value of the address part must be obtained from the address symbol SUB. We scan the label column and find this symbol in line 9. To determine its hexadecimal value we note that line 2 contains an instruction for location 100 and every other line specifies a machine instruction or an operand for sequential memory locations. Counting lines, we find that label SUB in line 9 corresponds to memory location 107. So the hexadecimal address of the instruction LDA must be 107. When the two parts of the instruction are assembled, we obtain the hexadecimal code 2107. The other lines representing machine instructions are translated in a similar fashion and their hexadecimal code is listed in Table 6-9.

Two lines in the symbolic program specify decimal operands with the pseudoinstruction DEC. A third specifies a zero by means of a HEX pseudoinstruction (DEC could be used as well). Decimal 83 is converted to binary and placed in location 106 in its hexadecimal equivalent. Decimal −23 is a negative number and must be converted into binary in signed-2's complement form.

TABLE 6-9 Listing of Translated Program of Table 6-8

Hexadecimal code			
Location	Content		Symbolic program
			ORG 100
100	2107		LDA SUB
101	7200		CMA
102	7020		INC
103	1106		ADD MIN
104	3108		STA DIF
105	7001		HLT
106	0053	MIN,	DEC 83
107	FFE9	SUB,	DEC −23
108	0000	DIF,	HEX 0
			END

The hexadecimal equivalent of the binary number is placed in location 107. The END symbol signals the end of the symbolic program telling us that there are no more lines to translate.

address symbol table

The translation process can be simplified if we scan the entire symbolic program twice. No translation is done during the first scan. We merely assign a memory location to each machine instruction and operand. The location assignment will define the address value of labels and facilitate the translation process during the second scan. Thus in Table 6-9, we assign location 100 to the first instruction after ORG. We then assign sequential locations for each line of code that has a machine instruction or operand up to the end of the program. (ORG and END are not assigned a numerical location because they do not represent an instruction or an operand.) When the first scan is completed, we associate with each label its location number and form a table that defines the hexadecimal value of each symbolic address. For this program, the address symbol table is as follows:

Address symbol	Hexadecimal address
MIN	106
SUB	107
DIF	108

During the second scan of the symbolic program we refer to the address symbol table to determine the address value of a memory-reference instruction. For example, the line of code LDA SUB is translated during the second scan by getting the hexadecimal value of LDA from Table 6-1 and the hexadecimal value of SUB from the address-symbol table listed above. We then assemble the two parts into a four-digit hexadecimal instruction. The hexadecimal code can be easily converted to binary if we wish to know exactly how this program resides in computer memory.

When the translation from symbols to binary is done by an assembler program, the first scan is called the *first pass*, and the second is called the *second pass*.

6-4 The Assembler

An assembler is a program that accepts a symbolic language program and produces its binary machine language equivalent. The input symbolic program is called the *source program* and the resulting binary program is called the *object program*. The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

Representation of Symbolic Program in Memory

Prior to starting the assembly process, the symbolic program must be stored in memory. The user types the symbolic program on a terminal. A loader program is used to input the characters of the symbolic program into memory. Since the program consists of symbols, its representation in memory must use an alphanumeric character code. In the basic computer, each character is represented by an 8-bit code. The high-order bit is always 0 and the other seven bits are as specified by ASCII. The hexadecimal equivalent of the character set is listed in Table 6-10. Each character is assigned two hexadecimal digits which can be easily converted to their equivalent 8-bit code. The last entry in the table does not print a character but is associated with the physical movement of the cursor in the terminal. The code for CR is produced when the return key is depressed. This causes the "carriage" to return to its initial position to start typing a new line. The assembler recognizes a CR code as the end of a line of code.

line of code

A line of code is stored in consecutive memory locations with two characters in each location. Two characters can be stored in each word since a memory word has a capacity of 16 bits. A label symbol is terminated with a comma. Operation and address symbols are terminated with a space and the end of the line is recognized by the CR code. For example, the following line of code:

```
PL3,    LDA SUB I
```

TABLE 6-10 Hexadecimal Character Code

Character	Code	Character	Code	Character	Code	
A	41	Q	51	6	36	
B	42	R	52	7	37	
C	43	S	53	8	38	
D	44	T	54	9	39	
E	45	U	55	space	20	
F	46	V	56	(	28	
G	47	W	57	)	29	
H	48	X	58	*	2A	
I	49	Y	59	+	2B	
J	4A	Z	5A	,	2C	
K	4B	0	30	−	2D	
L	4C	1	31	.	2E	
M	4D	2	32	/	2F	
N	4E	3	33	=	3D	
O	4F	4	34	CR	0D	(carriage
P	50	5	35			return)

TABLE 6-11 Computer Representation of the Line of Code: PL3, LDA SUB I

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3 ,	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

is stored in seven consecutive memory locations, as shown in Table 6-11. The label PL3 occupies two words and is terminated by the code for comma (2C). The instruction field in the line of code may have one or more symbols. Each symbol is terminated by the code for space (20) except for the last symbol, which is terminated by the code of carriage return (0D). If the line of code has a comment, the assembler recognizes it by the code for a slash (2F). The assembler neglects all characters in the comment field and keeps checking for a CR code. When this code is encountered, it replaces the space code after the last symbol in the line of code.

The input for the assembler program is the user's symbolic language program in ASCII. This input is scanned by the assembler twice to produce the equivalent binary program. The binary program constitutes the output generated by the assembler. We will now describe briefly the major tasks that must be performed by the assembler during the translation process.

First Pass

A two-pass assembler scans the entire symbolic program twice. During the first pass, it generates a table that correlates all user-defined address symbols with their binary equivalent value. The binary translation is done during the second pass. To keep track of the location of instructions, the assembler uses a memory word called a *location counter* (abbreviated LC). The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed. The ORG pseudoinstruction initializes the location counter to the value of the first location. Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code. To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.

location counter (LC)

The tasks performed by the assembler during the first pass are described in the flowchart of Fig. 6-1. LC is initially set to 0. A line of symbolic code is analyzed to determine if it has a label (by the presence of a comma). If the line

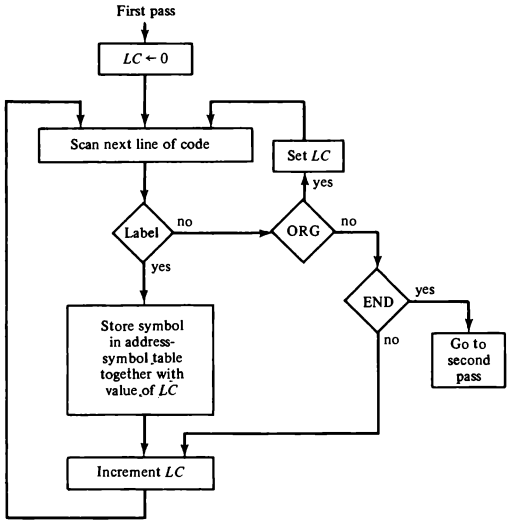


Figure 6-1 Flowchart for first pass of assembler.

of code has no label, the assembler checks the symbol in the instruction field. If it contains an ORG pseudoinstruction, the assembler sets LC to the number that follows ORG and goes back to process the next line. If the line has an END pseudoinstruction, the assembler terminates the first pass and goes to the second pass. (Note that a line with ORG or END should not have a label.) If the line of code contains a label, it is stored in the address symbol table together with its binary equivalent number specified by the content of LC. Nothing is stored in the table if no label is encountered. LC is then incremented by 1 and a new line of code is processed.

For the program of Table 6-8, the assembler generates the address symbol table listed in Table 6-12. Each label symbol is stored in two memory locations and is terminated by a comma. If the label contains less than three characters, the memory locations are filled with the code for space. The value found in LC while the line was processed is stored in the next sequential memory location. The program has three symbolic addresses: MIN, SUB, and DIF. These symbols represent 12-bit addresses equivalent to hexadecimal 106, 107, and 108,

TABLE 6-12 Address Symbol Table for Program in Table 6-8

Memory word	Symbol or (LC)*	Hexadecimal code	Binary representation
1	M I	4D 49	0100 1101 0100 1001
2	N ,	4E 2C	0100 1110 0010 1100
3	(LC)	01 06	0000 0001 0000 0110
4	S U	53 55	0101 0011 0101 0101
5	B ,	42 2C	0100 0010 0010 1100
6	(LC)	01 07	0000 0001 0000 0111
7	D I	44 49	0100 0100 0100 1001
8	F ,	46 2C	0100 0110 0010 1100
9	(LC)	01 08	0000 0001 0000 1000

* (LC) designates content of location counter.

respectively. The address symbol table occupies three words for each label symbol encountered and constitutes the output data that the assembler generates during the first pass.

Second Pass

table-lookup

Machine instructions are translated during the second pass by means of table-lookup procedures. A table-lookup procedure is a search of table entries to determine whether a specific item matches one of the items stored in the table. The assembler uses four tables. Any symbol that is encountered in the program must be available as an entry in one of these tables; otherwise, the symbol cannot be interpreted. We assign the following names to the four tables:

1. Pseudoinstruction table.
2. MRI table.
3. Non-MRI table.
4. Address symbol table.

The entries of the pseudoinstruction table are the four symbols ORG, END, DEC, and HEX. Each entry refers the assembler to a subroutine that processes the pseudoinstruction when encountered in the program. The MRI table contains the seven symbols of the memory-reference instructions and their 3-bit operation code equivalent. The non-MRI table contains the symbols for the 18 register-reference and input–output instructions and their 16-bit binary code equivalent. The address symbol table is generated during the first pass of the assembly process. The assembler searches these tables to find the symbol that it is currently processing in order to determine its binary value.

The tasks performed by the assembler during the second pass are de-

scribed in the flowchart of Fig. 6-2. LC is initially set to 0. Lines of code are then analyzed one at a time. Labels are neglected during the second pass, so the assembler goes immediately to the instruction field and proceeds to check the first symbol encountered. It first checks the pseudoinstruction table. A match with ORG sends the assembler to a subroutine that sets LC to an initial value. A match with END terminates the translation process. An operand pseudoinstruction causes a conversion of the operand into binary. This operand is placed in the memory location specified by the content of LC. The location counter is then incremented by 1 and the assembler continues to analyze the next line of code.

If the symbol encountered is not a pseudoinstruction, the assembler refers to the MRI table. If the symbol is not found in this table, the assembler refers to the non-MRI table. A symbol found in the non-MRI table corresponds to a register reference or input–output instruction. The assembler stores the 16-bit instruction code into the memory word specified by LC. The location counter is incremented and a new line analyzed.

When a symbol is found in the MRI table, the assembler extracts its equivalent 3-bit code and inserts it in bits 2 through 4 of a word. A memory reference instruction is specified by two or three symbols. The second symbol is a symbolic address and the third, which may or may not be present, is the letter I. The symbolic address is converted to binary by searching the address symbol table. The first bit of the instruction is set to 0 or 1, depending on whether the letter I is absent or present. The three parts of the binary instruction code are assembled and then stored in the memory location specified by the content of LC. The location counter is incremented and the assembler continues to process the next line.

error diagnostics

One important task of an assembler is to check for possible errors in the symbolic program. This is called *error diagnostics*. One such error may be an invalid machine code symbol which is detected by its being absent in the MRI and non-MRI tables. The assembler cannot translate such a symbol because it does not know its binary equivalent value. In such a case, the assembler prints an error message to inform the programmer that his symbolic program has an error at a specific line of code. Another possible error may occur if the program has a symbolic address that did not appear also as a label. The assembler cannot translate the line of code properly because the binary equivalent of the symbol will not be found in the address symbol table generated during the first pass. Other errors may occur and a practical assembler should detect all such errors and print an error message for each.

It should be emphasized that a practical assembler is much more complicated than the one explained here. Most computers give the programmer more flexibility in writing assembly language programs. For example, the user may be allowed to use either a number or a symbol to specify an address. Many assemblers allow the user to specify an address by an arithmetic expression. Many more pseudoinstructions may be specified to facilitate the programming

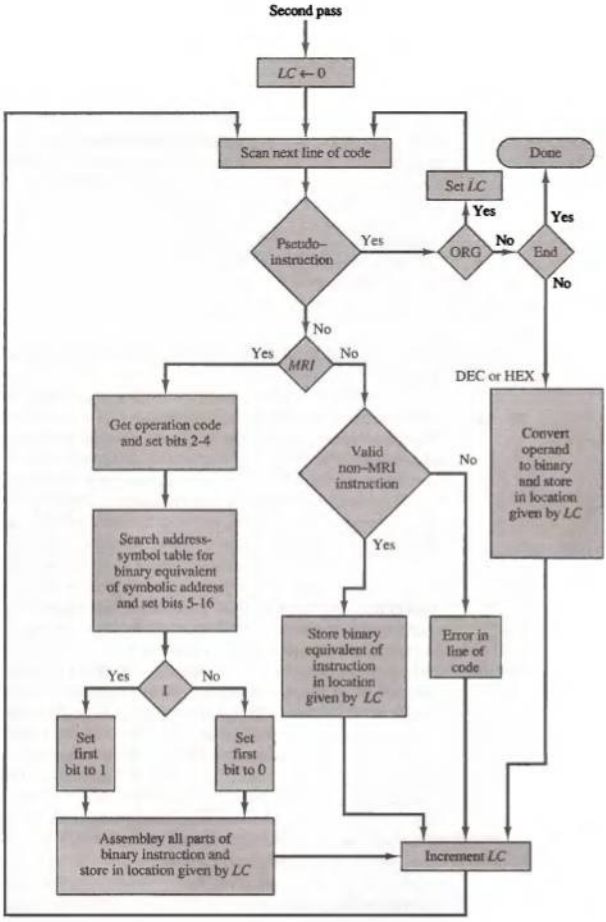


Figure 6-2 Flowchart for second pass of assembler.

task. As the assembly language becomes more sophisticated, the assembler becomes more complicated.

6-5 Program Loops

A program loop is a sequence of instructions that are executed many times, each time with a different set of data. Program loops are specified in Fortran by a DO statement. The following is an example of a Fortran program that forms the sum of 100 integer numbers.

```
      DIMENSION A(100)
      INTEGER SUM, A
      SUM = 0
      DO 3 J = 1, 100
    3 SUM = SUM + A(J)
```

Statement number 3 is executed 100 times, each time with a different operand A(J) for J = 1, 2, . . . , 100.

A system program that translates a program written in a high-level programming language such as the above to a machine language program is called a *compiler*. A compiler is a more complicated program than an assembler and requires knowledge of systems programming to fully understand its operation. Nevertheless, we can demonstrate the basic functions of a compiler by going through the process of translating the program above to an assembly language program. A compiler may use an assembly language as an intermediate step in the translation or may translate the program directly to binary.

compiler

The first statement in the Fortran program is a DIMENSION statement. This statement instructs the compiler to reserve 100 words of memory for 100 operands. The value of the operands is determined from an input statement (not listed in the program). The second statement informs the compiler that the numbers are integers. If they were of the *real* type, the compiler would have to reserve locations for floating-point numbers and generate instructions that perform the subsequent arithmetic with floating-point data. These two statements are nonexecutable and are similar to the pseudoinstructions in an assembly language. Suppose that the compiler reserves locations $(150)_{16}$ to $(1B3)_{16}$ for the 100 operands. These reserved memory words are listed in lines 19 to 118 in the translated program of Table 6-13. This is done by the ORG pseudoinstruction in line 18, which specifies the origin of the operands. The first and last operands are listed with a specific decimal number, although these values are not known during compilation. The compiler just reserves the data space in memory and the values are inserted later when an input data statement is executed. The line numbers in the symbolic program are for reference only and are not part of the translated symbolic program.

The indexing of the DO statement is translated into the instructions in

TABLE 6-13 Symbolic Program to Add 100 Numbers

Line			
1		ORG 100	/Origin of program is HEX 100
2		LDA ADS	/Load first address of operands
3		STA PTR	/Store in pointer
4		LDA NBR	/Load minus 100
5		STA CTR	/Store in counter
6		CLA	/Clear accumulator
7	LOP,	ADD PTR I	/Add an operand to *AC*
8		ISZ PTR	/Increment pointer
9		ISZ CTR	/Increment counter
10		BUN LOP	/Repeat loop again
11		STA SUM	/Store sum
12		HLT	/Halt
13	ADS,	HEX 150	/First address of operands
14	PTR,	HEX 0	/This location reserved for a pointer
15	NBR,	DEC −100	/Constant to initialized counter
16	CTR,	HEX 0	/This location reserved for a counter
17	SUM,	HEX 0	/Sum is stored here
18		ORG 150	/Origin of operands is HEX 150
19		DEC 75	/First operand
•			
•			
•			
118		DEC 23	/Last operand
119		END	/End of symbolic program

lines 2 through 5 and the constants in lines 13 through 16. The address of the first operand (150) is stored in location ADS in line 13. The number of times that Fortran statement number 3 must be executed is 100. So −100 is stored in location NBR. The compiler then generates the instructions in lines 2 through 5 to initialize the program loop. The address of the first operand is transferred to location PTR. This corresponds to setting A(J) to A(1). The number −100 is then transferred to location CTR. This location acts as a counter with its content incremented by one every time the program loop is executed. When the value of the counter reaches zero, the 100 operations will be completed and the program will exit from the loop.

Some compilers will translate the statement SUM = 0 into a machine instruction that initializes location SUM to zero. A reference to this location is then made every time Fortran statement number 3 is executed. A more intelligent compiler will realize that the sum can be formed in the accumulator and only the final result stored in location SUM. This compiler will produce an instruction in line 6 to clear the *AC*. It will also reserve a memory location

symbolized by SUM (in line 17) for storing the value of this variable at the termination of the loop.

The program loop specified by the DO statement is translated to the sequence of instructions listed in lines 7 through 10. Line 7 specifies an indirect ADD instruction because it has the symbol I. The address of the current operand is stored in location PTR. When this location is addressed indirectly the computer takes the content of PTR to be the address of the operand. As a result, the operand in location 150 is added to the accumulator. Location PTR is then incremented with the ISZ instruction in line 8, so its value changes to the value of the address of the next sequential operand. Location CTR is incremented in line 9, and if it is not zero, the computer does not skip the next instruction. The next instruction is a branch (BUN) instruction to the beginning of the loop, so the computer returns to repeat the loop once again. When location CTR reaches zero (after the loop is executed 100 times), the next instruction is skipped and the computer executes the instructions in lines 11 and 12. The sum formed in the accumulator is stored in SUM and the computer halts. The halt instruction is inserted here for clarity; actually, the program will branch to a location where it will continue to execute the rest of the program or branch to the beginning of another program. Note that ISZ in line 8 is used merely to add 1 to the address pointer PTR. Since the address is a positive number, a skip will never occur.

The program of Table 6-13 introduces the idea of a pointer and a counter which can be used, together with the indirect address operation, to form a program loop. The pointer points to the address of the current operand and the counter counts the number of times that the program loop is executed. In this example we use two memory locations for these functions. In computers with more than one processor register, it is possible to use one processor register as a pointer, another as a counter, and a third as an accumulator. When processor registers are used as pointers and counters they are called *index registers*. Index registers are discussed in Sec. 8-5.

pointer
counter

6-6 Programming Arithmetic and Logic Operations

The number of instructions available in a computer may be a few hundred in a large system or a few dozen in a small one. Some computers perform a given operation with one machine instruction; others may require a large number of machine instructions to perform the same operation. As an illustration, consider the four basic arithmetic operations. Some computers have machine instructions to add, subtract, multiply, and divide. Others, such as the basic computer, have only one arithmetic instruction, such as ADD. Operations not included in the set of machine instructions must be implemented by a program.

We have shown in Table 6-8 a program for subtracting two numbers. Programs for the other arithmetic operations can be developed in a similar fashion.

Operations that are implemented in a computer with one machine instruction are said to be implemented by hardware. Operations implemented by a set of instructions that constitute a program are said to be implemented by software. Some computers provide an extensive set of hardware instructions designed to speed up common tasks. Others contain a smaller set of hardware instructions and depend more heavily on the software implementation of many operations. Hardware implementation is more costly because of the additional circuits needed to implement the operation. Software implementation results in long programs both in number of instructions and in execution time.

This section demonstrates the software implementation of a few arithmetic and logic operations. Programs can be developed for any arithmetic operation and not only for fixed-point binary data but for decimal and floating-point data as well. The hardware implementation of arithmetic operations is carried out in Chap. 10.

Multiplication Program

We now develop a program for multiplying two numbers. To simplify the program, we neglect the sign bit and assume positive numbers. We also assume that the two binary numbers have no more than eight significant bits so their product cannot exceed the word capacity of 16 bits. It is possible to modify the program to take care of the signs or use 16-bit numbers. However, the product may be up to 31 bits in length and will occupy two words of memory.

The program for multiplying two numbers is based on the procedure we use to multiply numbers with paper and pencil. As shown in the numerical example of Fig. 6-3, the multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X as many times as there are 1's in Y, provided that the value of X is shifted left from one line to the next. Since the computer can add only two numbers at a time, we reserve a memory location, denoted by P, to store intermediate sums. The intermediate sums are called partial products since they hold a partial product until all numbers are added. As shown in the numerical example under P, the partial product starts with zero. The multiplicand X is added to the content of P for each bit of the multiplier Y that is 1. The value of X is shifted left after checking each bit of the multiplier. The final value in P forms the product. The numerical example has numbers with four significant bits. When multiplied, the product contains eight significant bits. The computer can use numbers with eight significant bits to produce a product of up to 16 bits.

The flowchart of Fig. 6-3 shows the step-by-step procedure for program-

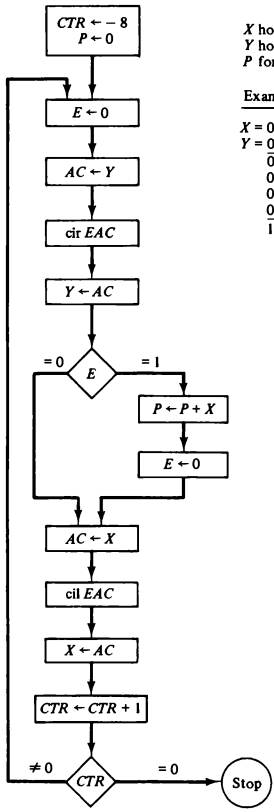


Figure 6-3 Flowchart for multiplication program.

ming the multiplication operation. The program has a loop that is traversed eight times, once for each significant bit of the multiplier. Initially, location X holds the multiplicand and location Y holds the multiplier. A counter CTR is set to −8 and location P is cleared to zero.

The multiplier bit can be checked if it is transferred to the *E* register. This is done by clearing *E*, loading the value of Y into the *AC*, circulating right *E* and *AC* and storing the shifted number back into location Y. This bit stored in *E* is the low-order bit of the multiplier. We now check the value of *E*. If it is 1, the multiplicand X is added to the partial product P. If it is 0, the partial product does not change. We then shift the value of X once to the left by loading it into the *AC* and circulating left *E* and *AC*. The loop is repeated eight times by incrementing location CTR and checking when it reaches zero. When the counter reaches zero, the program exits from the loop with the product stored in location P.

The program in Table 6-14 lists the instructions for multiplying two unsigned numbers. The initialization is not listed but should be included when the program is loaded into the computer. The initialization consists of bringing the multiplicand and multiplier into locations X and Y, respectively; initializing the counter to −8; and initializing location P to zero. If these locations are not

TABLE 6-14 Program to Multiply Two Positive Numbers

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

initialized, the program may run with incorrect data. The program itself is straightforward and follows the steps listed in the flowchart. The comments may help in following the step-by-step procedure.

This example has shown that if a computer does not have a machine instruction for a required operation, the operation can be programmed by a sequence of machine instructions. Thus we have demonstrated the software implementation of the multiplication operation. The corresponding hardware implementation is presented in Sec. 10-3.

Double-Precision Addition

When two 16-bit unsigned numbers are multiplied, the result is a 32-bit product that must be stored in two memory words. A number stored in two memory words is said to have double precision. When a partial product is computed, it is necessary that a double-precision number be added to the shifted multiplicand, which is also a double-precision number. For greater accuracy, the programmer may wish to employ double-precision numbers and perform arithmetic with operands that occupy two memory words. We now develop a program that adds two double-precision numbers.

One of the double-precision numbers is placed in two consecutive memory locations, AL and AH, with AL holding the 16 low-order bits. The other number is placed in BL and BH. The program is listed in Table 6-15. The two low-order portions are added and the carry transferred into E. The AC is cleared and the bit in E is circulated into the least significant position of the AC. The two high-order portions are then added to the carry and the double-precision sum is stored in CL and CH.

TABLE 6-15 Program to Add Two Double-Precision Numbers

	LDA AL	/Load *A* low
	ADD BL	/Add *B* low, carry in *E*
	STA CL	/Store in *C* low
	CLA	/Clear *AC*
	CIL	/Circulate to bring carry into *AC*(16)
	ADD AH	/Add *A* high and carry
	ADD BH	/Add *B* high
	STA CH	/Store in *C* high
	HLT	
AL,	—	/Location of operands
AH,	—	
BL,	—	
BH,	—	
CL,	—	
CH,	—	

Logic Operations

The basic computer has three machine instructions that perform logic operations: AND, CMA, and CLA. The LDA instruction may be considered as a logic operation that transfers a logic operand into the *AC*. In Sec. 4-5 we listed 16 different logic operations. All 16 logic operations can be implemented by software means because any logic function can be implemented using the AND and complement operations. For example, the OR operation is not available as a machine instruction in the basic computer. From DeMorgan's theorem we recognize the relation $x + y = (x'y')'$. The second expression contains only AND and complement operations. A program that forms the OR operation of two logic operands *A* and *B* is as follows:

```
LDA A    Load first operand A
CMA      Complement to get Ā
STA TMP  Store in a temporary location
LDA B    Load second operand B
CMA      Complement to get B̄
AND TMP  AND with Ā to get Ā ∧ B̄
CMA      Complement again to get A ∨ B
```

The other logic operations can be implemented by software in a similar fashion.

Shift Operations

The circular-shift operations are machine instructions in the basic computer. The other shifts of interest are the logical shifts and arithmetic shifts. These two shifts can be programmed with a small number of instructions.

The logical shift requires that zeros be added to the extreme positions. This is easily accomplished by clearing *E* and circulating the *AC* and *E*. Thus for a logical shift-right operation we need the two instructions

```
CLE
CIR
```

For a logical shift-left operation we need the two instructions

```
CLE
CIL
```

The arithmetic shifts depend on the type of representation of negative numbers. For the basic computer we have adopted the signed-2's complement representation. The rules for arithmetic shifts are listed in Sec. 4-6. For an arithmetic right-shift it is necessary that the sign bit in the leftmost position remain unchanged. But the sign bit itself is shifted into the high-order bit

position of the number. The program for the arithmetic right-shift requires that we set E to the same value as the sign bit and circulate right, thus:

```
CLE  /Clear E to 0
SPA  /Skip if AC is positive; E remains 0
CME  /AC is negative; set E to 1
CIR  /Circulate E and AC
```

For arithmetic shift-left it is necessary that the added bit in the least significant position be 0. This is easily done by clearing E prior to the circulate-left operation. The sign bit must not change during this shift. With a circulate instruction, the sign bit moves into E. It is then necessary to compare the sign bit with the value of E after the operation. If the two values are equal, the arithmetic shift has been correctly implemented. If they are not equal, an overflow occurs. An overflow indicates that the unshifted number was too large. When multiplied by 2 (by means of the shift), the number so obtained exceeds the capacity of the AC.

6-7 Subroutines

Frequently, the same piece of code must be written over again in many different parts of a program. Instead of repeating the code every time it is needed, there is an obvious advantage if the common instructions are written only once. A set of common instructions that can be used in a program many times is called a *subroutine*. Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine. After the subroutine has been executed, a branch is made back to the main program.

A subroutine consists of a self-contained sequence of instructions that carries out a given task. A branch can be made to the subroutine from any part of the main program. This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine. It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return. Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instructions to facilitate subroutine entry and return.

In the basic computer, the link between the main program and a subroutine is the BSA instruction (branch and save return address). To explain how this instruction is used, let us write a subroutine that shifts the content of the accumulator four times to the left. Shifting a word four times is a useful operation for processing binary-coded decimal numbers or alphanumeric characters. Such an operation could have been included as a machine instruction in the computer. Since it is not included, a subroutine is formed to accomplish this task. The program of Table 6-16 starts by loading the value of X into the

TABLE 6-16 Program to Demonstrate the Use of Subroutines

Location			
		ORG 100	/Main program
100		LDA X	/Load X
101		BSA SH4	/Branch to subroutine
102		STA X	/Store shifted number
103		LDA Y	/Load Y
104		BSA SH4	/Branch to subroutine again
105		STA Y	/Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/Subroutine to shift left 4 times
109	SH4,	HEX 0	/Store return address here
10A		CIL	/Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/Circulate left fourth time
10E		AND MSK	/Set *AC*(13–16) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

AC. The next instruction encountered is BSA SH4. The BSA instruction is in location 101. Subroutine SH4 must return to location 102 after it finishes its task. When the BSA instruction is executed, the control unit stores the return address 102 into the location defined by the symbolic address SH4 (which is 109). It also transfers the value of SH4 + 1 into the program counter. After this instruction is executed, memory location 109 contains the binary equivalent of hexadecimal 102 and the program counter contains the binary equivalent of hexadecimal 10A. This action has saved the return address and the subroutine is now executed starting from location 10A (since this is the content of *PC* in the next fetch cycle).

The computation in the subroutine circulates the content of *AC* four times to the left. In order to accomplish a logical shift operation, the four low-order bits must be set to zero. This is done by masking FFF0 with the content of *AC*. A mask operation is a logic AND operation that clears the bits of the *AC* where the mask operand is zero and leaves the bits of the *AC* unchanged where the mask operand bits are 1's.

The last instruction in the subroutine returns the computer to the main program. This is accomplished by the indirect branch instruction with an address symbol identical to the symbol used for the subroutine name. The address to which the computer branches is not SH4 but the value found in

location SH4 because this is an indirect address instruction. What is found in location SH4 is the return address 102 which was previously stored there by the BSA instruction. The computer returns to execute the instruction in location 102. The main program continues by storing the shifted number into location X. A new number is then loaded into the *AC* from location Y, and another branch is made to the subroutine. This time location SH4 will contain the return address 105 since this is now the location of the next instruction after BSA. The new operand is shifted and the subroutine returns to the main program at location 105.

From this example we see that the first memory location of each subroutine serves as a link between the main program and the subroutine. The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine *linkage*. The BSA instruction performs an operation commonly called subroutine *call*. The last instruction of the subroutine performs an operation commonly called subroutine *return*.

The procedure used in the basic computer for subroutine linkage is commonly found in computers with only one processor register. Many computers have multiple processor registers and some of them are assigned the name *index registers*. In such computers, an index register is usually employed to implement the subroutine linkage. A branch-to-subroutine instruction stores the return address in an index register. A return-from-subroutine instruction is effected by branching to the address presently stored in the index register.

Subroutine Parameters and Data Linkage

When a subroutine is called, the main program must transfer the data it wishes the subroutine to work with. In the previous example, the data were transferred through the accumulator. The operand was loaded into the *AC* prior to the branch. The subroutine shifted the number and left it there to be accepted by the main program. In general, it is necessary for the subroutine to have access to data from the calling program and to return results to that program. The accumulator can be used for a single input parameter and a single output parameter. In computers with multiple processor registers, more parameters can be transferred this way. Another way to transfer data to a subroutine is through the memory. Data are often placed in memory locations following the call. They can also be placed in a block of storage. The first address of the block is then placed in the memory location following the call. In any case, the return address always gives the link information for transferring data between the main program and the subroutine.

As an illustration, consider a subroutine that performs the logic OR operation. Two operands must be transferred to the subroutine and the subroutine must return the result of the operation. The accumulator can be used

to transfer one operand and to receive the result. The other operand is inserted in the location following the BSA instruction. This is demonstrated in the program of Table 6-17. The first operand in location X is loaded into the *AC*. The second operand is stored in location 202 following the BSA instruction. After the branch, the first location in the subroutine holds the number 202. Note that in this case, 202 is not the return address but the address of the second operand. The subroutine starts performing the OR operation by complementing the first operand in the *AC* and storing it in a temporary location TMP. The second operand is loaded into the *AC* by an indirect instruction at location OR. Remember that location OR contains the number 202. When the instruction refers to it indirectly, the operand at location 202 is loaded into the *AC*. This operand is complemented and then ANDed with the operand stored in TMP. Complementing the result forms the OR operation.

The return from the subroutine must be manipulated so that the main program continues from location 203 where the next instruction is located. This is accomplished by incrementing location OR with the ISZ instruction. Now location OR holds the number 203 and an indirect BUN instruction causes a return to the proper place.

It is possible to have more than one operand following the BSA instruc-

TABLE 6-17 Program to Demonstrate Parameter Linkage

Location			
		ORG 200	
200		LDA X	/Load first operand into *AC*
201		BSA OR	/Branch to subroutine OR
202		HEX 3AF6	/Second operand stored here
203		STA Y	/Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/First operand stored here
206	Y,	HEX 0	/Result stored here
207	OR,	HEX 0	/Subroutine OR
208		CMA	/Complement first operand
209		STA TMP	/Store in temporary location
20A		LDA OR I	/Load second operand
20B		CMA	/Complement second operand
20C		AND TMP	/AND complemented first operand
20D		CMA	/Complement again to get OR
20E		ISZ OR	/Increment return address
20F		BUN OR I	/Return to main program
210	TMP,	HEX 0	/Temporary storage
		END	

tion. The subroutine must increment the return address stored in its first location for each operand that it extracts from the calling program. Moreover, the calling program can reserve one or more locations for the subroutine to return results that are computed. The first location in the subroutine must be incremented for these locations as well, before the return. If there is a large amount of data to be transferred, the data can be placed in a block of storage and the address of the first item in the block is then used as the linking parameter.

A subroutine that moves a block of data starting at address 100 into a block starting with address 200 is listed in Table 6-18. The length of the block is 16 words. The first introduction is a branch to subroutine MVE. The first part of the subroutine transfers the three parameters 100, 200 and −16 from the main program and places them in its own storage location. The items are retrieved from their blocks by the use of two pointers. The counter ensures that only 16 items are moved. When the subroutine completes its operation, the data required is in the block starting from the location 200. The return to the main program is to the HLT instruction.

TABLE 6-18 Subroutine to Move a Block of Data

```
                        /Main program
        BSA MVE         /Branch to subroutine
        HEX 100         /First address of source data
        HEX 200         /First address of destination data
        DEC -16         /Number of items to move
        HLT
MVE,    HEX 0           /Subroutine MVE
        LDA MVE I       /Bring address of source
        STA PT1         /Store in first pointer
        ISZ MVE         /Increment return address
        LDA MVE I       /Bring address of destination
        STA PT2         /Store in second pointer
        ISZ MVE         /Increment return address
        LDA MVE I       /Bring number of items
        STA CTR         /Store in counter
        ISZ MVE         /Increment return address
LOP,    LDA PT1 I       /Load source item
        STA PT2 I       /Store in destination
        ISZ PT1         /Increment source pointer
        ISZ PT2         /Increment destination pointer
        ISZ CTR         /Increment counter
        BUN LOP         /Repeat 16 times
        BUN MVE I       /Return to main program
PT1,    —
PT2,    —
CTR,    —
```

6-8 Input–Output Programming

Users of the computer write programs with symbols that are defined by the programming language employed. The symbols are strings of characters and each character is assigned an 8-bit code so that it can be stored in computer memory. A binary-coded character enters the computer when an INP (input) instruction is executed. A binary-coded character is transferred to the output device when an OUT (output) instruction is executed. The output device detects the binary code and types the corresponding character.

Table 6-19(a) lists the instructions needed to input a character and store it in memory. The SKI instruction checks the input flag to see if a character is available for transfer. The next instruction is skipped if the input flag bit is 1. The INP instruction transfers the binary-coded character into *AC*(0–7). The character is then printed by means of the OUT instruction. A terminal unit that communicates directly with a computer does not print the character when a key is depressed. To type it, it is necessary to send an OUT instruction for the printer. In this way, the user is ensured that the correct transfer has occurred. If the SKI instruction finds the flag bit at 0, the next instruction in sequence is executed. This instruction is a branch to return and check the flag bit again. Because the input device is much slower than the computer, the two instructions in the loop will be executed many times before a character is transferred into the accumulator.

Table 6-19(b) lists the instructions needed to print a character initially stored in memory. The character is first loaded into the *AC*. The output flag is then checked. If it is 0, the computer remains in a two-instruction loop checking the flag bit. When the flag changes to 1, the character is transferred from the accumulator to the printer.

TABLE 6-19 Programs to Input and Output One Character

```
(a) Input a character:
CIF,    SKI         /Check input flag
        BUN CIF     /Flag=0, branch to check again
        INP         /Flag=1, input character
        OUT         /Print character
        STA CHR     /Store character
        HLT
CHR,    —           /Store character here
(b) Output one character:
        LDA CHR     /Load character into AC
COF,    SKO         /Check output flag
        BUN COF     /Flag=0, branch to check again
        OUT         /Flag=1, output character
        HLT
CHR,    HEX 0057    /Character is "W"
```

Character Manipulation

A computer is not just a calculator but also a symbol manipulator. The binary-coded characters that represent symbols can be manipulated by computer instructions to achieve various data-processing tasks. One such task may be to pack two characters in one word. This is convenient because each character occupies 8 bits and a memory word contains 16 bits. The program in Table 6-20 lists a subroutine named IN2 that inputs two characters and packs them into one 16-bit word. The packed word remains in the accumulator. Note that subroutine SH4 (Table 6-16) is called twice to shift the accumulator left eight times.

In the discussion of the assembler it was assumed that the symbolic program is stored in a section of memory which is sometimes called a *buffer*. The symbolic program being typed enters through the input device and is stored in consecutive memory locations in the buffer. The program listed in Table 6-21 can be used to input a symbolic program from the keyboard, pack two characters in one word, and store them in the buffer. The first address of the buffer is 500. The first double character is stored in location 500 and all characters are stored in sequential locations. The program uses a pointer for keeping track of the current empty location in the buffer. No counter is used in the program, so characters will be read as long as they are available or until the buffer reaches location 0 (after location FFFF). In a practical situation it may be necessary to limit the size of the buffer and a counter may be used for this purpose. Note that subroutine IN2 of Table 6-20 is called to input and pack the two characters.

In discussing the second pass of the assembler in Sec. 6-4 it was mentioned that one of the most common operations of an assembler is table lookup. This is an operation that searches a table to find out if it contains a given symbol. The search may be done by comparing the given symbol with each of the symbols stored in the table. The search terminates when a match occurs

TABLE 6-20 Subroutine to Input and Pack Two Characters

```
IN2,    —          /Subroutine entry
FST,    SKI
        BUN FST
        INP        /Input first character
        OUT
        BSA SH4    /Shift left four times
        BSA SH4    /Shift left four more times
SCD,    SKI
        BUN SCD
        INP        /Input second character
        OUT
        BUN IN2 I  /Return
```

TABLE 6-21 Program to Store Input Characters in a Buffer

```
        LDA ADS     /Load first address of buffer
        STA PTR     /Initialize pointer
LOP,    BSA IN2     /Go to subroutine IN2 (Table 6-20)
        STA PTR I   /Store double character word in buffer
        ISZ PTR     /Increment pointer
        BUN LOP     /Branch to input more characters
        HLT
ADS,    HEX 500     /First address of buffer
PTR,    HEX 0       /Location for pointer
```

or if none of the symbols match. When a match occurs, the assembler retrieves the equivalent binary value. A program for comparing two words is listed in Table 6-22. The comparison is accomplished by forming the 2's complement of a word (as if it were a number) and arithmetically adding it to the second word. If the result is zero, the two words are equal and a match occurs. If the result is not zero, the words are not the same. This program can serve as a subroutine in a table-lookup program.

Program Interrupt

The running time of input and output programs is made up primarily of the time spent by the computer in waiting for the external device to set its flag. The waiting loop that checks the flag keeps the computer occupied with a task that wastes a large amount of time. This waiting time can be eliminated if the interrupt facility is used to notify the computer when a flag is set. The advantage of using the interrupt is that the information transfer is initiated upon request from the external device. In the meantime, the computer can be busy performing other useful tasks. Obviously, if no other program resides in memory, there is nothing for the computer to do, so it might as well check for

TABLE 6-22 Program to Compare Two Words

```
        LDA WD1     /Load first word
        CMA
        INC         /Form 2's complement
        ADD WD2     /Add second word
        SZA         /Skip if AC is zero
        BUN UEQ     /Branch to "unequal" routine
        BUN EQL     /Branch to "equal" routine
WD1,    —
WD2,    —
```

the flags. The interrupt facility is useful in a multiprogram environment when two or more programs reside in memory at the same time.

Only one program can be executed at any given time even though two or more programs may reside in memory. The program currently being executed is referred to as the running program. The other programs are usually waiting for input or output data. The function of the interrupt facility is to take care of the data transfer of one (or more) program while another program is currently being executed. The running program must include an ION instruction to turn the interrupt on. If the interrupt facility is not used, the program must include an IOF instruction to turn it off. (The *start* switch of the computer should also turn the interrupt off.)

The interrupt facility allows the running program to proceed until the input or output device sets its ready flag. Whenever a flag is set to 1, the computer completes the execution of the instruction in progress and then acknowledges the interrupt. The result of this action is that the return address is stored in location 0. The instruction in location 1 is then performed; this initiates a service routine for the input or output transfer. The service routine can be stored anywhere in memory provided a branch to the start of the routine is stored in location 1. The service routine must have instructions to perform the following tasks:

1. Save contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

The contents of processor registers before the interrupt and after the return to the running program must be the same; otherwise, the running program may be in error. Since the service routine may use these registers, it is necessary to save their contents at the beginning of the routine and restore them at the end. The sequence by which the flags are checked dictates the priority assigned to each device. Even though two or more flags may be set at the same time, the devices nevertheless are serviced one at a time. The device with higher priority is serviced first followed by the one with lower priority.

The occurrence of an interrupt disables the facility from further interrupts. The service routine must turn the interrupt on before the return to the running program. This will enable further interrupts while the computer is executing the running program. The interrupt facility should not be turned on until after the return address is inserted into the program counter.

An example of a program that services an interrupt is listed in Table 6-23.

TABLE 6-23 Program to Service an Interrupt

Location			
0	ZRO,	—	/Return address stored here
1		BUN SRV	/Branch to service routine
100		CLA	/Portion of running program
101		ION	/Turn on interrupt facility
102		LDA X	
103		ADD Y	/Interrupt occurs here
104		STA Z	/Program returns here after interrupt
•		•	
•		•	
•			/Interrupt service routine
200	SRV,	STA SAC	/Store content of *AC*
		CIR	/Move *E* into *AC*(1)
		STA SE	/Store content of *E*
		SKI	/Check input flag
		BUN NXT	/Flag is off, check next flag
		INP	/Flag is on, input character
		OUT	/Print character
		STA PT1 I	/Store it in input buffer
		ISZ PT1	/Increment input pointer
	NXT,	SKO	/Check output flag
		BUN EXT	/Flag is off, exit
		LDA PT2 I	/Load character from output buffer
		OUT	/Output character
		ISZ PT2	/Increment output pointer
	EXT,	LDA SE	/Restore value of *AC*(1)
		CIL	/Shift it to *E*
		LDA SAC	/Restore content of *AC*
		ION	/Turn interrupt on
		BUN ZRO I	/Return to running program
	SAC,	—	/*AC* is stored here
	SE,	—	/*E* is stored here
	PT1,	—	/Pointer of input buffer
	PT2,	—	/Pointer of output buffer

Location 0 is reserved for the return address. Location 1 has a branch instruction to the beginning of the service routine SRV. The portion of the running program listed has an ION instruction that turns the interrupt on. Suppose that an interrupt occurs while the computer is executing the instruction in location 103. The interrupt cycle stores the binary equivalent of hexadecimal 104 in location 0 and branches to location 1. The branch instruction in location 1 sends the computer to the service routine SRV.

The service routine performs the six tasks mentioned above. The contents of *AC* and *E* are stored in special locations. (These are the only processor registers in the basic computer.) The flags are checked sequentially, the input flag first and the output flag second. If any or both flags are set, an item of data is transferred to or from the corresponding memory buffer. Before returning to the running program the previous contents of *E* and *AC* are restored and the interrupt facility is turned on. The last instruction causes a branch to the address stored in location 0. This is the return address stored there previously during the interrupt cycle. Hence the running program will continue from location 104, where it was interrupted.

A typical computer may have many more input and output devices connected to the interrupt facility. Furthermore, interrupt sources are not limited to input and output transfers. Interrupts can be used for other purposes, such as internal processing errors or special alarm conditions. Further discussion of interrupts and some advanced concepts concerning this important subject can be found in Sec. 11-5.

PROBLEMS

6-1. The following program is stored in the memory unit of the basic computer. Show the contents of the *AC*, *PC*, and *IR* (in hexadecimal), at the end, after each instruction is executed. All numbers listed below are in hexadecimal.

Location	Instruction
010	CLA
011	ADD 016
012	BUN 014
013	HLT
014	AND 017
015	BUN 013
016	C1A5
017	93C6

6-2. The following program is a list of instructions in hexadecimal code. The computer executes the instructions starting from address 100. What are the content of *AC* and the memory word at address 103 when the computer halts?

Location	Instruction
100	5103
101	7200
102	7001
103	0000
104	7800
105	7020
106	C103

6-3. List the assembly language program (of the equivalent binary instructions) generated by a compiler from the following Fortran program. Assume integer variables.

```
SUM = 0
SUM = SUM + A + B
DIF = DIF - C
SUM = SUM + DIF
```

6-4. Can the letter I be used as a symbolic address in the assembly language program defined for the basic computer? Justify the answer.

6-5. What happens during the first pass of the assembler (Fig. 6-1) if the line of code that has a pseudoinstruction ORG or END also has a label? Modify the flowchart to include an error message if this occurs.

6-6. A line of code in an assembly language program is as follows:

```
DEC -35
```

a. Show that four memory words are required to store the line of code and give their binary content.

b. Show that one memory word stores the binary translated code and give its binary content.

6-7. **a.** Obtain the address symbol table generated for the program of Table 6-13 during the first pass of the assembler.

b. List the translated program in hexadecimal.

6-8. The pseudoinstruction BSS N (block started by symbol) is sometimes employed to reserve *N* memory words for a group of operands. For example, the line of code

```
A,  BSS 10
```

informs the assembler that a block of 10 (decimal) locations is to be left free, starting from location A. This is similar to the Fortran statement DIMENSION A(10). Modify the flowchart of Fig. 6-1 to process this pseudoinstruction.

6-9. Modify the flowchart of Fig. 6-2 to include an error message when a symbolic address is not defined by a label.

6-10. Show how the MRI and non-MRI tables can be stored in memory.

6-11. List the assembly language program (of the equivalent binary instructions) generated by a compiler for the following IF statement:

```
IF(A - B) 10, 20, 30
```

The program branches to statement 10 if $A - B < 0$; to statement 20 if $A - B = 0$; and to statement 30 if $A - B > 0$.

6-12. **a.** Explain in words what the following program accomplishes when it is executed. What is the value of location CTR when the computer halts?
b. List the address symbol table obtained during the first pass of the assembler.
c. List the hexadecimal code of the translated program.

```
        ORG 100
        CLE
        CLA
        STA CTR
        LDA WRD
        SZA
        BUN ROT
        BUN STP
ROT,    CIL
        SZE
        BUN AGN
        BUN ROT
AGN,    CLE
        ISZ CTR
        SZA
        BUN ROT
STP,    HLT
CTR,    HEX 0
WRD,    HEX 62C1
        END
```

6-13. Write a program loop, using a pointer and a counter, that clears to 0 the contents of hexadecimal locations 500 through 5FF.

6-14. Write a program to multiply two positive numbers by a repeated addition method. For example, to multiply 5×4, the program evaluates the product by adding 5 four times, or $5 + 5 + 5 + 5$.

6-15. The multiplication program of Table 6-14 is not initialized. After the program is executed once, location CTR will be left with zero. Show that if the program is executed again starting from location 100, the loop will be traversed 65536 times. Add the needed instructions to initialize the program.

6-16. Write a program to multiply two unsigned positive numbers, each with 16 significant bits, to produce an unsigned double-precision product.

6-17. Write a program to multiply two signed numbers with negative numbers being initially in signed-2's complement representation. The product should be single-precision and signed-2's complement representation if negative.

6-18. Write a program to subtract two double-precision numbers.

6-19. Write a program that evaluates the logic exclusive-OR of two logic operands.

6-20. Write a program for the arithmetic shift-left operation. Branch to OVF if an overflow occurs.

6-21. Write a subroutine to subtract two numbers. In the calling program, the BSA instruction is followed by the subtrahend and minuend. The difference is returned to the main program in the third location following the BSA instruction.

6-22. Write a subroutine to complement each word in a block of data. In the calling program, the BSA instruction is followed by two parameters: the starting address of the block and the number of words in the block.

6-23. Write a subroutine to circulate E and AC four times to the right. If AC contains hexadecimal 079C and $E = 1$, what are the contents of AC and E after the subroutine is executed?

6-24. Write a program to accept input characters, pack two characters in one word and store them in consecutive locations in a memory buffer. The first address of the buffer is $(400)_{16}$. The size of the buffer is $(512)_{10}$ words. If the buffer overflows, the computer should halt.

6-25. Write a program to unpack two characters from location WRD and store them in bits 0 through 7 of locations CH1 and CH2. Bits 9 through 15 should contain zeros.

6-26. Obtain a flowchart for a program to check for a CR code (hexadecimal 0D) in a memory buffer. The buffer contains two characters per word. When the code for CR is encountered, the program transfers it to bits 0 through 7 of location LNE without disturbing bits 8 through 15.

6-27. Translate the service routine SRV from Table 6-23 to its equivalent hexadecimal code. Assume that the routine is stored starting from location 200.

6-28. Write an interrupt service routine that performs all the required functions but the input device is serviced only if a special location, MOD, contains all 1's. The output device is serviced only if location MOD contains all 0's.

REFERENCES

1. Booth, T. L., *Introduction to Computer Engineering*, 3rd ed. New York: John Wiley, 1984.

2. Gear, C. W., *Computer Organization and Programming*, 3rd ed. New York: McGraw-Hill, 1980.

3. Gibson, G. A., *Computer Systems Concepts and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
4. Gray, N. A. B., *Introduction to Computer Systems*. Englewood Cliffs, NJ: Prentice Hall, 1987.
5. Levy, H. M., and R. H. Eckhouse, Jr., *Computer Programming and Architecture: The VAX-11*. Bedford, MA: Digital Press, 1980.
6. Lewin, M. H., *Logic Design and Computer Organization*. Reading, MA: Addison-Wesley, 1983.
7. Prosser, F. P., and D. E. Winkel, *The Art of Digital Design*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1987.
8. Shiva, S. G., *Computer Design and Architecture*, 2nd ed. New York: HarperCollins Publishers, 1991.
9. Tanenbaum, A. S., *Structured Computer Organization*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1990.
10. Wakerly, J. F., *Microcomputer Architecture and Programming*. New York: John Wiley, 1981.

CHAPTER SEVEN

Microprogrammed Control

IN THIS CHAPTER

7-1 Control Memory

The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be *hardwired*. Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the microoperation sequences in a digital computer.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. A control variable in the opposite binary state does not change the state of the registers in the system. The active state of a control variable may be either the 1 state or the 0 state, depending on the application. In a bus-organized system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.

The control unit initiates a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. 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 perform various operations on the components of the system. A control unit whose binary control variables are stored in memory is called

control word

microinstruction

microprogram

a *microprogrammed control unit*. Each word in control memory contains within it a *microinstruction*. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a *microprogram*. Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM). The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM. ROM words are made permanent during the hardware production of the unit. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies a microinstruction.

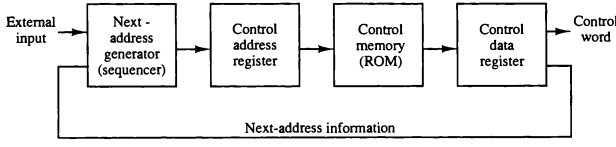
A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the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

control memory

A computer that employs a microprogrammed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations. Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Fig. 7-1. The control memory is assumed to be a ROM, within which all control information is permanently stored. The

Figure 7-1 Microprogrammed control organization.



control address register

control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

sequencer

The next address generator is sometimes called a microprogram *sequencer*, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

pipeline register

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is sometimes called a *pipeline register*. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single-phase clock to the address register. The control word and next-address information are taken directly from the 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 in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed as in a random-access memory. Each clock pulse will execute the microoperations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single-phase clock and therefore we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components: the sequencer and the control memory are combinational circuits and do not need a clock.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control se-

quence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

It should be mentioned that most computers based on the reduced instruction set computer (RISC) architecture concept (see Sec. 8-8) use hardwired control rather than a control memory with a microprogram. An example of a hardwired control for a simple computer is presented in Sec. 5-4.

hardwired control

7-2 Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a *routine*. Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another. To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.

routine

An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions. At the end of the fetch routine, the instruction is in the instruction register of the computer.

The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. When the effective address computation routine is completed, the address of the operand is available in the memory address register.

The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a *mapping* process. A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register, but sometimes the sequence of microopera-

mapping

tions will depend on values of certain status bits in processor registers. Microprograms that employ subroutines will require an external register for storing the return address. Return addresses cannot be stored in ROM because the unit has no writing capability.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine. In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Figure 7-2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address. The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

Conditional Branching

special bits

The branch logic of Fig. 7-2 provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

branch logic

The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented.

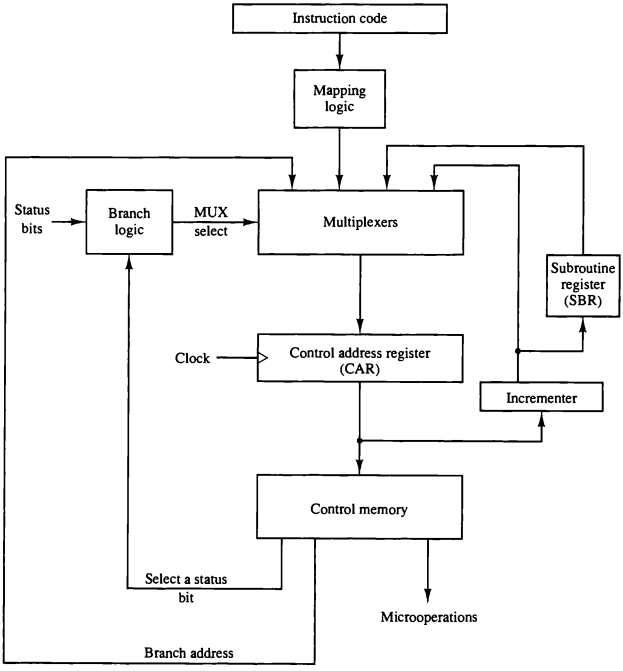


Figure 7-2 Selection of address for control memory.

This can be implemented with a multiplexer. Suppose that there are eight status bit conditions in the system. Three bits in the microinstruction are used to specify any one of eight status bit conditions. These three bits provide the selection variables for the multiplexer. If the selected status bit is in the 1 state, the output of the multiplexer is 1; otherwise, it is 0. A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register. A 0 output in the multiplexer causes the address register to be incremented. In this configuration, the microprogram follows one of two possible paths, depending on the value of the selected status bit.

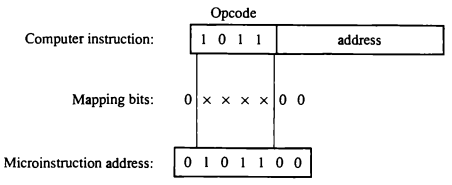
An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status bit select lines from control memory causes the branch address to be loaded into the control address register unconditionally.

Mapping of Instruction

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Fig. 7-3 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Fig. 7-3. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. In this configuration, the bits of the instruction specify the address of a mapping ROM. The contents of the mapping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory. The mapping concept provides flexibility for adding instructions for control memory as the need arises.

Figure 7-3 Mapping from instruction code to microinstruction address.



The mapping function is sometimes implemented by means of an integrated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses. The interconnection between inputs, AND gates, OR gates, and outputs can be programmed as in ROM. A mapping function that can be expressed in terms of Boolean expressions can be implemented conveniently with a PLD.

Subroutines

Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many microprograms contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented output from the control address register into a subroutine register and branching to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine. The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack. The use of a stack in subroutine calls and returns is explained in more detail in Sec. 8-7.

subroutine register

7-3 Microprogram Example

Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming. To appreciate this process, we present here a simple digital computer and show how it is microprogrammed. The computer used here is similar but not identical to the basic computer introduced in Chap. 5.

Computer Configuration

The block diagram of the computer is shown in Fig. 7-4. It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are

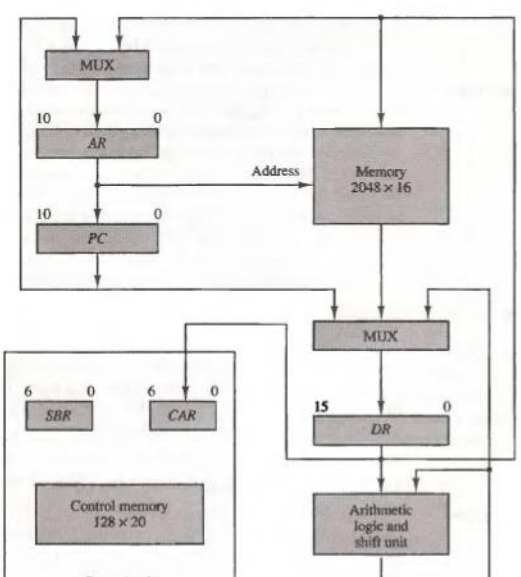


Figure 7-4 Computer hardware configuration.

program counter *PC*, address register *AR*, data register *DR*, and accumulator register *AC*. The function of these registers is similar to the basic computer introduced in Chap. 5 (see Fig. 5-3). The control unit has a control address register *CAR* and a subroutine register *SBR*. The control memory and its registers are organized as a microprogrammed control unit, as shown in Fig. 7-2.

The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. *DR* can receive information from *AC*, *PC*, or memory. *AR* can receive information from *PC* or *DR*. *PC* can receive information only from *AR*. The arithmetic, logic, and shift unit per-

forms microoperations with data from *AC* and *DR* and places the result in *AC*. Note that memory receives its address from *AR*. Input data written to memory come from *DR*, and data read from memory can go only to *DR*.

instruction format

The computer instruction format is depicted in Fig. 7-5(a). It consists of three fields: a 1-bit field for indirect addressing symbolized by *I*, a 4-bit operation code (opcode), and an 11-bit address field. Figure 7-5(b) lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of *AC*. The BRANCH instruction causes a branch to the effective address if the operand in *AC* is negative. The program proceeds with the next consecutive instruction if *AC* is not negative. The *AC* is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of *AC* into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between *AC* and the memory word specified by the effective address.

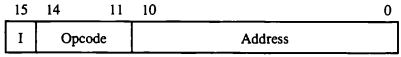
It will be shown subsequently that each computer instruction must be microprogrammed. In order not to complicate the microprogramming example, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be microprogrammed by the procedure outlined below.

Microinstruction Format

microinstruction format

The microinstruction format for the control memory is shown in Fig. 7-6. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field

Figure 7-5 Computer instructions.

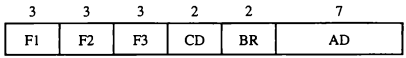


(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 7-6 Microinstruction code format (20 bits).

selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.

microoperations

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in Table 7-1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. As an illustration, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

$$DR \leftarrow M[AR] \qquad \text{with F2} = 100$$

$$\text{and} \quad PC \leftarrow PC + 1 \qquad \text{with F3} = 101$$

The nine bits of the microoperation fields will then be 000 100 101. It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear *AC* to 0 and subtract *DR* from *AC* at the same time.

Each microoperation in Table 7-1 is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram. All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letter is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer $AC \leftarrow DR$ (F1 = 100) has the symbol DRTAC, which stands for a transfer from *DR* to *AC*.

condition field

The *CD* (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 7-1. The first condition is always a 1, so that a reference to *CD* = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit

TABLE 7-1 Symbols and Binary Code for Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

I is available from bit 15 of *DR* after an instruction is read from memory. The sign bit of *AC* provides the next status bit. The zero value, symbolized by *Z*, is a binary variable whose value is equal to 1 if all the bits in *AC* are equal to zero. We will use the symbols U, I, S, and Z for the four status bits when we write microprograms in symbolic form.

branch field

The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction. As shown in Table 7-1, when BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine register *SBR*. The jump and call operations depend on the value of the CD field. If the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register *CAR*. Otherwise, *CAR* is incremented by 1.

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from *SBR* to *CAR*. The mapping from the operation code bits of the instruction to an address for *CAR* is accomplished when the BR field is equal to 11. This mapping is as depicted in Fig. 7-3. The bits of the operation code are in *DR*(11–14) after an instruction is read from memory. Note that the last two conditions in the BR field are independent of the values in the CD and AD fields.

Symbolic Microinstructions

The symbols defined in Table 7-1 can be used to specify microinstructions in symbolic form. A symbolic microprogram can be translated into its binary equivalent by means of an assembler. A microprogram assembler is similar in concept to a conventional computer assembler as defined in Sec. 6-3. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.

Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following information.

1. The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2. The microoperations field consists of one, two, or three symbols, separated by commas, from those defined in Table 7-1. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.

3. The CD field has one of the letters U, I, S, or Z.
4. The BR field contains one of the four symbols defined in Table 7-1.

address field

5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:
   a. With a symbolic address, which must also appear as a label.
   b. With the symbol NEXT to designate the next address in sequence.
   c. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

ORG

We will use also the pseudoinstruction ORG to define the origin, or first address, of a microprogram routine. Thus the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000.

The Fetch Routine

The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose. A convenient starting location for the fetch routine is address 64. The microinstructions needed for the fetch routine are

$$AR \leftarrow PC$$

$$DR \leftarrow \text{M}[AR], \quad PC \leftarrow PC + 1$$

$$AR \leftarrow DR(0\text{–}10), \quad CAR(2\text{–}5) \leftarrow DR(11\text{–}14), \quad CAR(0,1,6) \leftarrow 0$$

The address of the instruction is transferred from *PC* to *AR* and the instruction is then read from memory into *DR*. Since no instruction register is available, the instruction code remains in *DR*. The address part is transferred to *AR* and then control is transferred to one of 16 routines by mapping the operation code part of the instruction from *DR* into *CAR*.

fetch and decode

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. Using the assembly language conventions defined previously, we can write the symbolic microprogram for the fetch routine as follows:

```
        ORG 64
FETCH:  PCTAR          U    JMP    NEXT
        READ, INCPC    U    JMP    NEXT
        DRTAR          U    MAP
```

The translation of the symbolic microprogram to binary produces the following binary microprogram. The bit values are obtained from Table 7-1.

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

The three microinstructions that constitute the fetch routine have been listed in three different representations. The register transfer representation shows the internal register transfer operations that each microinstruction implements. The symbolic representation is useful for writing microprograms in an assembly language format. The binary representation is the actual internal content that must be stored in control memory. It is customary to write microprograms in symbolic form and then use an assembler program to obtain a translation to binary.

Symbolic Microprogram

The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address 0xxxx00, where xxxx are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will transfer to *CAR* the address 0000000, which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20, . . . , 60. This gives four words in control memory for each routine.

In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions. A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown in Table 7-2. The table also shows the symbolic microprogram for the fetch routine and the microinstruction routines that execute four computer instructions.

To see how the transfer and return from the indirect subroutine occurs, assume that the MAP microinstruction at the end of the fetch routine caused a branch to address 0, where the ADD routine is stored. The first microinstruction in the ADD routine calls subroutine INDRCT, conditioned on status bit *I*. If $I = 1$, a branch to INDRCT occurs and the return address (address 1 in this case) is stored in the subroutine register *SBR*. The INDRCT subroutine has two microinstructions:

```
INDRCT:   READ     U    JMP    NEXT
          DRTAR    U    RET
```

TABLE 7-2 Symbolic Microprogram (Partial)

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

Remember that an indirect address considers the address part of the instruction as the address where the effective address is stored rather than the address of the operand. Therefore, the memory has to be accessed to get the effective address, which is then transferred to *AR*. The return from subroutine (RET) transfers the address from *SBR* to *CAR*, thus returning to the second microinstruction of the ADD routine.

execution of instructions

The execution of the ADD instruction is carried out by the microinstructions at addresses 1 and 2. The first microinstruction reads the operand from memory into *DR*. The second microinstruction performs an add microoperation with the content of *DR* and *AC* and then jumps back to the beginning of the fetch routine.

The BRANCH instruction should cause a branch to the effective address

if $AC < 0$. The AC will be less than zero if its sign is negative, which is detected from status bit S being a 1. The BRANCH routine in Table 7-2 starts by checking the value of S. If S is equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S is equal to 1, the first JMP microinstruction transfers control to location OVER. The microinstruction at this location calls the INDRCT subroutine if $I = 1$. The effective address is then transferred from AR to PC and the microprogram jumps back to the fetch routine.

The STORE routine again uses the INDRCT subroutine if $I = 1$. The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address in AR.

The EXCHANGE routine reads the operand from the effective address and places it in DR. The contents of DR and AC are interchanged in the third microinstruction. This interchange is possible when the registers are of the edge-triggered type (see Fig. 1-23). The original content of AC that is now in DR is stored back in memory.

Note that Table 7-2 contains a partial list of the microprogram. Only four out of 16 possible computer instructions have been microprogrammed. Also, control memory words at locations 69 to 127 have not been used. Instructions such as multiply, divide, and others that require a long sequence of microoperations will need more than four microinstructions for their execution. Control memory words 69 to 127 can be used for this purpose.

Binary Microprogram

The symbolic microprogram is a convenient form for writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in memory. The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough as in this example.

The equivalent binary form of the microprogram is listed in Table 7-3. The addresses for control memory are given in both decimal and binary. The binary content of each microinstruction is derived from the symbols and their equivalent binary values as defined in Table 7-1.

Note that address 3 has no equivalent in the symbolic microprogram since the ADD routine has only three microinstructions at addresses 0, 1, and 2. The next routine starts at address 4. Even though address 3 is not used, some binary value must be specified for each word in control memory. We could have specified all 0's in the word since this location will never be used. However, if some unforeseen error occurs, or if a noise signal sets CAR to the value of 3, it will be wise to jump to address 64, which is the beginning of the fetch routine.

control memory

The binary microprogram listed in Table 7-3 specifies the word content of the control memory. When a ROM is used for the control memory, the

TABLE 7-3 Binary Microprogram for Control Memory (Partial)

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

microprogram binary list provides the truth table for fabricating the unit. This fabrication is a hardware process and consists of creating a mask for the ROM so as to produce the 1's and 0's for each word. The bits of ROM are fixed once the internal links are fused during the hardware production. The ROM is made of IC packages that can be removed if necessary and replaced by other packages. To modify the instruction set of the computer, it is necessary to generate a new microprogram and mask a new ROM. The old one can be removed and the new one inserted in its place.

If a writable control memory is employed, the ROM is replaced by a RAM. The advantage of employing a RAM for the control memory is that the microprogram can be altered simply by writing a new pattern of 1's and 0's without resorting to hardware procedures. A writable control memory possesses the flexibility of choosing the instruction set of a computer dynamically by changing the microprogram under processor control. However, most microprogrammed systems use a ROM for the control memory because it is

cheaper and faster than a RAM and also to prevent the occasional user from changing the architecture of the system.

7-4 Design of Control Unit

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2^k microoperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

decoding of F fields

Figure 7-7 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3×8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 7-1. For example, when $F1 = 101$ (binary 5), the next clock pulse transition transfers the content of DR(0-10) to AR (symbolized by DRTAR in Table 7-1). Similarly, when $F1 = 110$ (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 7-7, outputs 5 and 6 of decoder $F1$ are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active. The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.

arithmetic logic shift unit

The arithmetic logic shift unit can be designed as in Figs. 5-19 and 5-20. Instead of using gates to generate the control signals marked by the symbols AND, ADD, and DR in Fig. 5-19, these inputs will now come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC, respec-

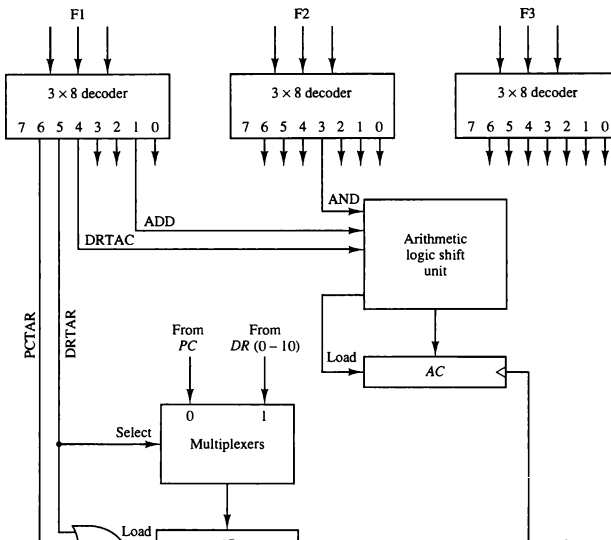


Figure 7-7 Decoding of microoperation fields.

tively, as shown in Fig. 7-7. The other outputs of the decoders that are associated with an *AC* operation must also be connected to the arithmetic logic shift unit in a similar fashion.

Microprogram Sequencer

The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.

The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction. Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular unit that is suitable for use in the microprogram computer example developed in the preceding section. The block diagram of the microprogram sequencer is shown in Fig. 7-8. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register *CAR*. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from *CAR* provides the address for the control memory. The content of *CAR* is incremented and applied to one of the multiplexer inputs and to the subroutine register *SBR*. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of *SBR*, and from an external source that maps the instruction. Although the diagram shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

The CD (condition) field of the microinstruction selectgs one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

design of input logic

The input logic circuit in Fig. 7-8 has three inputs, I_0, I_1, and T, and three outputs, S_0, S_1, and L. Variables S_0 and S_1 select one of the source addresses for *CAR*. Variable L enables the load input in *SBR*. The binary values of the two selection variables determine the path in the multiplexer. For example, with $S_1S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer

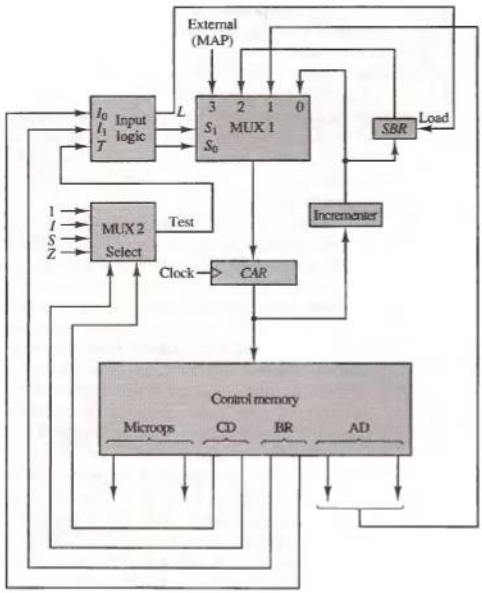


Figure 7-8 Microprogram sequencer for a control memory.

path from *SBR* to *CAR*. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

The truth table for the input logic circuit is shown in Table 7-4. Inputs I_1 and I_0 are identical to the bit values in the BR field. The function listed in each entry was defined in Table 7-1. The bit values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of *CAR* during a call microinstruction (BR = 01) provided that the status bit condition is satisfied ($T = 1$). The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

$$S_1 = I_1$$
$$S_0 = I_1 I_0 + I_1' T$$
$$L = I_1' I_0 T$$

TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer

BR Field		Input			MUX 1		Load *SBR*
		I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

The circuit can be constructed with three AND gates, an OR gate, and an inverter.

Note that the incrementer circuit in the sequencer of Fig. 7-8 is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits (see Fig. 4-8). The output carry from one stage must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation.

PROBLEMS

7-1. What is the difference between a microprocessor and a microprogram? Is it possible to design a microprocessor without a microprogram? Are all microprogrammed computers also microprocessors?

7-2. Explain the difference between hardwired control and microprogrammed control. Is it possible to have a hardwired control associated with a control memory?

7-3. Define the following: (a) microoperation; (b) microinstruction; (c) microprogram; (d) microcode.

7-4. The microprogrammed control organization shown in Fig. 7-1 has the following propagation delay times. 40 ns to generate the next address, 10 ns to transfer the address into the control address register, 40 ns to access the control memory ROM, 10 ns to transfer the microinstruction into the control data register, and 40 ns to perform the required microoperations specified by the control word. What is the maximum clock frequency that the control can use? What would the clock frequency be if the control data register is not used?

7-5. The system shown in Fig. 7-2 uses a control memory of 1024 words of 32 bits each. The microinstruction has three fields as shown in the diagram. The microoperations field has 16 bits.

a. How many bits are there in the branch address field and the select field?

b. If there are 16 status bits in the system, how many bits of the branch logic are used to select a status bit?
c. How many bits are left to select an input for the multiplexers?

7-6. The control memory in Fig. 7-2 has 4096 words of 24 bits each.
a. How many bits are there in the control address register?
b. How many bits are there in each of the four inputs shown going into the multiplexers?
c. What are the number of inputs in each multiplexer and how many multiplexers are needed?

7-7. Using the mapping procedure described in Fig. 7-3, give the first microinstruction address for the following operation code: (a) 0010; (b) 1011; (c) 1111.

7-8. Formulate a mapping procedure that provides eight consecutive microinstructions for each routine. The operation code has six bits and the control memory has 2048 words.

7-9. Explain how the mapping from an instruction code to a microinstruction address can be done by means of a read-only memory. What is the advantage of this method compared to the one in Fig. 7-3?

7-10. Why do we need the two multiplexers in the computer hardware configuration shown in Fig. 7-4? Is there another way that information from multiple sources can be transferred to a common destination?

7-11. Using Table 7-1, give the 9-bit microoperation field for the following microoperations:
a. $AC \leftarrow AC + 1, \quad DR \leftarrow DR + 1$
b. $PC \leftarrow PC + 1, \quad DR \leftarrow M[AR]$
c. $DR \leftarrow AC, \quad AC \leftarrow DR$

7-12. Using Table 7-1, convert the following symbolic microoperations to register transfer statements and to binary.
a. READ, INCPC
b. ACTDR, DRTAC
c. ARTPC, DRTAC, WRITE

7-13. Suppose that we change the ADD routine listed in Table 7-2 to the following two microinstructions.

```
ADD:    READ    I    CALL    INDR2
        ADD     U    JMP     FETCH
```

What should be subroutine INDR2?

7-14. The following is a symbolic microprogram for an instruction in the computer defined in Sec. 7-3.

```
        ORG 40
NOP             S    JMP     FETCH
NOP             Z    JMP     FETCH
NOP             I    CALL    INDRCT
ARTPC           U    JMP     FETCH
```

a. Specify the operation performed when the instruction is executed.
b. Convert the four microinstructions into their equivalent binary form.

7-15. The computer of Sec. 7-3 has the following binary microprogram:

Address	Binary Microprogram
60	0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1
61	1 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0
62	0 0 1 0 0 1 0 0 0 0 1 0 1 0 0 1 1 1 1 1 1
63	1 0 1 1 1 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0

a. Translate it to a symbolic microprogram as in Table 7-2. (FETCH is in address 64 and INDRCT in address 67.)

b. List all the things that will be wrong when this microprogram is executed in the computer.

7-16. Add the following instructions to the computer of Sec 7-3 (*EA* is the effective address). Write the symbolic microprogram for each routine as in Table 7-2. (Note that *AC* must not change in value unless the instruction specifies a change in *AC*.)

Symbol	Opcode	Symbolic Function	Description
AND	0100	$AC \leftarrow AC \wedge M[EA]$	AND
SUB	0101	$AC \leftarrow AC - M[EA]$	Subtract
ADM	0110	$M[EA] \leftarrow M[EA] + AC$	Add to memory
BTCL	0111	$AC \leftarrow AC \wedge \overline{M[EA]}$	Bit clear
BZ	1000	If $(AC = 0)$ then $(PC \leftarrow EA)$	Branch if *AC* zero
SEQ	1001	If $(AC = M[EA])$ then $(PC \leftarrow PC + 1)$	Skip if equal
BPNZ	1010	If $(AC > 0)$ then $(PC \leftarrow EA)$	Branch if positive and nonzero

7-17. Write a symbolic microprogram routine for the ISZ (increment and skip if zero) instruction defined in Chap. 5 (Table 5-4). Use the microinstruction format of Sec. 7-3. Note that $DR = 0$ status condition is not available in the *CD* field of the computer defined in Sec. 7-3. However, you can exchange *AC* and *DR* and check if $AC = 0$ with the Z bit.

7-18. Write the symbolic microprogram routines for the BSA (branch and save address) instructions defined in Chap. 5 (Table 5-4). Use the microinstruction format of Sec. 7-3. Minimize the number of microinstructions.

7-19. Show how outputs 5 and 6 of decoder F3 in Fig. 7-7 are to be connected to the program counter *PC*.

7-20. Show how a 9-bit microoperation field in a microinstruction can be divided into subfields to specify 46 microoperations. How many microoperations can be specified in one microinstruction?

7-21. A computer has 16 registers, an ALU (arithmetic logic unit) with 32 operations, and a shifter with eight operations, all connected to a common bus system.

a. Formulate a control word for a microoperation.
b. Specify the number of bits in each field of the control word and give a general encoding scheme.
c. Show the bits of the control word that specify the microoperation $R4 \leftarrow R5 + R6$.

7-22. Assume that the input logic of the microprogram sequencer of Fig. 7-8 has four inputs, I_2, I_1, I_0, T (test), and three outputs, S_1, S_0, and L. The operations that are performed in the unit are listed in the following table. Design the input logic circuit using a minimum number of gates.

I_2	I_1	I_0	Operation
0	0	0	Increment *CAR* if $T = 1$, jump to *AD* if $T = 0$
×	0	1	Jump to *AD* unconditionally
1	0	0	Increment *CAR* unconditionally
0	1	0	Jump to *AD* if $T = 1$, increment *CAR* if $T = 0$
1	1	0	Call subroutine if $T = 1$, increment *CAR* if $T = 0$
0	1	1	Return from subroutine unconditionally
1	1	1	Map external address unconditionally

7-23. Design a 7-bit combinational circuit incrementer for the microprogram sequencer of Fig. 7-8 (see Fig. 4-8). Modify the incrementer by including a control input D. When $D = 0$, the circuit increments by one, but when $D = 1$, the circuit increments by two.

7-24. Insert an exclusive-OR gate between MUX 2 and the input logic of Fig. 7-8. One input to the gate comes from the test output of the multiplexer. The other input to the gate comes from a bit labeled P (for polarity) in the microinstruction from control memory. The output of the gate goes to the input T of the input logic. What does the polarity control P accomplish?

REFERENCES

1. Dasgupta, S., *Computer Architecture: A Modern Synthesis*. Vol. 1. New York: John Wiley, 1989.
2. Gorsline, G. W., *Computer Organization: Hardware/Software*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1986.
3. Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 3rd ed. New York: McGraw-Hill, 1990.

4. Hays, J. F., *Computer Architecture and Organization*, 2nd ed. New York: McGraw-Hill, 1988.
5. Langholz, G., J. Francioni, and A. Kandel, *Elements of Computer Organization*. Englewood Cliffs, NJ: Prentice Hall, 1989.
6. Lewin, M. H., *Logic Design and Computer Organization*. Reading, MA: Addison-Wesley, 1983.
7. Mano, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.
8. Rafiquzzaman, M., and R. Chandra, *Modern Computer Architecture*. St Paul, MN: West Publishing, 1988.
9. Stallings, W., *Computer Organization and Architecture*, 2nd ed. New York: Macmillan, 1989.
10. Tanenbaum, A. S., *Structured Computer Organization*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1990.
11. Ward, S. A., and R. H. Halstead, Jr., *Computation Structures*. Cambridge, MA: MIT Press, 1990.