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.

effective address

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.

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:

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

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:

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

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

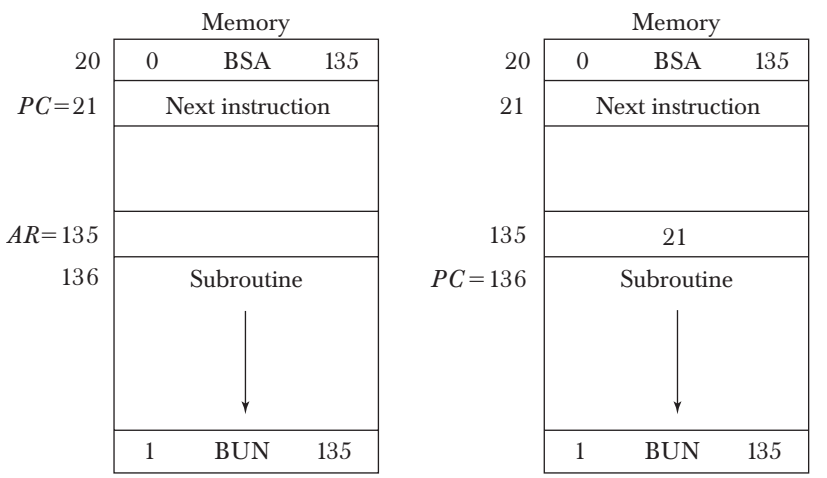
$$M[135] \leftarrow 21, \qquad PC \leftarrow 135 + 1 = 136$$

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

Figure 5-10 Example of BSA instruction execution.



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

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.

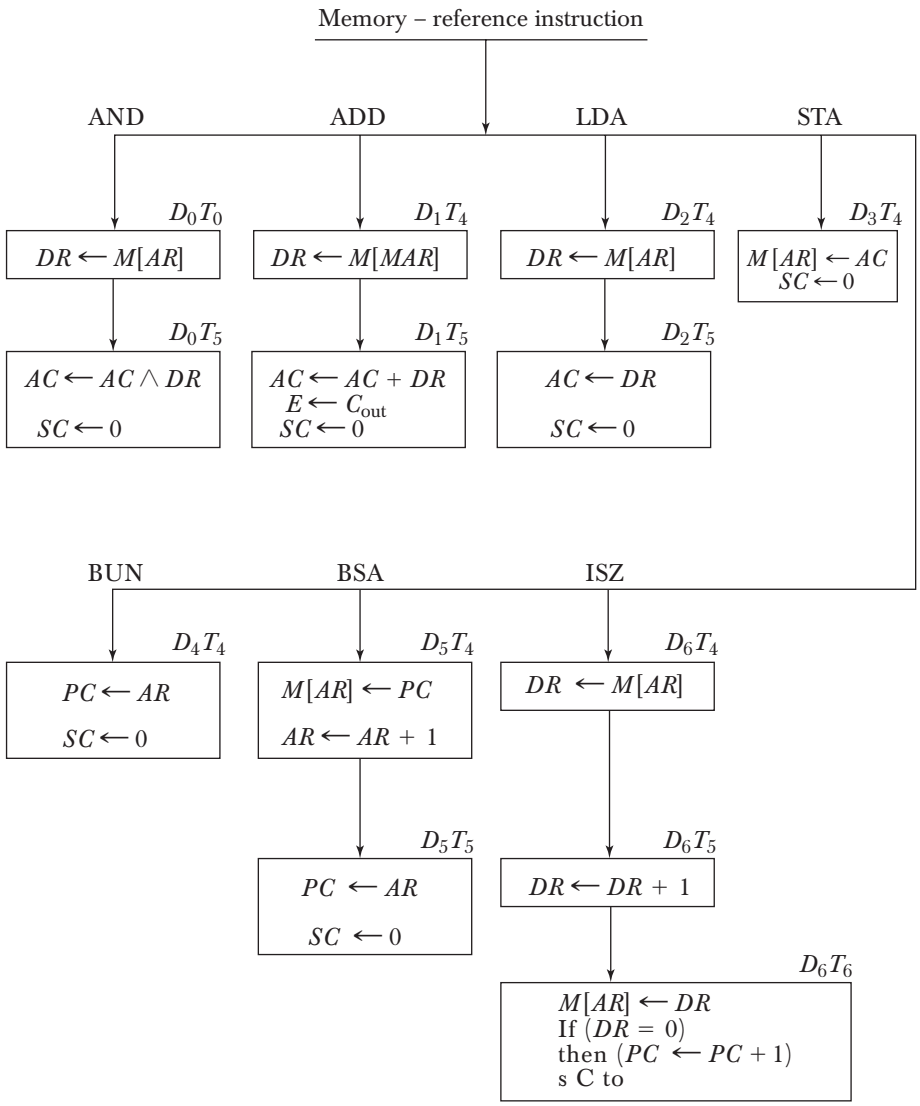


Figure 5-11 Flowchart for memory-reference instructions.

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 discussed further in Chap. 11.

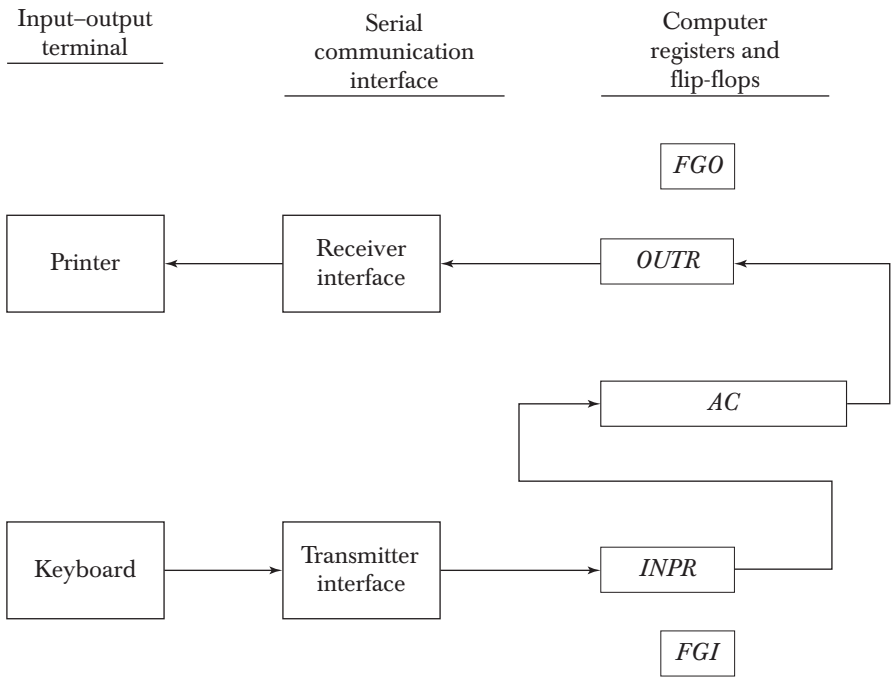
Input–Output Configuration

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 can

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