CHAPTER SIX

BASIC COMPUTER ORGANIZATION

AND DESIGN

6.1. Instruction Codes

The organization of a digital computer defined by:

1. The set of registers it contains and their function.
2. The set of instructions used.
3. The timing and control structure.

The user of a computer can control the process by means of a program. **A Program** can be defined as a set of instructions that specify the operations, operands, and the sequence by which processing has to occur.

A Computer Instruction is a binary code that specifies a sequence of microoperations for the computer.

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

The Operation Code is groups of bits that define operations such 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. For example, a computer with 32 distinct operations needs an operation code that consists of five bits.

The relationship between a computer operation and a microoperation is recognized as that, the operation code is part of instruction stored in computer memory, which 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 this reason, an operation code sometimes called a **macrooperation** because it specifies a set of microoperations.
The operation part of an instruction code specifies the operation to be performed. It is obvious that the operation must be performed on some

data stored in processor register or in memory. Therefore, the instruction code must specify 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, in addition to the type of operation to be executed.

6.2. Stored Program Organization and Addressing

Suppose that we have a computer with one processor register named **Accumulator (**abbreviated **AC)** and an instruction code format with two parts, one part for the type of operation to be performed and the second specifies an address. The memory address indicates the control where to find the operand in memory. The operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

Figure 6.1 illustrates this type of organization. For this example, it is shown that the set of instructions (programs) are stored in one section of memory and data in another section.

For a memory unit with **4096** words, we need **12** bits to specify an address since $\mathbf{2^{12} = 4096}$ and **4** bits for the operation code (Opcode) to specify one out of 16 possible operations.

To execute the program, the control reads the first 16-bit instruction from the program portion of memory. The 12-bit address part of the instruction is used to read a 16-bit operand from the data portion of memory. The control then executes the operation specified by the operation code. The operation is performed with the memory operand and the content of AC.

In case the operation indicated by the instruction code does not need an operand from memory, the 12-bits used to specify the address of the operand are not used and therefore can be used for other purpose. Examples of these operations that operate on data stored in the Accumulator register (AC) are, **clear AC, complement AC, and increment AC**.

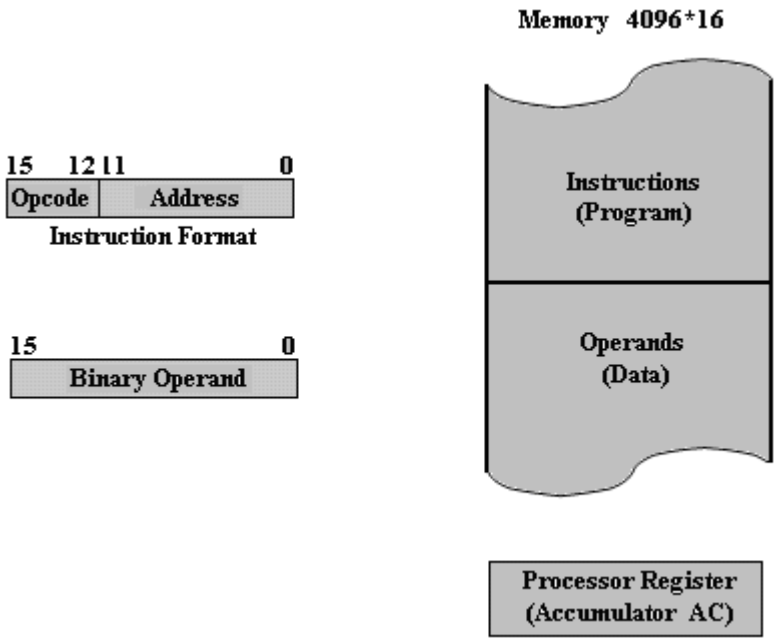


Figure 6.1

Direct and Indirect Addressing

When the second part of an instruction code specifies an operand, the instruction is said to have an **Immediate Operand** and this type of instructions are called **Immediate Instructions.**

When the second part of an instruction code specifies the address of an operand, the instruction is said to have a **Direct Address.**

When the second part of an instruction code designates an address of a memory word, in which the address of the operand is found, the instruction is said to have an **Indirect Address**.
The memory word that holds the address of the operand is used as a pointer to an array of data. The pointer could be placed in a processor register instead of memory as done in commercial computers.

To distinguish between a direct and indirect address, the most significant bit **(bit 15)** of the instruction code is used, in which **0** indicates direct address, while **1** indicates indirect address.

To illustrate these configurations, we consider the instruction code format shown in figure 6.2(a). The instruction consists of a one bit designated by **I** for **addressing mode**, 3-bit for **operation code**, and the remaining 12 bits for an address.

A direct address instruction is shown in figure 6.2(b), where **I** = **0**. This instruction is placed in address 10 in memory. The Opcode specifies an ADD instruction, and the address part is the binary equivalent of 1020.

The control finds the operand in memory at address 1020 and adds it to the content of AC.

An indirect address instruction is shown in figure 6.2(c), where **I = 1**. This instruction is placed in address 20 in memory. The Opcode specifies an ADD instruction, and the address part is the binary equivalent of 90. The control goes to the word at address 90 to find the address of the operand in memory at address 1340 and adds it to the content of AC. It is clear that, the indirect address instruction needs two references to memory to fetch an operand.

From the above examples, it is shown that there is what is called **Effective Address, which** can be defined as 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 figure 6.2(b) is **1020** and the effective address in the instruction of figure 6.2(c) is **1340.**

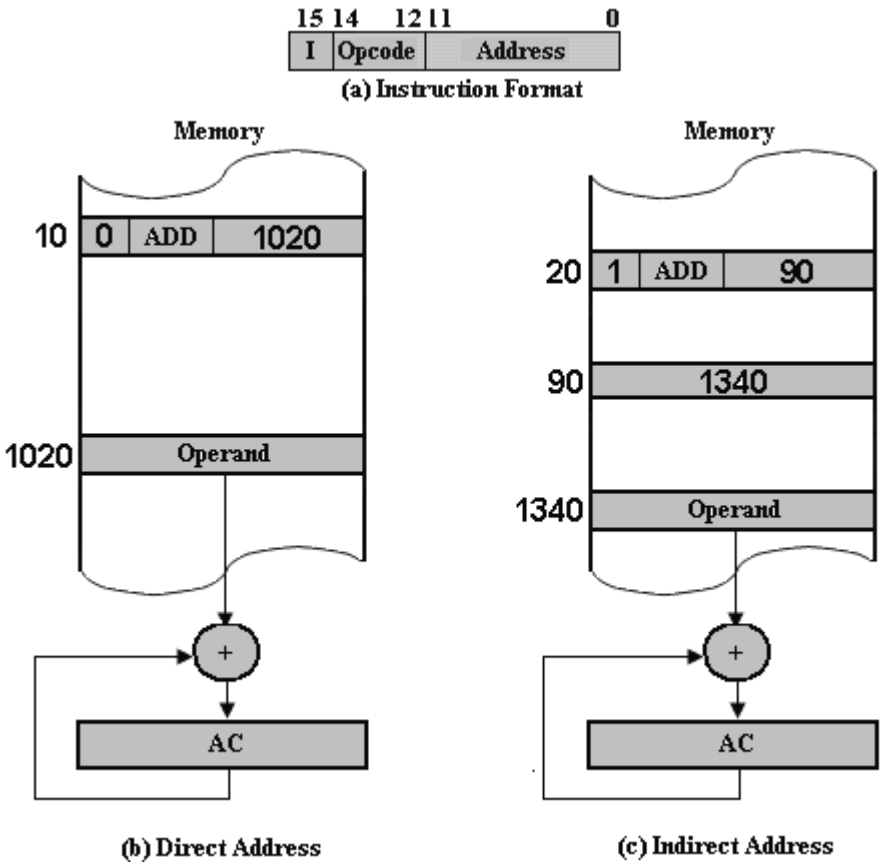


Figure 6.2

6.3. Computer Registers

As shown in the previous section, computer instructions are stored in consecutive memory locations and are executed sequentially one at a time. Therefore, in this case, a counter is needed to calculate the address of the next instruction after execution of the current instruction is completed. In addition, it is necessary to provide a register in the control unit for storing the instruction code after reading it from memory. The computer needs processor register for manipulating data and a register for holding a memory address.

Figure 6.3 shows the memory and register configuration of the Basic Computer.

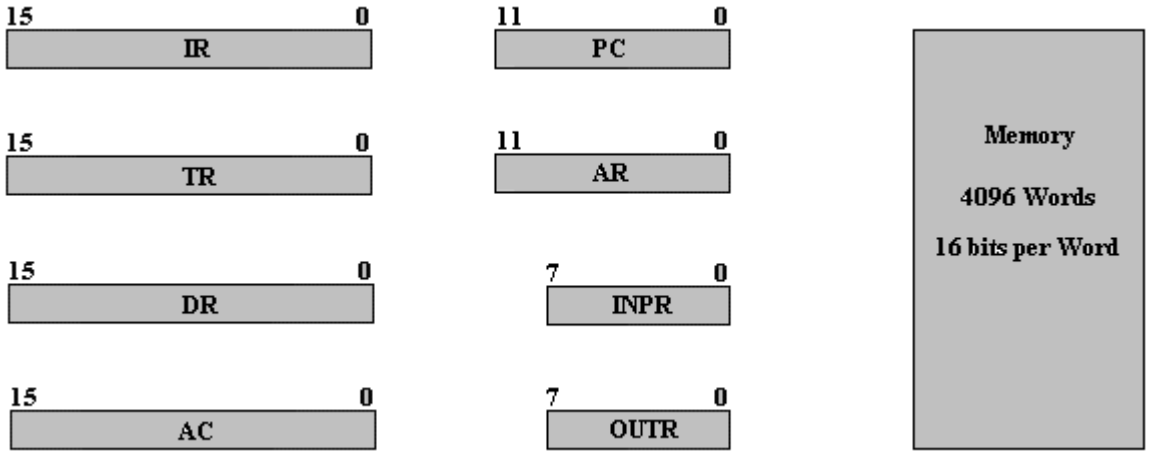


Figure 6.3

Table 6.1 lists the eight registers of the basic computer with a brief description for each.

Table 6.1

Register Symbol	Number of bits	Register Name	Register Function
DR	16	Data Register	Holds the operand read from memory
TR	16	Temporary Register	Holds temporary data during processing
AC	16	Accumulator	General purpose processor register
IR	16	Instruction Register	Holds instruction code read from Memory
PC	12	Program Counter	Holds the address of the next Instruction to be read from memory after the current instruction is executed
AR	12	Address Register	Holds Address for Memory
INPR	8	Input Register	Receives an 8-bit character from an Input device
OUTR	8	Output Register	Holds an 8-bit character for an output device

From table 6.1, it is indicated that the memory address register (AR) has 12 bits since this is the width of memory address. The program counter (PC) has also12 bits since 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.

The input and output register [(INPR) and (OUTR)] have 8 bits since each holds an 8-bits character.

Common Bus System

Figure 6.4 shows the basic computer in which it has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers via a common bus.

The outputs of six registers and memory are connected to the common bus. The selection variables S_2, S_1, and S_0 are used to select the output of one of the six registers or memory at a given time.
The number shown along each output indicates the decimal equivalent of the required binary selection. For example, the number along the output of memory unit is 7, and that along the output of register PC is 2. The 12-bit outputs of PC are placed on the bus lines when $S_2S_1S_0 = 010$.

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 (Loud) 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 and $S_2S_1S_0 = 111$. The memory places its 16-bit output onto the bus when the read input is activated.

Two registers, **AR & PC**, have 12 bits each since as mentioned before they hold a memory address. When their contents are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR & PC receive information from the common bus, only the 12 least significant bits of the bus are transferred into these registers.

Four registers, **DR, AC, IR**, and **TR**, each have 16 bits. They receive and transfer 16 bits from and to the common bus.

The input and output registers **(INPR & OUTR)** have 8 bits each and communicate with the eight least significant bits in the bus. The input register (INPR) receives a character from an input device and it is connected in such a case to provide information to the bus via the accumulator. The output register (OUTR) can only receive information from the accumulator via the bus and delivers it to an output device.

The common bus receives information from six registers (AR, PC, DR, AC, IR, and TR) and the memory, in other hand the common bus is connected to the inputs of six registers (AR, PC, DR, IR, TR, and OUTR) and the memory.

Registers (AR, PC, DR, AC, and TR) have three control inputs, LD (load register), INR (increment register), and CLR (clear or reset register), while registers (IR, and OUTR) have only one control input (LD).

The input and output data of the memory are connected to the common bus, while the memory address is connected to address register (AR). The content of any register except INPR & OUTR can be specified for the memory during a write operation. Similarly, any register can receive the data from memory after a read operation except AC & INPR.

The output of adder and logic circuit goes to the input of the 16-bit AC. There are three sets of inputs to the adder and logic circuit:

1. A set of 16-bit comes from the outputs of the accumulator AC. They are used to implement register microoperations such as complement AC, and shift AC.

2. A set of 16-bit comes 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).

3. A set of 8-bit comes from the input register INPR.

From the diagram, it is clear 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 following two microoperations can be executed at the same time.

DR ← AC and AC ← DR

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, transferring 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.

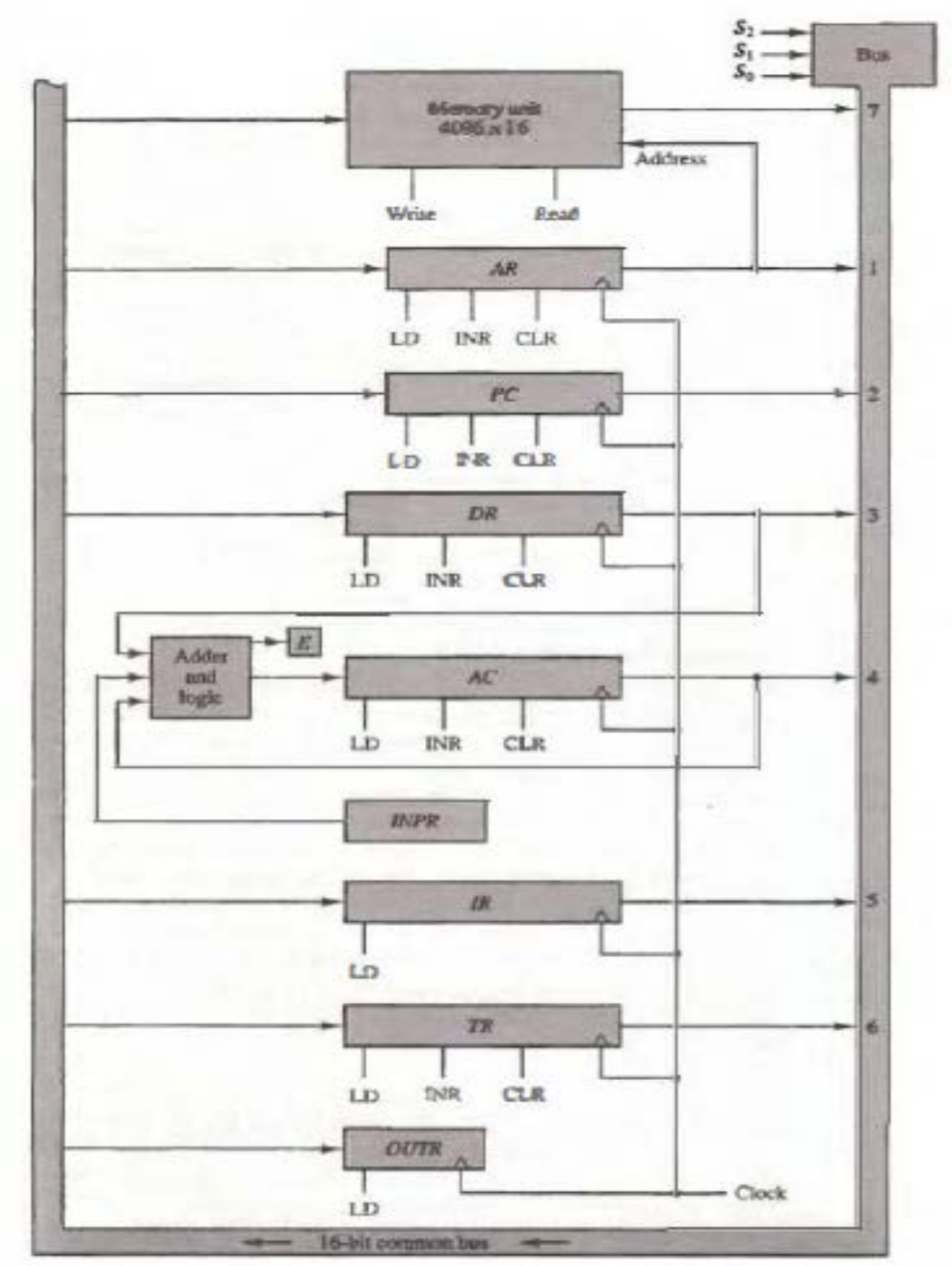


Figure 6.4

6.4. Computer Instructions

The basic computer has three instruction code formats. The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction as follows:

1. **Memory-Reference Instruction**
   For this type of instructions, the three Opcode bits in position 12 through 14 may be any combination except 111, and bit 15 is designate the addressing mode **I**. when **I** = **0** indicates direct address and when **I** = **1** indicates indirect address. The 12 bits in position 0 through 11 are used to specify an address. See figure 6.5(a).

2. **Register-Reference Instruction**
   For this type of instructions, the three Opcode bits in position 12 through 14 is equal to 111 and the bit in position 15 equals to 0. This type of instructions 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. See figure 6.5(b).

3. **Input-Output Instruction**
   For this type of instructions, the three Opcode bits in position 12 through 14 is equal to 111 and the bit in position 15 equals to 1. This type of instructions does not need a reference to memory. The remaining 12 bits are used to specify the type of input-output operation or test performed. See figure 6.5(c).

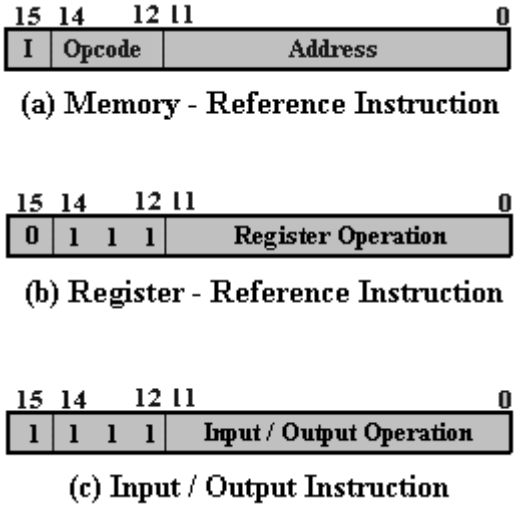


Figure 6.5

Table 6.2 shows the instructions used by the computer. Three letters are used to represent each instruction. The hexadecimal code is equal

to the equivalent hexadecimal number of the binary code used for the instruction.

Table 6.2

Instruction type	Symbol	Instruction Hexadecimal Code I = 0	I = 1	Description
Memory-Reference Instructions	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
Register - Reference Instructions	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 is positive
	SNA	7008		Skip next instruction if AC is negative
	SZA	7004		Skip next instruction if AC is zero
	SZE	7002		Skip next instruction if E is zero
	HLT	7001		Halt computer
Input – Output Instructions	INP		F000	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

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.

This set of instructions is distributed on the following categories:

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

Although the set of instruction for the basic computer listed in table 6.2 is complete, it is not efficient because some 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.

6.5. Timing and Control

A master clock generator controls the timing of all registers in the basic computer. However, 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.

There are two major types of control organization:

1. **Hardware Organization Control**

   In this type of 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. It has the disadvantage that it requires changes in the wiring among the various components if the design has to be modified or changed.

2. **Microprogrammed Organization Control**

   In this type of organization, the control information is stored in a sequence of microoperations. It has the advantage that any required changes or modifications could be done by updating the microprogram in control memory.

   In this section, we present the hardware control for the basic computer, while the microprogrammed control the same computer will be studied later.

Figure 6.6 presents the block diagram of the control unit. The control unit consists of two decoders, a sequence counter, and a number of control logic gates. The instruction register (IR) is shown in the diagram is divided into three parts:

1. Bit 15 is transferred to a flip-flop designated by the symbol I, and its function is as mentioned before.
2. Bits 12 through 14 presents the operation code are decoded by a 3 × 8 decoder. The eight outputs of the decoder are designated by the symbols **D_0 through D_7**. Note that the subscripted decimal number is equivalent to the binary value of the corresponding operation code.
3. 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 sequence counter (SC) can be incremented and cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 ×16 decoder.

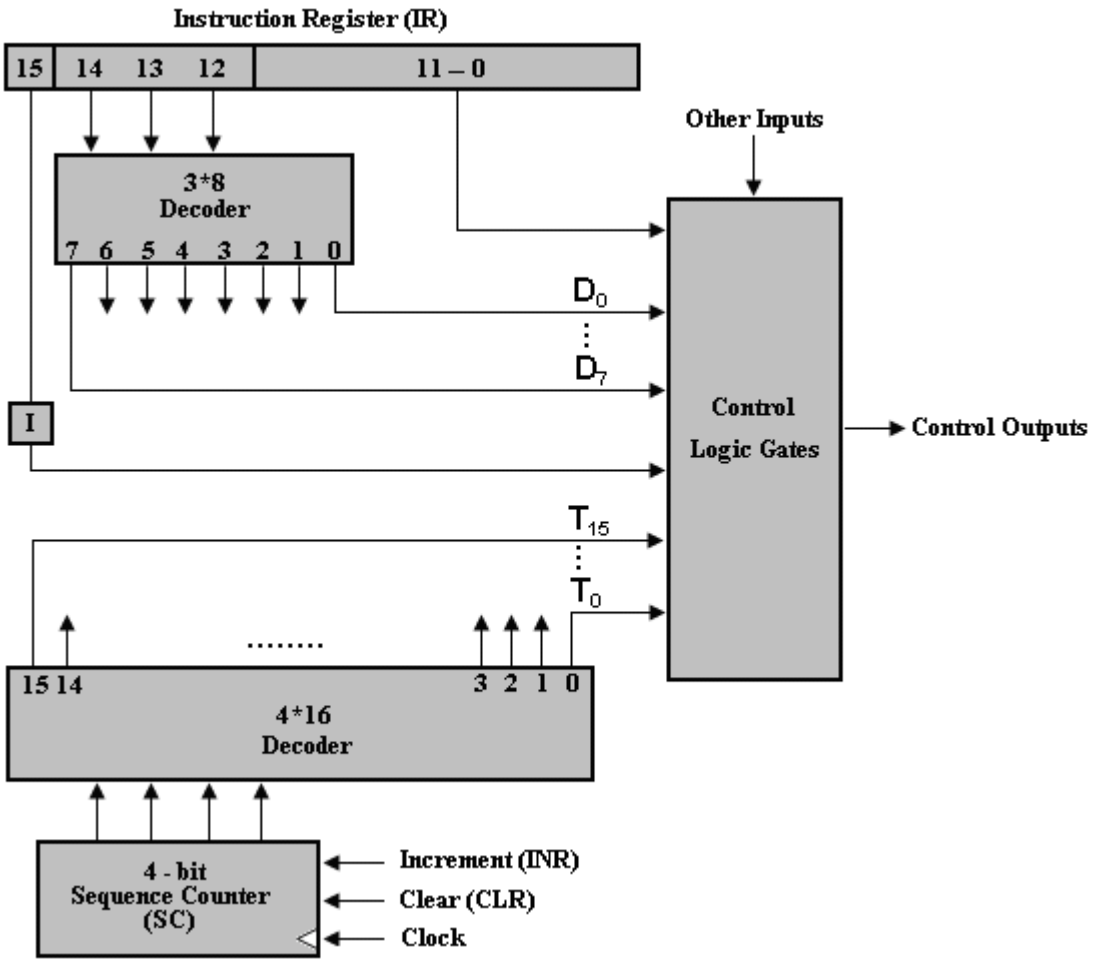


Figure 6.6

The timing diagram of figure 6.7 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 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. SC is incremented with every positive clock transition, unless its clear (CLR) input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4, and so on. If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0. For example, suppose that SC is incremented so that it provides 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 the decoder output D_3 is active. This is expressed symbolically by the statement **D_3T_4: SC ← 0**.

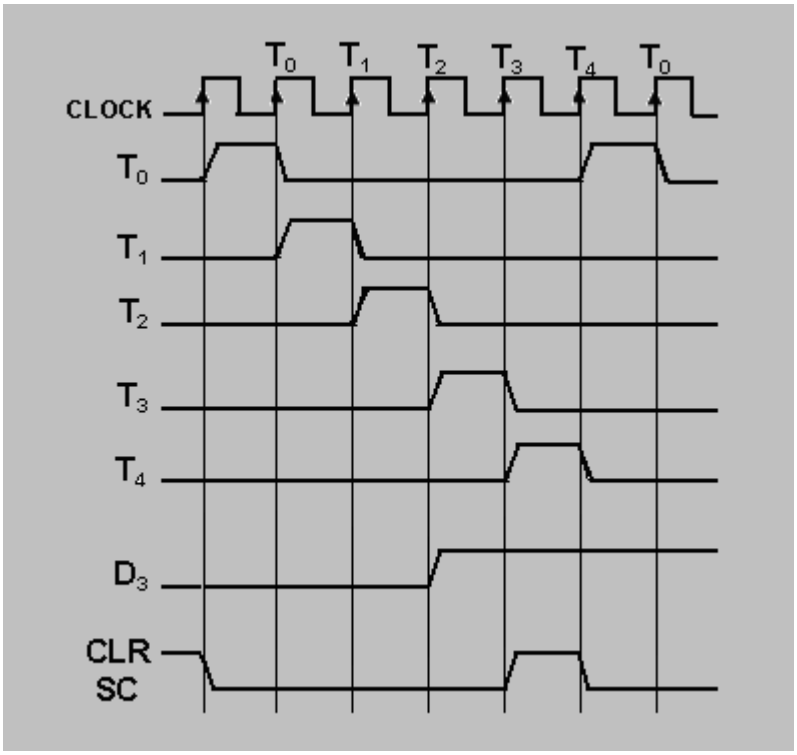


Figure 6.7

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.

To understand the operation of the computer, it's necessary to understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement **T_0: AR ← 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.

6.6. Instruction Cycle

As mention before a program consists of a sequence of instructions, and this program is residing in the computer memory. The program is executed instruction by instruction through a cycle for each instruction.

In the basic computer, each instruction cycle consists of the following phases:

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

After the completion of phase 4, the control goes back to phase 1. This process continuous indefinitely unless a HALT instruction is detected.

Fetch and Decode Phases

Initially, the program counter PC is loaded with the address of the first instruction in the program.
SC is cleared to 0, providing a decoded timing signal T_0. As mentioned before, 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, see the timing diagram shown in figure 6.7.

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 $IR(12\text{-}14),\ AR \leftarrow IR(0\text{-}11),\ I \leftarrow IR(15)$

During the clock transition associated with timing signal T_0, the address transfer from PC to AR.
During the clock transition associated with timing signal T_1, the instruction read from memory is placed in the instruction register IR the register PC is incremented by one to prepare the address of the next instruction in the program.
During the clock transition associated with timing signal T_2, the operation code in IR is decoded, the indirect bit is transferred to the flip-flop I, and the address part of the instruction is transferred to AR.

Figure 6.8 illustrates how the first two register transfer statements are implemented in the bus system.

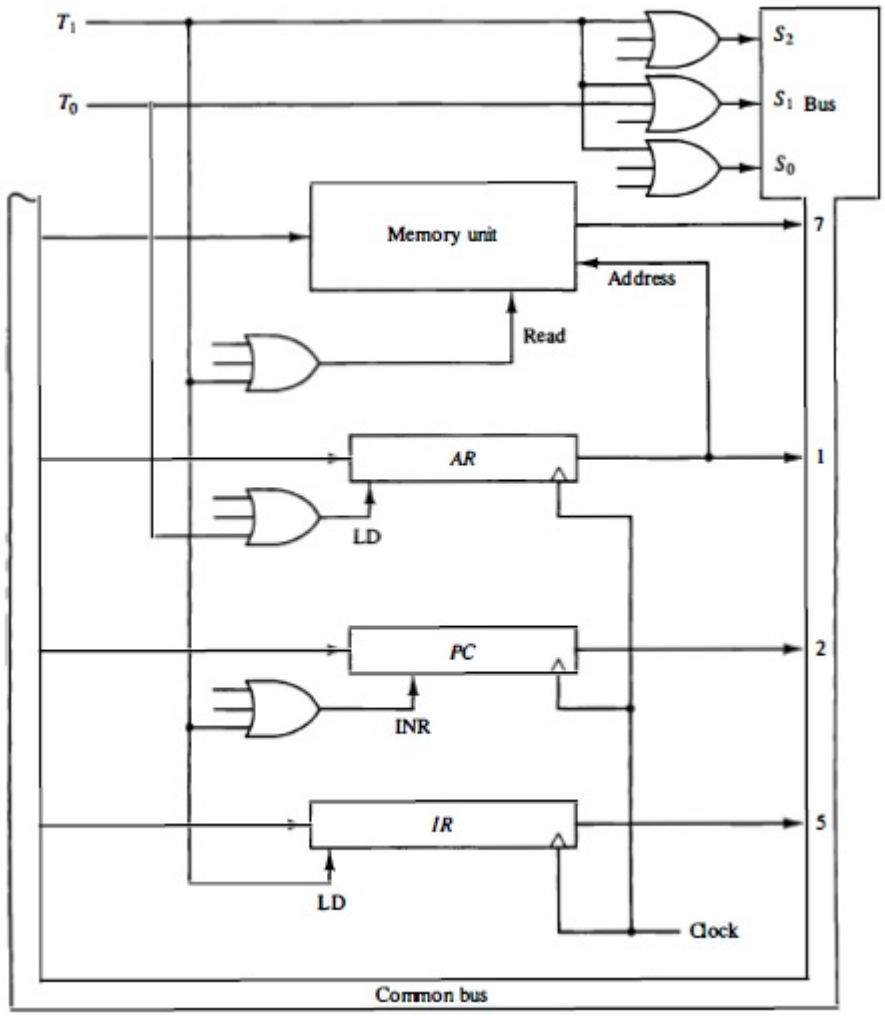


Figure 6.8

1. In order to implement the first statement

T_0: AR ← PC

It is necessary to use the timing signal **T_0** to provide the following connections in the bus system.

- **a.** Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0 = 010$.
- **b.** 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 **$T_0 = 1$**

2. In order to implement the second statement

T_1: **IR ← M [AR], PC ← PC+1**

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

a. Enable the read input of memory.

b. Place the content of memory onto the bus by making the bus selection inputs $S_2S_1S_0 = 111$.

c. Transfer the content of the bus to IR by enabling the LD input of IR.

d. Increment PC by enabling the INR input of PC.

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

The flowchart shown in figure 6.9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

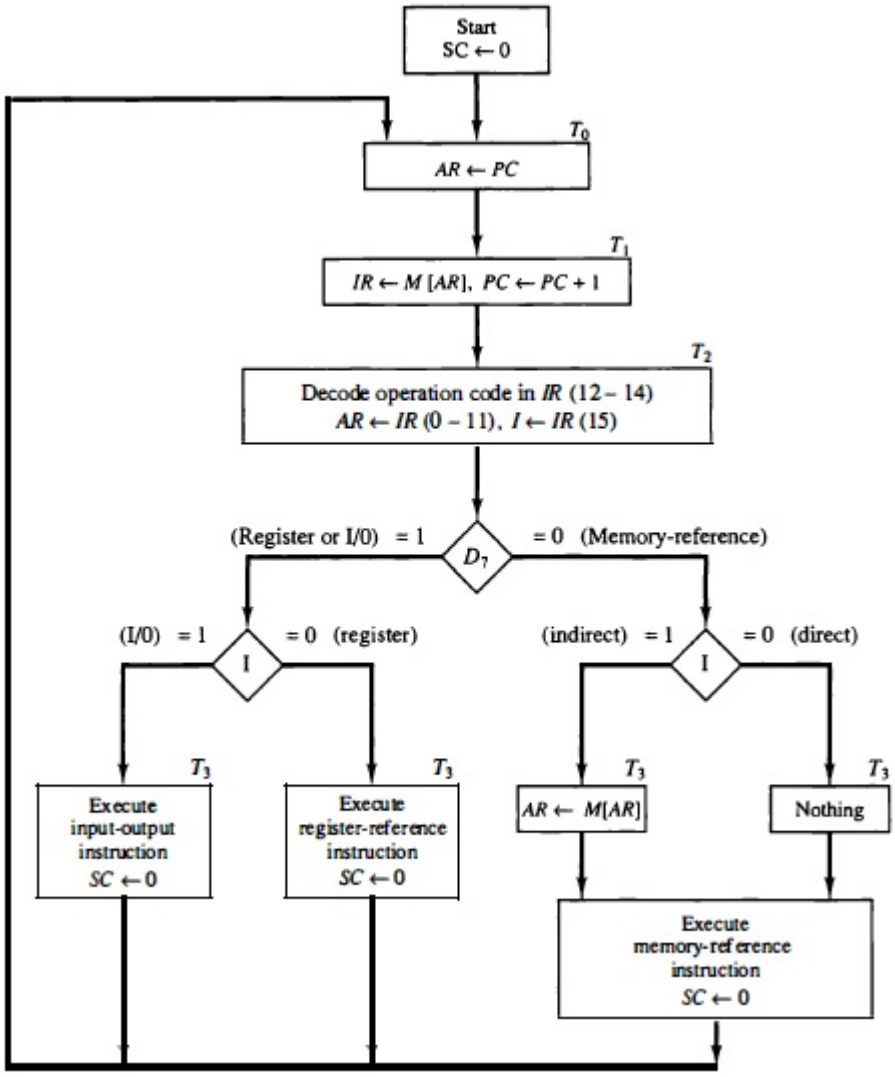


Figure 6.9

For the operation code (bits 12-14) **111 (i.e. D_7 = 1)**, the instruction must be a register or input / output type. Depending on the most significant bit (flip-flop I), when **I** = **0**, then the instruction is **register-reference instruction**, while for **I** = **1**, then the instruction is **input / output instruction.**
It is clear from the flowchart that these types of instructions are 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.

For the operation code (bits 12-14) **≠ 111 (i.e. D_7 = 0)**, one of the other seven values **000** through **110**, specifying a **memory-reference instruction.** Depending on the most significant bit (flip-flop I), when **I** = **1**, then the instruction is **a memory- reference instruction with an**

indirect address. While for **I** = **0**, then the instruction is **a memory-reference instruction with a direct address.**
For a memory-reference instruction with an indirect address, it is necessary to read the effective address from memory. The register transfer microoperation for the indirect address can be symbolized as

$$\mathbf{AR} \leftarrow \mathbf{M}\ [\mathbf{AR}]$$

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.
For a memory-reference instruction with a direct address, it is not necessary to do anything since the effective address is already in AR.
It is clear from the flowchart, for these types of instructions, the sequence counter SC must be incremented when $\overline{D}_7 T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable $\mathbf{T_4}$.. After the instruction is executed, **SC** is cleared to **0** and control returns to the fetch phase with $\mathbf{T_0}$.

Generally, the three instruction types are subdivided into four separate paths. This can be symbolized as follows: -

$\overline{D}_7 I\ T_3$: $AR \leftarrow M\ [AR]$

$\overline{D}_7 \bar{I}\ T_3$: *Nothing*

$D_7 \bar{I}\ T_3$: *Execute a register – reference instruction*

$D_7 I\ T_3$: *Execute an input – output instruction*

6.6.1. Register Reference Instructions

The control functions and microoperations for the register-reference instructions are listed in table 6.3. As shown in the flowchart of figure 6.9, these instructions are executed with the clock transition associated with timing variable $\mathbf{T_3}$**.** Each control function needs the Boolean relation $D_7 \bar{I} T_3$, 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 $\mathbf{B_i}$ to bit **i** of **IR**, all control functions can be simply denoted by $\mathbf{rB_i}$.

Example

The instruction **CMA** has the hexadecimal code **7200** (see table 6.2), which gives the binary equivalent **0111 0010 0000 0000**.

1. The first bit is **zero**, which indicates $\bar{I}$.
2. The next three bits constitute the T_0 operation code and are recognized from decoder output D_7.
3. Bit **9** in **IR** is 1 and is recognized from B_9.

Therefore, the control function that initiates the microoperation for this instruction is

$$D_7 \bar{I} T_3 B_9 = r B_9$$

The execution of a register-reference instruction is completed at time T_3. The sequence counter **SC** is cleared to **zero** and the control goes back to fetch the next instruction with timing signal T_0.

Table 6.3

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

6.6.2. Memory-Reference Instructions

Table 6.4 shows the seven memory-reference instructions. The first column in the table shows the symbol of the instruction. The second column shows the operation decoder output D_i that belongs to each instruction. 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 third column shows the symbolic description of each instruction which is specified in the table in terms of register transfer notation.

Actually, the execution of the memory-reference instruction in the bus system will require a sequence of microoperations since data stored in memory cannot be processed directly.

Table 6.4

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \cap M[AR]$
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 $PC \leftarrow PC + 1$

We now explain the operation of the seven instructions and list the control functions and microoperations needed for their execution.

AND (AND with accumulator) Instruction

This instruction performs the **AND** logic operation between the contents of the **AC** and the memory word specified by the effective address then transferring the result in **AC**.

The microoperations needed to execute the AND instruction are:

D_0T_4: $DR \leftarrow M[AR]$

D_0T_5: $AC \leftarrow AC \cap DR$, $SC \leftarrow 0$

The operation decoder D_0 is active when the instruction has an **AND** operation whose binary code value is **000.**

To execute the AND instruction, two timing signals are needed: -

1. The clock transition associated with timing signal T_4 transfers the operand from memory into **DR**.

2. The clock transition associated with timing signal T_5 transfers the result of the **AND** logic operation between the contents of **DR** and **AC** into **AC**. In the same clock transition, **SC** is cleared to 0 which transfers the control to timing signal T_0 to start a new instruction cycle.

ADD (ADD to accumulator) Instruction

This instruction adds the content of the memory word specified by the effective address to the contents of the **AC** then transferring the result into **AC** and the output carry C_{out} is transferred to the **E** flip-flop (extended accumulator).

The microoperations needed to execute the ADD instruction are:

D_1T_4: $DR \leftarrow M[AR]$

D_1T_5: $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$

The operation decoder D_1 is active when the instruction has an **ADD** operation whose binary code value is **001.**

To execute the ADD instruction, two timing signals are needed: -

1. The clock transition associated with timing signal T_4 transfers the operand from memory into **DR**.

2. The clock transition associated with timing signal T_5 transfers the result of the addition of the contents of **DR** and **AC** into **AC,** in addition, C_{out} is transferred into the flip-flop **E.** In the same clock transition, **SC** is cleared to 0 which transfers the control to timing signal T_0 to start a new instruction cycle.

LDA (LoaD Accumulator) Instruction

This instruction transfers the content of the memory word specified by the effective address to **AC**.

The microoperations needed to execute the LDA instruction are:

D_2T_4: $DR \leftarrow M[AR]$

D_2T_5: $AC \leftarrow DR$, $SC \leftarrow 0$

The operation decoder **D2** is active when the instruction has an **ADD** operation whose binary code value is **010.**

To execute the LDA instruction, two timing signals are needed: -

1. The clock transition associated with timing signal T_4 transfers the operand from memory into **DR**.

2. The clock transition associated with timing signal T_5 transfers the contents of **DR** into **AC** via the **Adder and Logic Circuit.** In the same clock transition, **SC** is cleared to 0 which transfers the control to timing signal T_0 to start a new instruction cycle.

STA (STore Accumulator) Instruction

This instruction stores the content of **AC** into the memory word specified by the effective address.

One microoperation needed to execute the STA instruction:

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

The operation decoder D_3 is active when the instruction has an **STA** operation whose binary code value is **011.**

To execute the **STA** instruction, one timing signal T_4 is needed since, the output of **AC** is applied to the bus and the data input of memory is connected to the same bus. In the same clock transition, **SC** is cleared to **0** which transfers the control to timing signal T_0 to start a new instruction cycle.

BUN (Branch UNconditionally) Instruction

This instruction allows the programmer to specify an instruction out of the program instructions sequence (i.e. the program branches or jumps unconditionally).

One microoperation needed to execute the BUN instruction:

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

The operation decoder D_4 is active when the instruction has a **BUN** operation whose binary code value is **100.**

To execute the **BUN** instruction, one timing signal T_4 is needed since, the effective address from **AR** is transferred through the common bus to **PC**. In the same clock transition, **SC** is cleared to **0** which transfers the control to timing signal T_0 to start a new instruction cycle.

BSA (Branch and Save return Address) Instruction

This instruction allows the programmer to branch to a portion of the program called a subroutine or procedure.

The microoperations needed to execute the BSA instruction are:

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

D_5T_5: $PC \leftarrow AR, \quad SC \leftarrow 0$

The operation decoder D_5 is active when the instruction has an **BSA** operation whose binary code value is **101.**

To execute the BSA instruction, two timing signals are needed: -

1. The clock transition associated with timing signal T_4 initiates a memory write operation, places the content of **PC** onto the bus, and enables the **Wright** control signal of memory and **INR** input of **AR**. The memory write operation is completed and **AR** is incremented by the time the next clock transition occurs.

2. The clock transition associated with timing signal T_5 transfers the contents of **AR** into **PC.** In the same clock transition, **SC** is cleared to **0** which transfers the control to timing signal T_0 to start a new instruction cycle.

EXAMPLE

Assume the **BSA** instruction is stored in a memory word at address 30. The bit (**I = 0)** and the address part of the instruction is the binary equivalent **170**.

Execution of the instruction

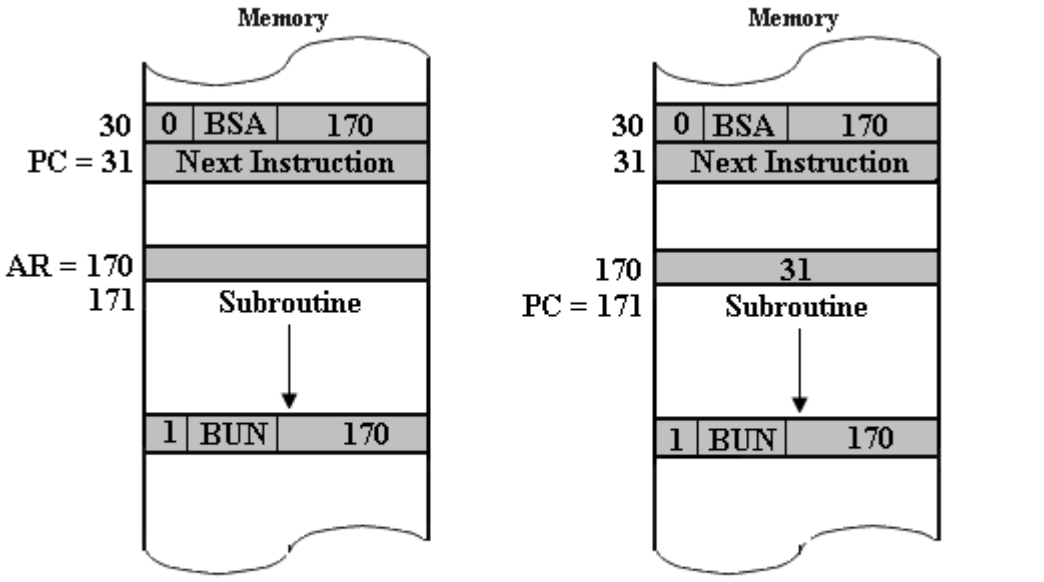
After the fetch and decode phase, **PC** contains **31,** which is the address of the next instruction in the program (the return address). The register **AR** holds the effective address **170** [see figure 6.10(a)]. The **BSA** instruction performs the following numerical operation:

$M[170] \leftarrow 31, \; PC \leftarrow 170 + 1 = 171$

The result of this operation is shown in figure 6.10(b). The return address (31) is stored in memory location **170** and control continues with the subroutine program starting from address **171**.

To return to the original program (at address 31), this is accomplished by means of an indirect **BUN** instruction placed at the end of the subroutine.

When **BUN** is executed, control goes to the indirect phase to read the effective address at location **170**, where it finds the previously saved address **31**. Then the effective address **31** is transferred to **PC.** The next instruction cycle finds **PC** with the value **31**, so control continues to execute the instruction at the return address.



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

Figure 6.10

ISZ (Increment and Skip if Zero) Instruction

This instruction increments the word specified by the effective address, and if the result is zero, **PC** is incremented by one. When **PC** is incremented by one, the next instruction in the sequence is skipped.

The microoperations needed to execute the ISZ instruction are:

D_6T_4: **DR ← M [AR]**

D_6T_5: **DR ← DR + 1**

D_6T_6: **M [AR] ← DR, if (DR = 0) then (PC $\overset{T_1}{\leftarrow}$ PC + 1), SC ← 0**

The operation decoder D_6 is active when the instruction has an **ISZ** operation whose binary code value is **110.**

To execute the ISZ instruction, three timing signals are needed: -

1. The clock transition associated with timing signal T_4 read the memory into **DR**.

2. The clock transition associated with timing signal T_5 increments **DR.**

3. The clock transition associated with timing signal T_6 store the word back into memory. In the same clock transition, **SC** is cleared to 0 which transfers the control to timing signal T_0 to start a new instruction cycle.

6.7. Input-Output and Interrupt

Computer systems include many types of input and output devices. To demonstrate the most basic requirements for input and output communication, a terminal unit with a **keyboard** and **printer** used for this illustration.

6.7.1. Input-Output Configuration

We mentioned before that the terminals send and receive serial information. Each type of this information has eight bits of an alphanumeric code. The serial information from the keyboard shifted into the input register **INPR**. The serial information for the printer is stored in the output register **OUTR**.

The two registers **INPR & OUTR** communicate with an interface serially, and with an accumulator **AC** in parallel.

Figure 6.11 shows the input-output configuration. Where the transmitter interface receives serial information from the keyboard and transmits it to **INPR,** while the receiver interface receives information from **OUTR** and sends it serially to the printer**.**
The input and output registers **INPR & OUTR** respectively are 8-bits. They hold alphanumeric information. The input and output flags **FGI & FGO** are 1-bit control flip-flops. The flag **FGI** is 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.

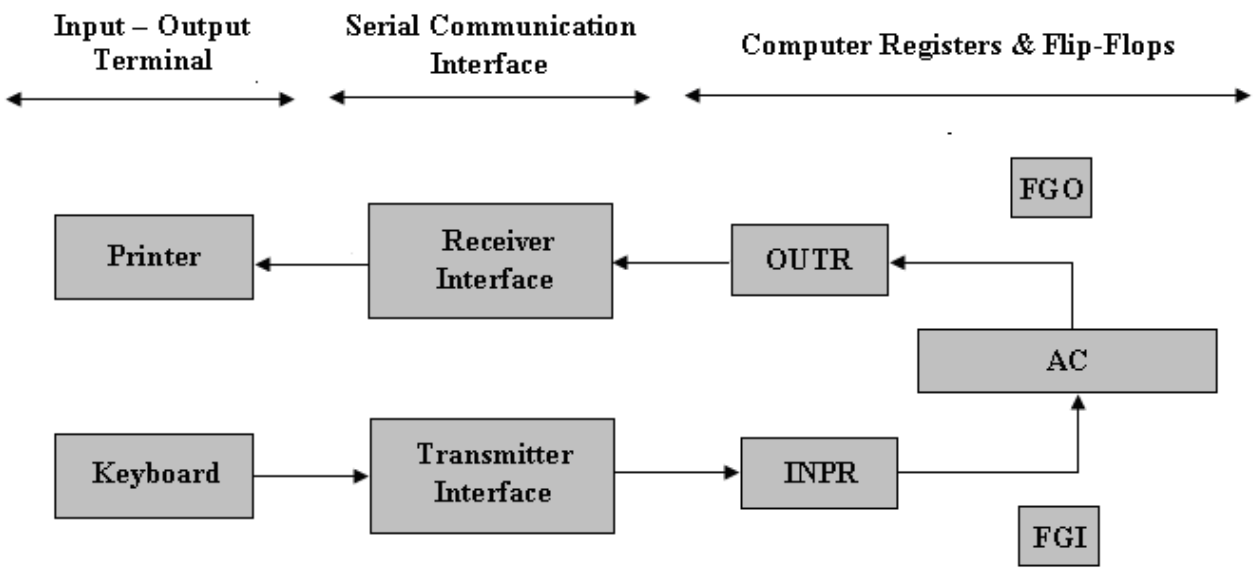


Figure 6.11

Transfer of information from the keyboard to the computer

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 cleared, new information shifted into INPR by striking another key.

Transfer of information from the computer to the printer

The output register **OUTR** works similar to ***INPR,*** but the direction of information flow reversed.

Initially the output flag FGO is set to 1. The computer checks the output flag; 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.

6.7.2. Input-Output Instructions

As mentioned before, input-output instructions have an operation code **1111** are recognized by the control ($\mathbf{D_7 = 1}$ **& I = 1).** Bits (0 – 11) of the instruction specify the particular operation.

The control functions and microoperations for the input-output instructions are listed in table 6.5. As shown in the flowchart of figure 6.9, these instructions are executed with the clock transition associated with timing variable $\mathbf{T_3}$**.** Each control function needs the Boolean relation $D_7\, I\, T_3$, 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 $\mathbf{B_i}$ to bit **i** of **IR**, all control functions can be simply denoted by $\mathbf{pB_i}$ for **i** = **6** through **11**. The sequence counter **SC** is cleared to **0** when **p** = $D_7\, I\, T_3$ = **1**.

Table 6.5

Symbol	Microoperation	Description
	p: SC ←0	**Clear SC**
INP	**pB_{11}: AC(0-7) ← INPR, FGI←0**	**Input character**
OUT	**pB_{10}: OUTR ← AC(0-7), FGO ←0**	**Output character**
SKI	**pB_9: if (FGI = 1) then (PC ← PC + 1)**	**Skip on input flag**
SKO	**pB_8: if (FGO = 1) then (PC ← PC + 1)**	**Skip on output flag**
ION	**pB_7: IEN ← 1**	**Interrupt enable on**
IOF	**pB_6: IEN ← 0**	**Interrupt enable off**

Example

The instruction **INP** has the hexadecimal code **F800** (see table 6.2), which gives the binary equivalent **1111 1000 0000 0000**.

1. The first bit is **one**, which indicates I.
2. The next three bits constitute the operation code and are recognized from decoder output $\mathbf{D_7}$**.**
3. Bit **11** in **IR** is **1** and is recognized from $\mathbf{B_{11}}$. $\mathbf{T_2}$

Therefore the control function that initiates the microoperation for the instruction INP is

$$\mathbf{D_7 I\, T_3\, B_{11} = p\, B_{11}}$$

The execution of input-output instructions completed at time T_3 as in the case of the register-reference instructions. The sequence counter **SC** is cleared to **0** and the control goes back to fetch the next instruction with timing signal T_0.

The **INP** instruction transfers the input information from **INPR** into the eight least significant bits of accumulator and clears the input flag to **0**.

The **OUT** instruction transfers the eight least significant bits of accumulator into the output register **OUTR** and clears the output flag to **0**.

The instructions **SKI & SKO** checks the status of the flags **FGI & FGO** respectively and causes a skip of the next instructions if the flag is **1**. The instruction that is skipped will normally be a branch instruction to return and check the flag again.

The instructions **ION & IOF** set and clear an interrupt enable flip-flop **IEN**. The purpose of the flip-flop **IEN** will be explained later in conjunction with the interrupt operation.

6.7.3. Program Interrupt

The difference of information flow rate between the computer and the input/output device according to the **programmed control transfer procedure** mentioned before makes this type of transfer inefficient.

The alternative efficient procedure is to let the external device to inform the computer when it is ready for the transfer. This type of transfer uses the **interrupt** facility.

While the program running, it does not check the input or output flags. However, when a flag is set, the computer shortly interrupted from proceeding with the current program and is informed of the fact that a flag has been set. In that, time the computer deviates shortly from what it is doing to take care of the input or output transfer. After completing the input or output transfer, the computer returns to the current program to continue what it was doing before the interrupt.

The interrupt flip-flop **IEN** can be set and cleared according to two instructions as follows:

1. **With the IOF instruction, the IEN is cleared to 0 and the input or output flags cannot interrupt the computer.**

2. **With the ION instruction, the IEN is set to 1, and the computer can be interrupted.**

How the computer handles the interrupt process

The flowchart shown in figure 6.12 explains the process. The computer contains an interrupt flip-flop designated **by R**.

1. When **R** = **0**, the computer goes through instruction cycle. During the instruction cycle, **IEN** is checked, if it is **0**, indicates no need for interruption, and the 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 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.

2. When **R** = **1**, the computer goes through interrupt cycle, which 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. Here a memory location at address 0 is chosen 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.

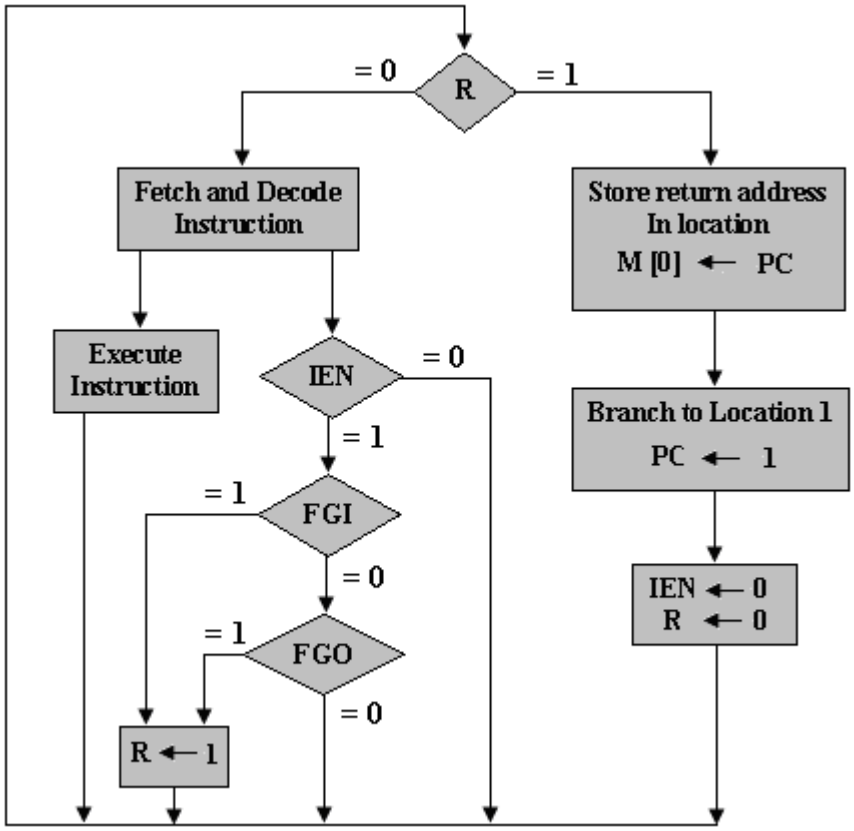


Figure 6.12

Example. (See Figure 6.13)

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 **BUN 1120** instruction at address **1**, **see figure 6.13(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, **see figure 6.13(b).**

A branch indirect instruction with an address part of **0** placed at the end of the **I/O** program, returns the computer to the original place in

the main program. After this instruction is 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**.

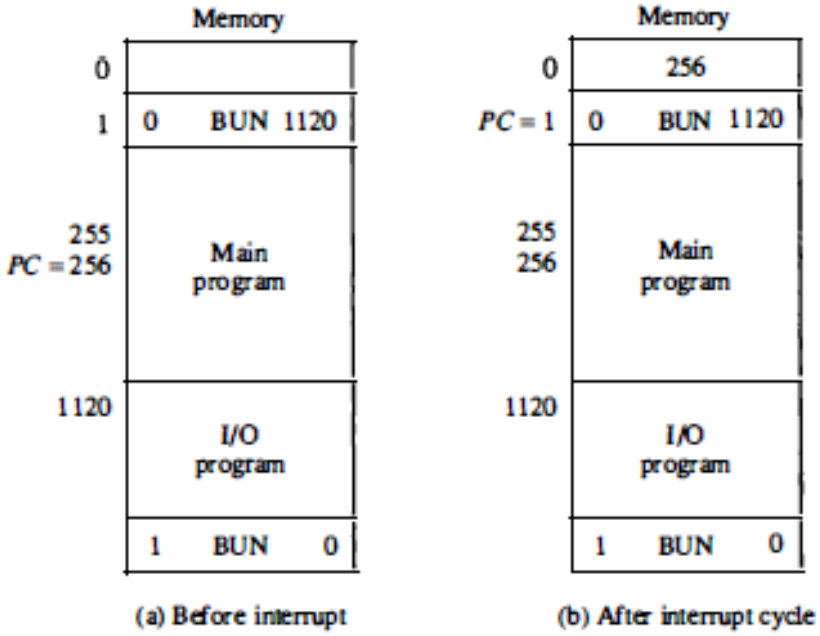


Figure 6.13

6.7.4. Interrupt Cycle

From the flowchart shown in figure 6.12, it is clear that the condition for setting flip-flop **R** to **1** can be expressed with the following register transfer statement:

$$\overline{T}_0\overline{T}_1\overline{T}_2(IEN)(FGI + FGO): \quad R \leftarrow 1$$

The interrupt cycle stores the return address which is 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:

$$RT_0: \quad AR \leftarrow 0, \quad TR \leftarrow PC$$
$$RT_1: \quad M[AR] \leftarrow TR, \quad PC \leftarrow 0$$
$$RT_2: \quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0$$

1. **During timing signal T_0**, **AR** is cleared to **0**, and the content of **PC** is transferred to the temporary register **TR**.

2. **During timing signal T_1**, the return address is stored in memory at location **0** and **PC** is cleared to **0**.

3. **During timing signal T_3**, the **PC** incremented by **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 $\overline{R}T_0$ 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**.

6.8. Complete Computer Description

Figure 6.14 shows the final flowchart of the instruction cycle, including the interrupt cycle for the basic computer. As mentioned before the control returns to timing signal, T_0 after **SC** is cleared to **0**. if **R** = **1**, the computer executes an interrupt cycle, while for **R** = **0**, the computer executes an instruction cycle.

Table 6.6 summarizes the control functions and microoperations for the entire basic computer.

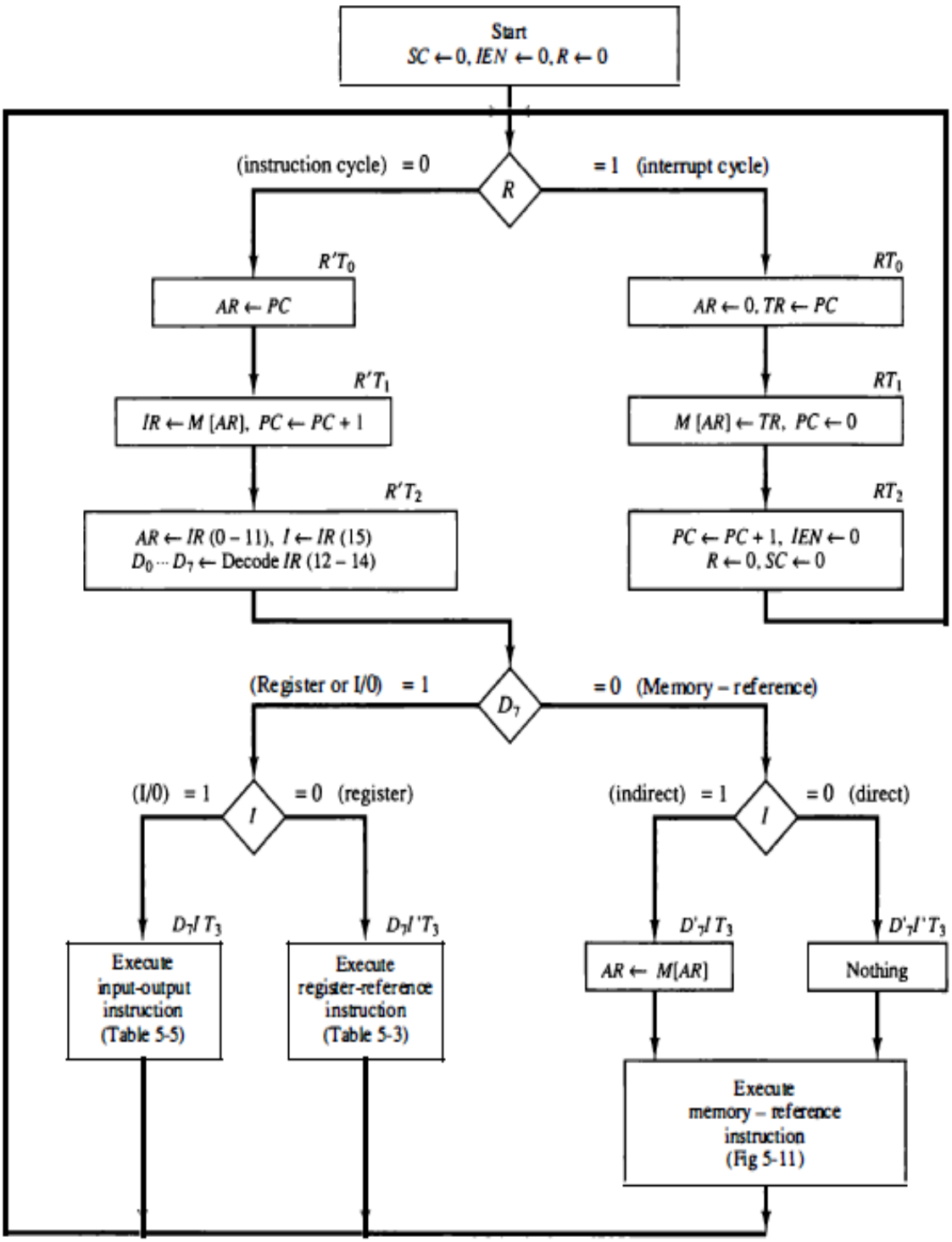

Start
SC ← 0, IEN ← 0, R ← 0
(instruction cycle) = 0
R
= 1 (interrupt cycle)
R'T0
AR ← PC
RT0
AR ← 0, TR ← PC
R'T1
IR ← M [AR], PC ← PC + 1
RT1
M [AR] ← TR, PC ← 0
R'T2
AR ← IR (0 – 11), I ← IR (15)
D0 ··· D7 ← Decode IR (12 – 14)
RT2
PC ← PC + 1, IEN ← 0
R ← 0, SC ← 0
(Register or I/0) = 1
D7
= 0 (Memory – reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
D7I T3
Execute input-output instruction (Table 5-5)
D7I'T3
Execute register-reference instruction (Table 5-3)
D'7I T3
AR ← M[AR]
D'7I'T3
Nothing
Execute memory – reference instruction (Fig 5-11)

Figure 6.14

Table 6.6

	Description	Microoperation
	Fetch	**$R'T_0$: AR ← PC** **$R'T_1$: IR ← M [AR], PC ← PC+1**
	Decode	**$R'T_2$: $D_0, \ldots, D_7$ ← Decode IR(12-14),** **AR ← IR(0-11), I ← IR(15)**
	Indirect	**D'_7IT_3: AR ← M [AR]**
	Interrupt	**$T'_0T'_1T'_2$(FGI + FGO): R ← 1** **RT_0: AR ← 0, TR ← PC** **RT_1: M[AR] ← TR, PC ← 0** **RT_2: PC ← PC+1, IEN ← 0, R ← 0, SC ← 0**
Memory –Reference Instructions	**AND**	**D_0T_4: DR ← M [AR]** **D_0T_5: AC ← AC ∩ DR, SC ← 0**
	ADD	**D_1T_4: DR ← M [AR]** **D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0**
	LDA	**D_2T_4: DR ← M [AR]** **D_2T_5: AC ← DR, SC ← 0**
	STA	**D_3T_4: M [AR] ← AC, SC ← 0**
	BUN	**D_4T_4: PC ← AR, SC ← 0**
	BSA	**D_5T_4: M [AR] ← PC, AR ← AR + 1** **D_5T_5: PC ← AR, SC ← 0**
	ISZ	**D_6T_4: DR ← M [AR]** **D_6T_5: DR ← DR + 1** **D_6T_6: M [AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0**
Register-Reference Instructions		**r: SC ←0**
	CLA	**rB_{11}: AC ←0**
	CMA	**rB_{10}: E ←0**
	CMA	**rB_9: AC ← $\overline{AC}$**
	CME	**rB_8: E ← $\overline{E}$**
	CIR	**rB_7: AC ← shr AC, AC(15) ←E, E ← AC(0)**
	CIL	**rB_6: AC ← shl AC, AC(0) ←E, E ← AC(15)**
	INC	**rB_5: AC ← AC + 1**
	SPA	**rB_4: If (AC(15) = 0) then (PC← PC + 1)**
	SNA	**rB_3: If (AC(15) = 1) then (PC← PC + 1)**
	SZA	**rB_2: If (AC = 0) then (PC← PC + 1)**
	SZE	**rB_1: If (E = 0) then (PC← PC + 1)**
	HLT	**rB_0: S←0 (S is a start-stop flip-flop)**
Input / Output Instructions		**p: SC ←0**
	INP	**pB_{11}: AC(0-7) ← INPR, FGI←0**
	OUT	**pB_{10}: OUTR ← AC(0-7), FGO ←0**
	SKI	**pB_9: if (FGI = 1) then (PC ← PC + 1)**
	SKO	**pB_8: if (FGO = 1) then (PC ← PC + 1)**
	ION	**pB_7: IEN ← 1**
	IOF	**pB_6: IEN ← 0**

6.9. Design of the Basic Computer

The hardware of the basic computer consists of the following parts:

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:* ***S, E, R, IEN, FGI, FGO, and I.***
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, registers, flip-flops, **3 × 8** and **4 ×16** decoders, and the16-bit common bus) have been discussed in details previously in this chapter. These parts can be obtained from a commercial source.
Now we are going to design the remaining parts, Control logic gates, and the adder and logic circuit associated with accumulator.

6.9.1. Design of the Control logic gates

Returning to figure 6.6, which shows some of the inputs to the control logic gates which comes from the **two decoders, I flip-flop, and bits 0-11 of IR**. **Other inputs to control logic gates which are not shown in the figure are:**

1. Accumulator 16 bits to check if AC = 0 and to detect the sign bit in AC (15).
2. Data register DR 16 bits to check if DR = 0.
3. Value of the seven flip-flops S, E, R, IEN, FGI, FGO, and I.

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 to S_2, S_1, and S_0 To select a register or memory for the bus.
5. Signals to control the AC adder and logic circuit.

6.9.1.1. Control of Memory and Registers

The control inputs of the registers connected to the common bus (see figure 6.4) are:

1. Load the register (**LD**).
2. Increment the register (**INR**).
3. Clear the register (**CLR**).

Suppose that it is desired to derive the gate structure associated with the control inputs of the data register (**DR**). By scanning table 6.6, it is shown that the statements that change the content of **DR** are the following:

D_0T_4: DR ← M [AR]
D_1T_4: DR ← M [AR]
D_2T_4: DR ← M [AR]
D_6T_4: DR ← M [AR]
D_6T_5: DR ← DR + 1

The first four statements specify transfer of information from the memory to **DR.** The content of memory is placed on the bus and the content of the bus is transferred into **DR** by enabling its **LD** control input. The fifth statement increments the data register **(DR)** by 1. The control functions can be combined into two Boolean expressions as follows:

$$LD\,(DR) = (D_0 + D_1 + D_2 + D_6)T_4$$
$$INR\,(DR) = D_6T_5$$

Figure 6.15 shows the control gate logic associated with **DR.**

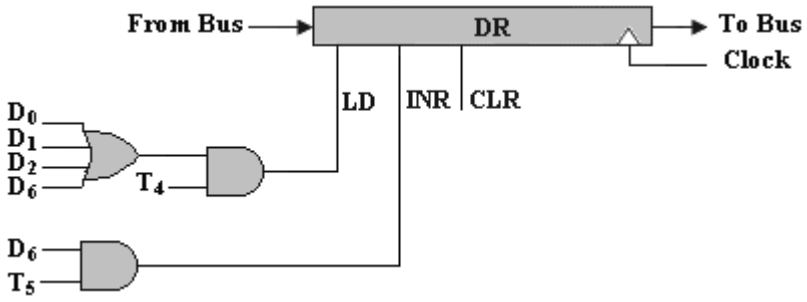


Figure 6.15

In similar way, one 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 write input of memory is derived by scanning table 6.6 to find the statements that specify a write operation. Note that the write operation is recognized from the symbol M **[AR]** ←**.**

$$Write = RT_1 + D_3T_4 + D_5T_4 + D_6T_6$$

6.9.1.2. Control of Single Flip-Flops

In similar manner, the control gates for the seven flip-flops S, E, R, IEN, FGI, FGO, and I can be determined.

Example

Show the complete logic control of the **IEN** Flip-flop in the basic computer. Use a **JK** flip-flop for this purpose.

Solution

By scanning table 6.6, it is shown that the statements that change the state of the Flip-flop **IEN** are the following:

$$pB_7: \quad IEN \leftarrow 1$$
$$pB_6: \quad IEN \leftarrow 0$$
$$RT_2: \quad IEN \leftarrow 0$$

Where $p = D_7 I T_3$ and B_6 and B_7 are bits 6 and 7 respectively.

Using **JK** flip-flop for the **IEN**, the complete control logic will be as shown in figure 6.16.

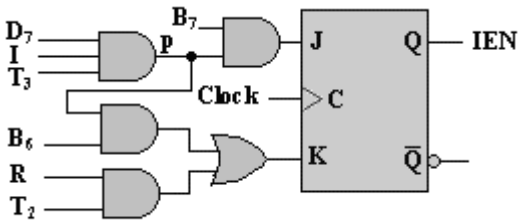


Figure 6.16

6.9.1.3. Control of Common Bus

As explained before, the 16-bit common bus is controlled by the three selection inputs **S_2, S_1, and S_0** (see figure 6.4). To select any one of the registers or the memory, a binary number equivalent to the decimal number shown with each bus input must be applied to the selection inputs **S_2, S_1, and S_0** in order to select the corresponding register or memory.

Table 6.7 is recognized as the truth table of a binary encoder, which specifies the binary numbers for **$S_2S_1S_0$** that select each register or memory. Each binary number is associated with a Boolean variable **X_1 to X_7**, corresponding to the gate structure that must be active in order to select the register or memory for the bus.

As an example, when **X_3 = 1**, the corresponding value of **$S_2S_1S_0$** must be **011** and the output of **DR** will be selected for the bus.

Table 6.7

Inputs X_1 X_2 X_3 X_4 X_5 X_6 X_7	Outputs S_2 S_1 S_0	Selected Register for the Bus
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

Figure 6.17 shows the encoder at the inputs of the bus selection logic. The Boolean functions for the encoder are as follows:

$$S_0 = X_1 + X_3 + X_5 + X_7$$
$$S_1 = X_2 + X_3 + X_6 + X_7$$
$$S_2 = X_4 + X_5 + X_6 + X_7$$

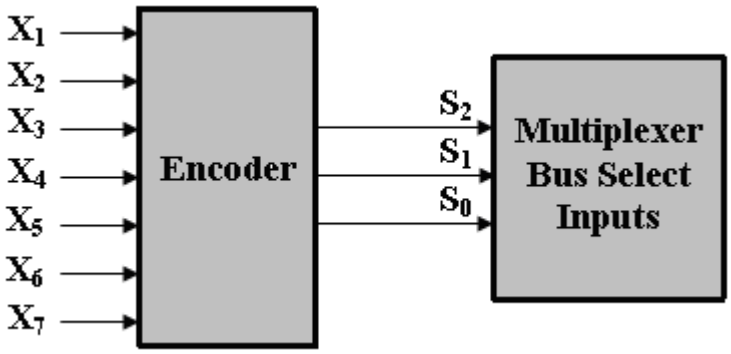


Figure 6.17

How to determine the control logic for each encoder input

Suppose that it is required to find the control logic that makes $X_2 = 1$ and $S_2S_1S_0 = 010$ (select the register **PC** as the source register).
We proceed as follows:

1. Scan all register transfer statements in table 6.6 and extract those statements that have **PC** as a source register.

$\overline{R}T_0 \;:\; AR \leftarrow PC$

$RT_0 \;:\; TR \leftarrow PC$

$D_5T_4 : \; M[AR] \leftarrow PC$

2. The Boolean function for $\mathbf{X_2}$ is:

$$X_2 = \overline{R}T_0 + RT_0 + D_5T_4$$

3. Draw the control logic for $\mathbf{X_2}$ **(**Figure 6.18).

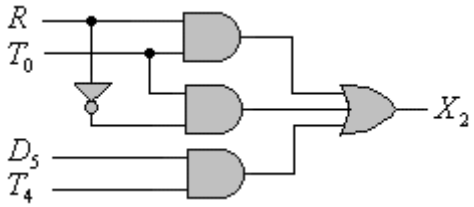


Figure 6.18

In a similar manner, one can determine the gate logic for other registers or memory.

Homework

Find the control logic that makes $\mathbf{X_1}$ = 1 and $\mathbf{S_2S_1S_0}$ = **001** (select the register **AR** as the source register).

6.9.2. Design of Accumulator Logic

Figure 6.19 presents the circuit block diagram associated with the accumulator register (**AC**).

The adder and logic circuit has three inputs:

1. **16-bit** inputs from the outputs of the accumulator register (**AC**) it self.
2. **16-bit** inputs from the data register **DR**.
3. **8-bit** inputs from the input register **INPR**.

The output of the adder and logic circuit provides the data inputs for the accumulator register (**AC**).

In addition, it is necessary to include in the design the logic gates for controlling the **LD**, **INR**, and **CLR** of the register and the controlling operation of the adder and logic circuit.

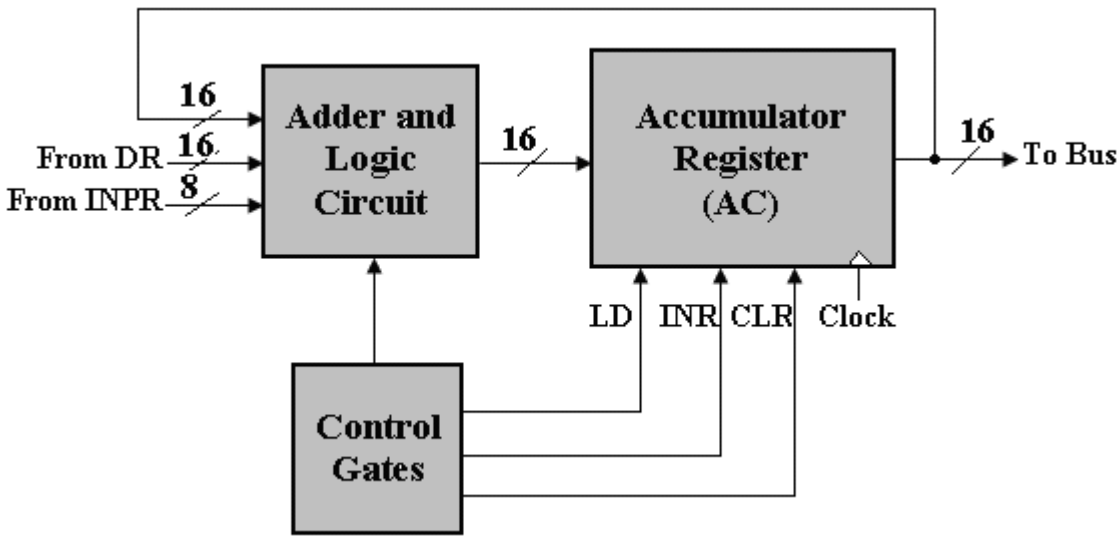


Figure 6.19

6.9.2.1. Control of Accumulator Register

In the same manner in order to design the logic that controls the accumulator register **AC**, it is necessary to scan all register transfer statements in table 6.6 and extract those statements that change the content of **AC**. These register transfer statements are:

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

From the control functions in the list above, the gate configuration is derived as follows:

1. The control function for the increment microoperation **(INR)** is rB_5, where $r = D_7 \bar{I} T_3$ and $B_5 = IR(5)$.

2. The control function for the clear microoperation **(CLR)** is rB_{11}, where $r = D_7 \bar{I} T_3$ and $B_{11} = IR(11)$.

3. The control function for the load microoperation (**LD)** is the result of the remaining seven microoperations, which generated in the adder and logic circuit and are loaded at the proper time.

Figure 6.20 shows the gate structure that controls the **LD**, **INR**, and **CLR** inputs of the accumulator register **AC**. **Note that the outputs of the gates for each control function are marked with a symbolic name.**

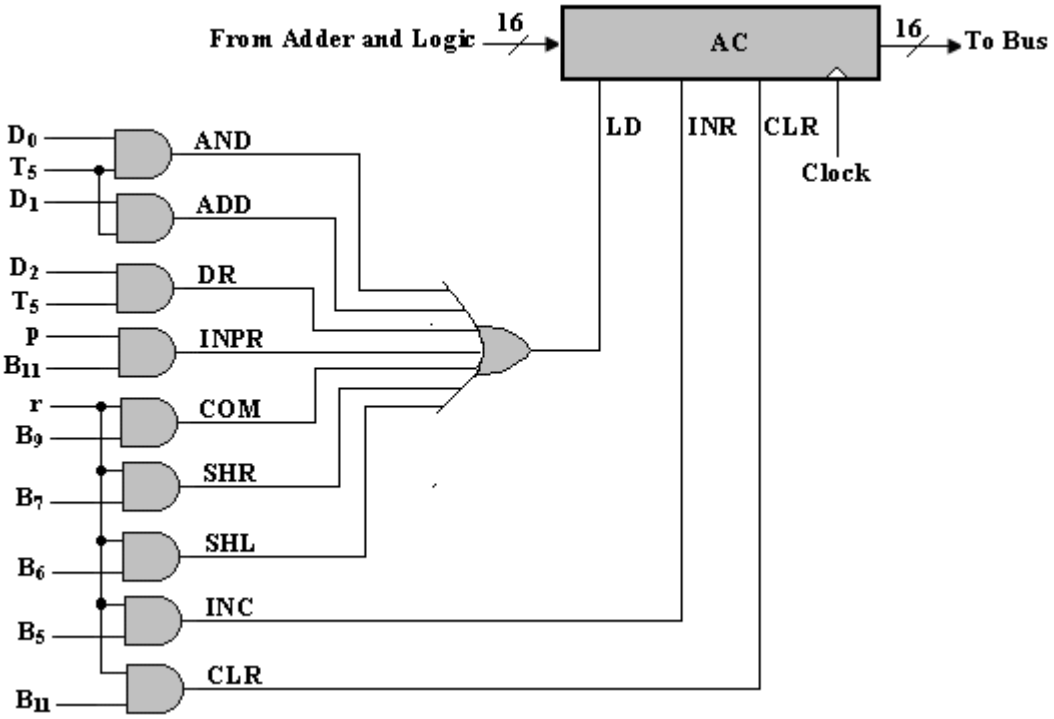


Figure 6.20

6.9.2.2. Adder and Logic Circuit

Figure 6.21 shows the internal construction of the accumulator register **AC**. Each stage has a **JK flip-flop**, two **OR** gates, and two **AND** gates. The load (**LD**) is connected to the inputs of the **AND** gates.

Figure 6.23 shows one stage of **AC** register (here the **OR** gates are removed) since the other functions (i.e. Clear and Increment of the register **AC** are not included). When **LD** input is enabled, the **16** inputs $\mathbf{I_i}$ for **i = 0,1,2, …, 14,15** are transferred to **AC (0-15).**

Every stage of Adder and Logic Circuit consists of:

1. Seven **AND** gates, every one has one of its inputs one of the functions comes from figure 6.20 as an enable. For example, the input named **DR** in figure 6.22 connected to the output marked **DR** in figure 6.20.
2. One **OR** gate.
3. One Full-adder (**FA**).

How the arithmetic and logic operations are achieved?

1. ADD operation

This operation achieved using a binary full-adder having their two inputs from the corresponding bits (**i**) of registers **AC** and **DR**, the input carry from the previous stage, and the result transfers through **OR** gate to register **AC**, and the output carry to next stage or to the **E** flip-flop when this stage is the last stage.

2. AND operation

The **AND** operation is obtained through **ANDing AC** (**i**) with the corresponding bit in the data register **DR (i)**, and the result transfers through **OR** gate to register **AC**.

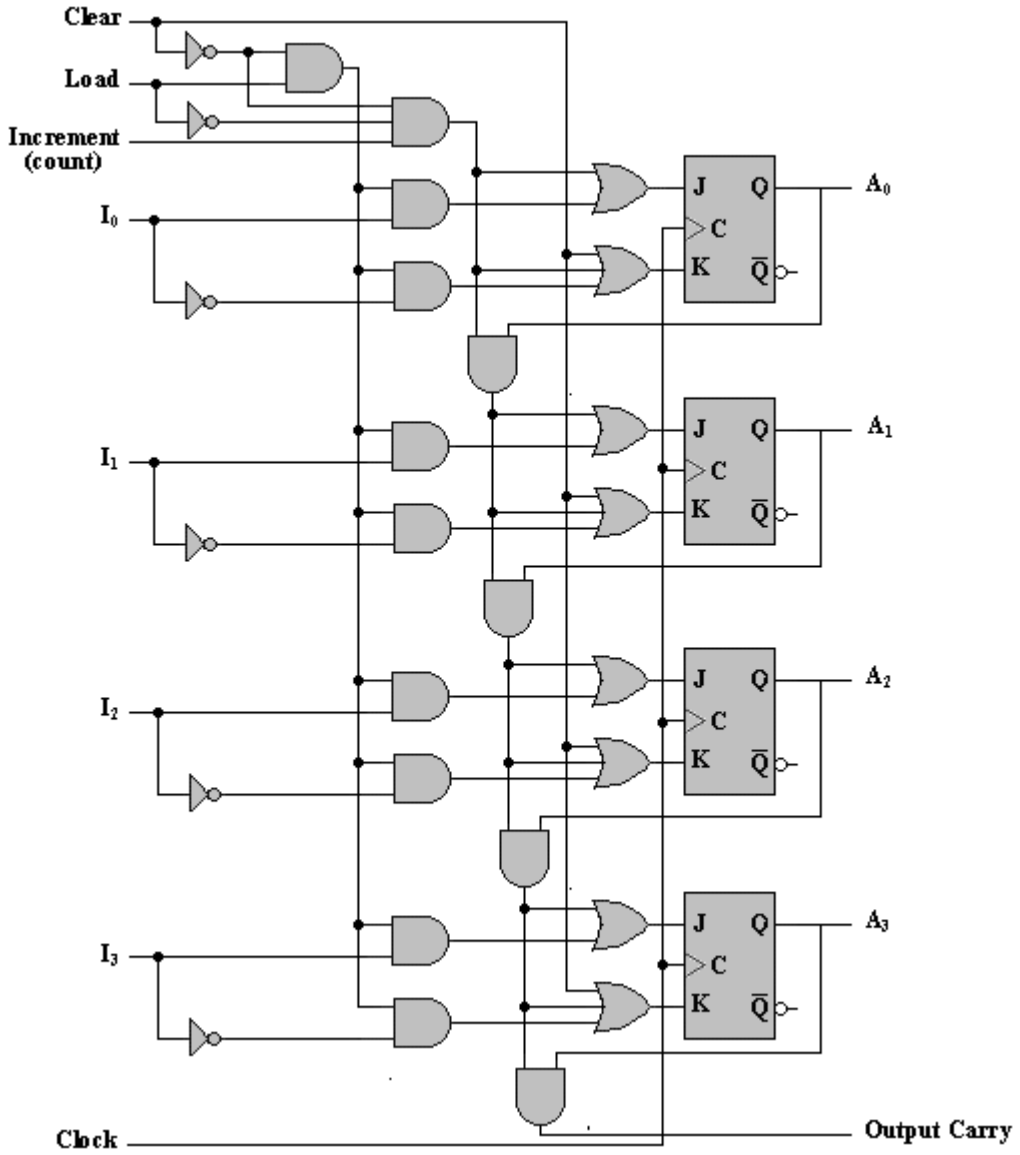


Figure 6.21

3. Transfer the content of the register DR

When DR input is active, the content of the register DR transferred to the register AC.

4. Transfer the content of the register INPR

When **INPR** input is active, the 8-bits of the register **INPR** transferred to **Bits 0-7 of the accumulator register AC.**

5. Complementation of the contents of AC register

When **COM** input is active, the content of register **AC** is inverted and re-transferred to the register**.**

6. **Shift-right operation**

This operation transfers bit AC (i+1) into bit AC (i).

7. **Shift-left operation**

This operation transfers bit AC (i -1) into bit AC (i).

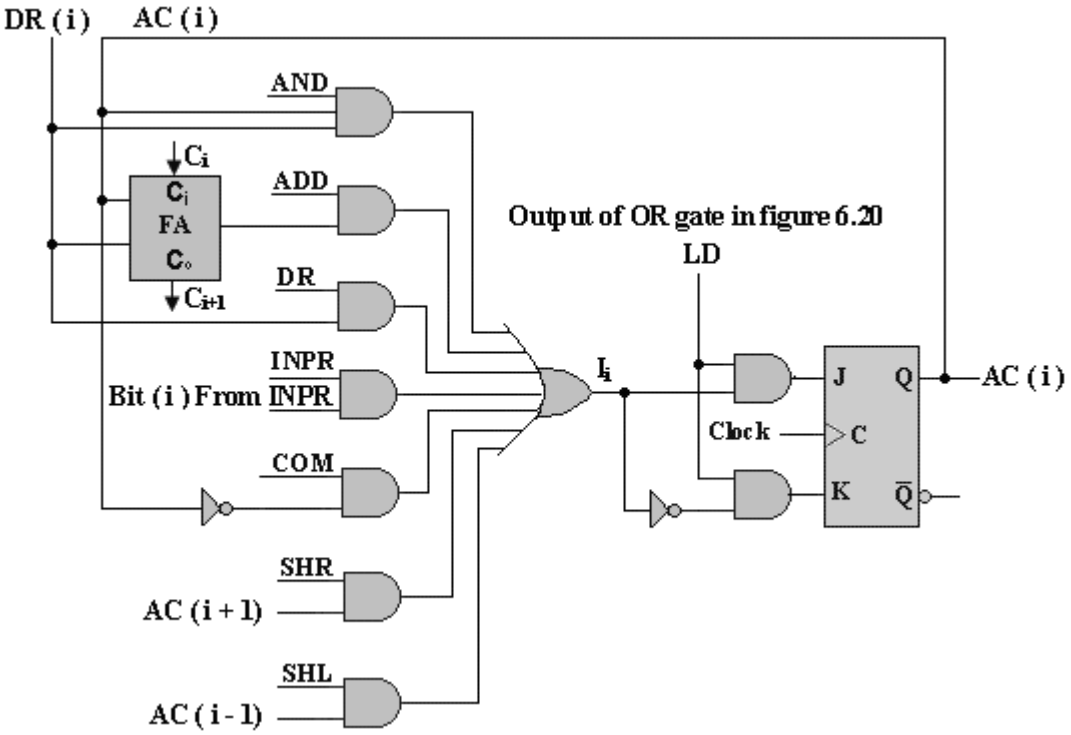


Figure 6.22