UNIT- II Syllabus:

Register Transfer Language and Microoperations: Register Transfer language. Register Transfer, Bus and Memory Transfers, Arithmetic Micro operations, Logic Micro Operations, Shift Micro Operations, Arithmetic Logic Shift Unit.
Basic Computer Organization and Design: Instruction Codes, Computer Register, Computer Instructions, Instruction Cycle, Memory – Reference Instructions. Input–Output and Interrupt, Complete Computer Description.

Register Transfer Language and Microoperations

Register Transfer language:

- A digital system is an interconnection of digital hardware modules.
- These 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.
- 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 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** (using symbols) to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers.

- The symbolic notation used to describe the microoperation transfers among registers is called a **"Register Transfer Language (RTL)"**.
- 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 **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.

Register Transfer:

→ A **register** is a quickly accessible small memory area by the CPU. It holds the data or an instruction or any other information temporarily.

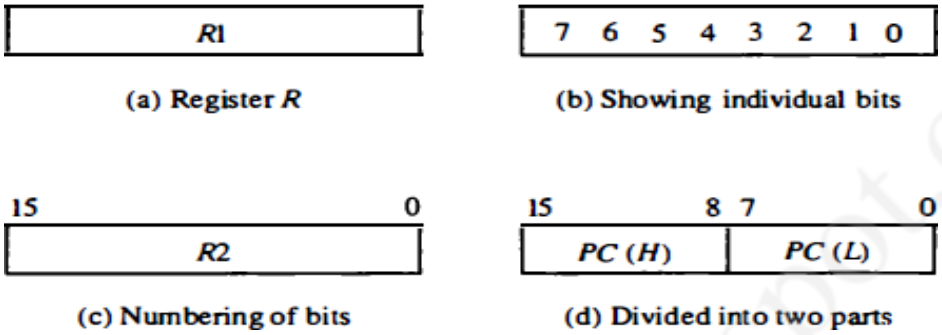
Computer registers are designated by **capital letters (sometimes followed by numerals)** to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR.

Some of other registers are PC (program counter), IR (instruction register, and R1 (it is a 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.

A register is represented by a rectangular box with the name of the register inside, as in Figure (a). The individual bits of register can be distinguished as in Figure (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in Figure (c). A 16-bit register is partitioned into two parts into two parts as in Figure (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.

Computer registers can be represented in various ways, some of those are shown here:

Block diagram of register.



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

R2 ← R1

This statement denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R l . By definition, the content of the source register R1 does not change after the transfer.

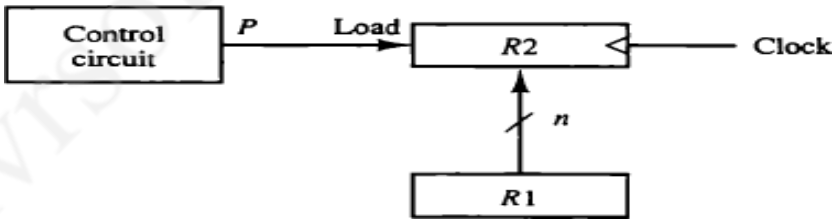
✓ Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

If **(P = 1)** *then* **(R2 ← R1)**

Where P is a control signal generated in the control section.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. The below figure shows the block diagram that depicts the transfer from R1 to R2.

Transfer from R1 to R2 when P = 1.



As shown 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.

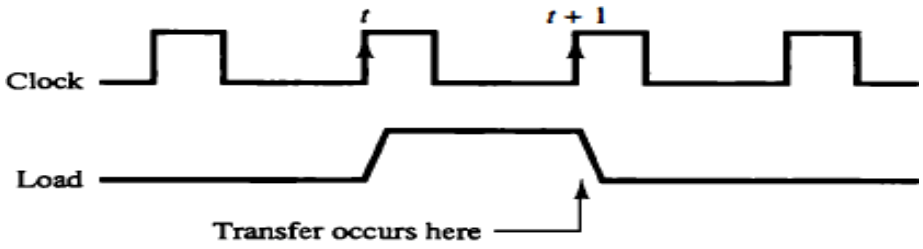


Figure: Timing diagram

The basic symbols of the register transfer notation are listed in the below shown table.

TABLE : 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*

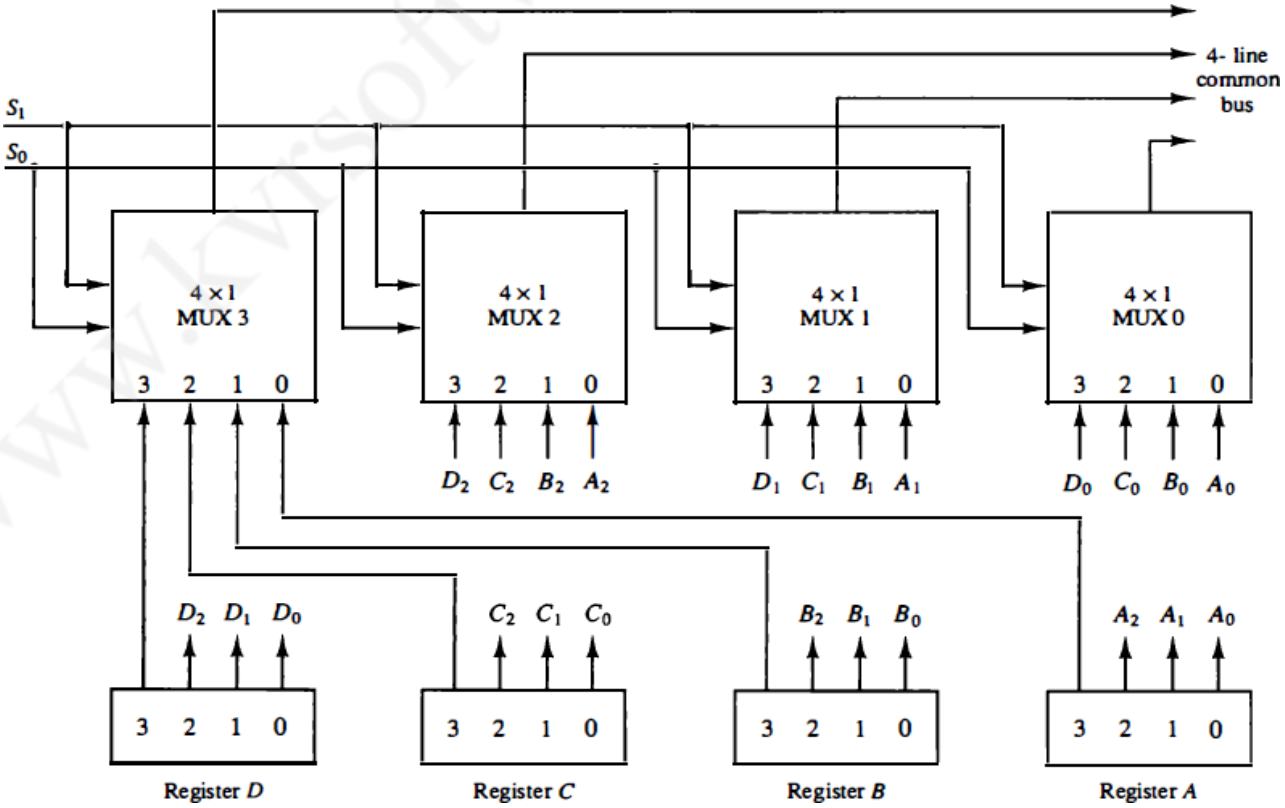
- ➔ Registers are denoted by capital letters, and numerals may follow the letters.
- ➔ Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register.
- ➔ The arrow denotes a transfer of information and the direction of transfer.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

Bus and Memory Transfers:

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

One way of constructing a common bus system is with multiplexers is shown in the below figure(a).

[A multiplexer takes the **input** on any one of 'n' **input** lines and giving this **input** to one **output** line. In our example 4X1 multiplexer is used, n=4 (inputs) and 1 output.]



Figure(a): Bus system for four registers

- The multiplexers select the source register whose binary information is then placed on the bus.
- Each register has four bits, numbered 0 through 3. The bus consists of **four 4 x 1** multiplexers each having four data inputs, 0 through 3, and two selection inputs, **S_1 and S_0.**

To avoid the complexity in 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.

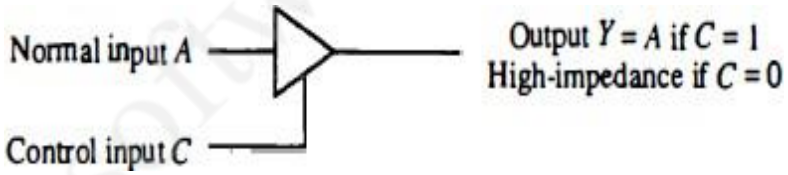
For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labelled A1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of all the registers, MUX 1 multiplexes the four 1 bits of all the registers, and similarly for the other two bits.

- ❑ 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_1 S_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.

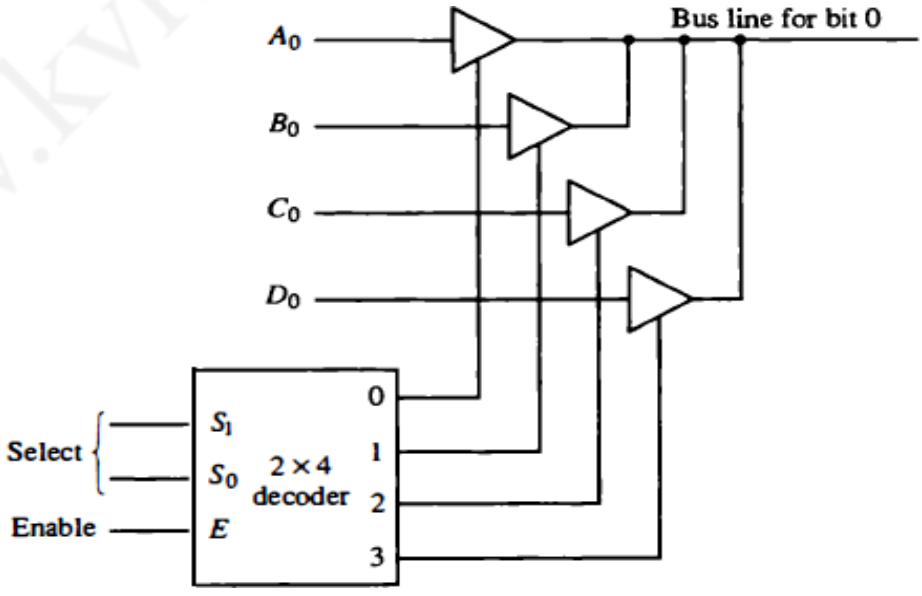
S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

Three-State Bus Buffers:

- 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 graphic symbol of a **three-state buffer gate** is shown in the below Figure.



The construction of a bus system with three-state buffers is shown in the below figure(b).



Figure(b): Bus line with three state-buffers.

- The outputs of four buffers are connected together to form a single bus line. 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.

To construct a common bus for four registers of n bits each using three- state buffers, we need n circuits with four buffers in each as shown in the below figure. 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 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:

Read: DR ← M [AR]

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

The **write** operation can be stated as follows:

Write : M [AR] ← R1

→ This causes a transfer of information from R1 into the memory word M selected by the address in AR.

Arithmetic Microoperations:

A **microoperation** is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations
2. Arithmetic microoperations
3. Logic microoperations
4. Shift microoperations

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

R3 ← R1+R2

It specifies an **add microoperation**. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3.

Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

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

$\overline{R2}$ is the 1's complement of R2, Adding 1 to the 1's Complement produces the 2's Complement form. Adding the R1 to the 2's complement of R2 is equivalent to R1-R2.

All the arithmetic microoperations are shown in the below Table (a).

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

Table(a): Arithmetic Microoperations

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**. The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a **binary adder**.

The below figure (c) shows the interconnections of four full-adders (FA) to provide **a 4-bit binary adder.**

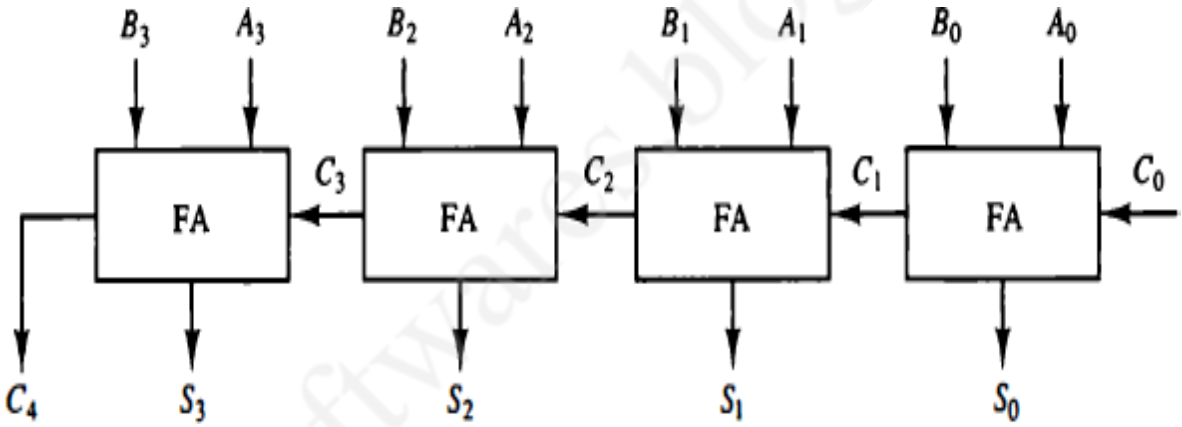


Figure (c): a 4-bit binary adder

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.

➔ An n-bit binary adder requires n full-adders.

Binary Adder-Subtractor :

The subtraction of binary numbers can be done most conveniently by means of complements. A - B can be done by taking the 2's complement of B and added with A.

- 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 the below figure (d).
- 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 ⊕ 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 ⊕ 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 2's complement of B.

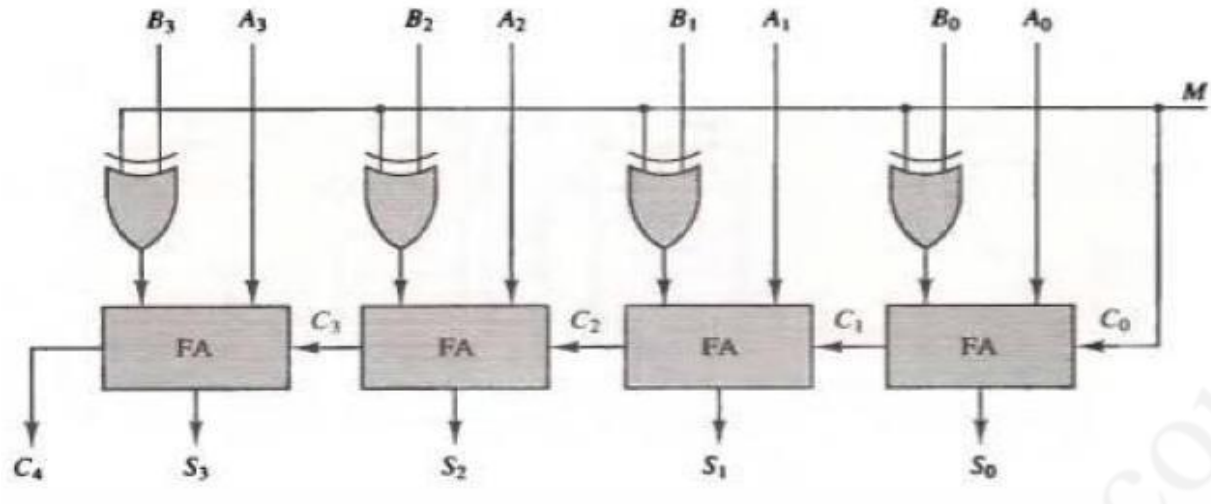


Figure (d): 4-bit adder-subtractor

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 can be easily implemented with half-adders connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in the below figure(e).

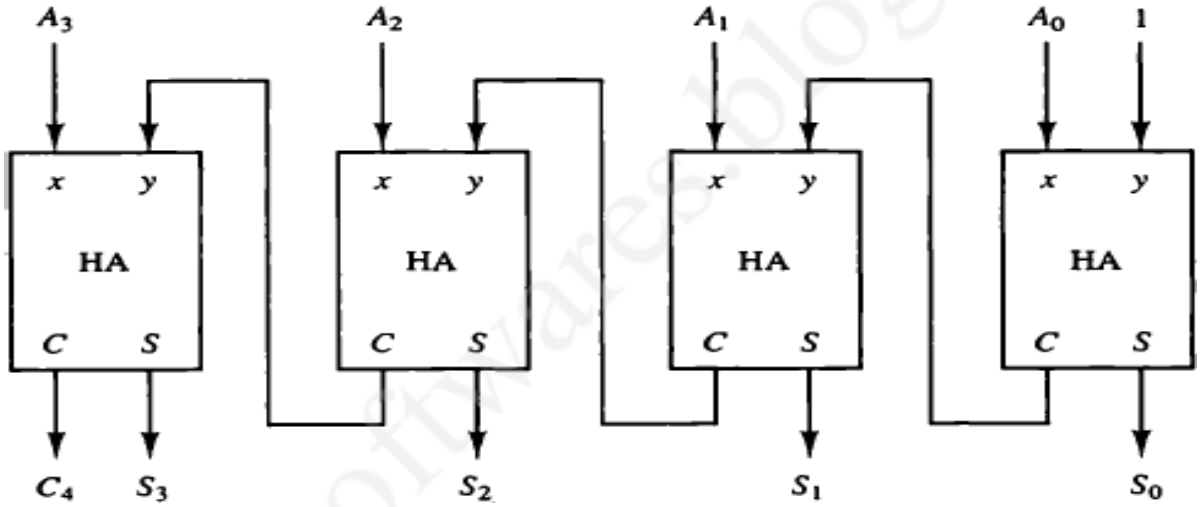


Figure (e): A 4-bit binary Incrementer

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 A0 through A3, adds one to it, and generates the incremented output in S0 through S3.

- ✓ The above 4-bit incrementer circuit can be extended to an **n-bit binary incrementer** by including n half-adders.

Arithmetic Circuit

The arithmetic microoperations listed in Table (a) can be implemented by using one 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 the below figure (f).

- ➔ 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

➔ 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.

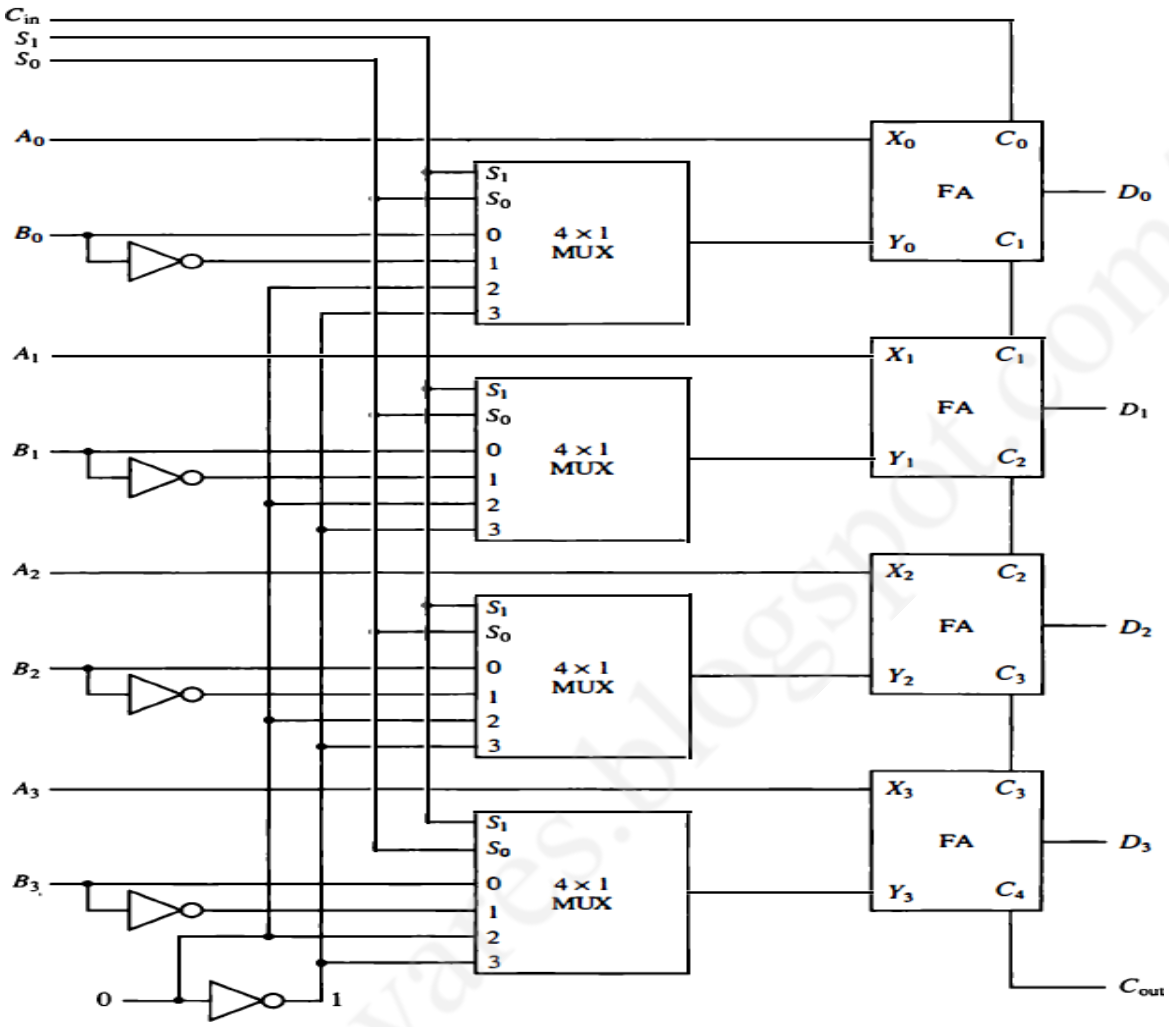


Figure (f): 4-bit Arithmetic Circuit

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 (b).

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

Table (b): Arithmetic Circuit Function Table

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

Subtraction: When S_1S_0 = 01, the complement of B is applied to the Y inputs of the adder. If Cin = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A - B. When Cin = 0, then D = A + B . This is equivalent to a subtract with borrow, that is, A - B – 1.

Increment: 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 + Cin· This gives D = A when Cin = 0 and D = A + 1 when Cin = 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.

Decrement: 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 Cin = 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 Cin = 1, then D = A - 1 + 1 = A, which causes a direct transfer from input A to output D.

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: R1 ← R1⊕R2

- 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:

```
1010    Content of R1
1100    Content of R2
0110    Content of R1 after P = 1
```

Special symbols will be used for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions.

i. The symbol V will be used to denote an OR microoperation.

ii. The symbol Λ to denote an AND microoperation.

iii. 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.

For example, in the statement P + Q: R1← R2 + R3, R4 ← R5 V 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 V between registers R5 and R6.

List of Logic Microoperations:

There are 16 different logic microoperations 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 the below Table(c).

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

Table (c): Truth Tables for 16 functions of two variables

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 (d) 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.

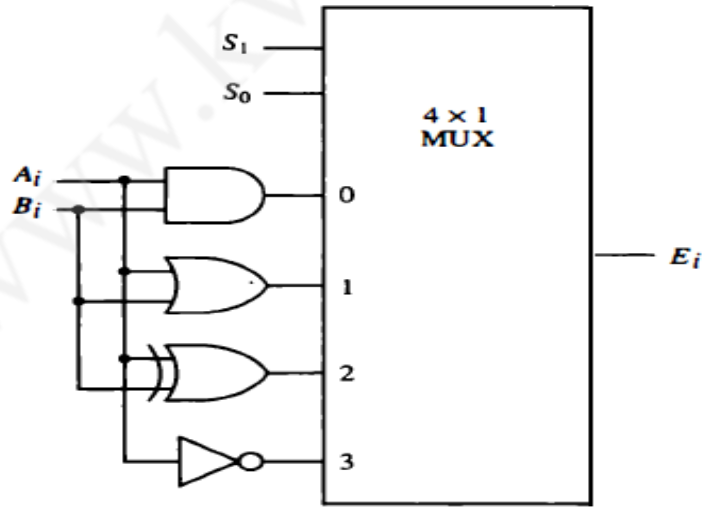
Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	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

Table (d): Sixteen Logic Microoperations

Hardware Implementation:

The hardware implementation of logic rnicrooperations 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 rnicrooperations, most computers use only four-AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.

The below figure(g) shows one stage of a circuit that generates the four basic logic rnicrooperations . 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.



(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

Figure (g): One stage of logic circuit

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.

1. **selective-set (or) the OR microoperation**

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.

EX:

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

2. **selective-complement (or) the exclusive-OR microoperation**

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 .

EX:

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

3. **selective-clear (or) the AB' microoperation**

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

EX:

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

4. **Mask (or) AND microoperation**

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.

EX:

```
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.

5. **Insert (or) the OR microoperation**

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.

EX:

```
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
```

6. **clear**

The clear operation compares the words in A and B and produces an all 0' s result if the two numbers are equal.

EX:

```
1010    A
1010    B
0000    A ← A ⊕ 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.

Shift Microoperations:

Shift rnicrooperations are used for serial transfer of data. 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:

1. Logical shift
2. Circular shift
3. Arithmetic shift

1. 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 rnicrooperations. For example:

R1← shl R1
R2← shr R2

- First statement specifies a 1-bit shift to the left of the content of register R1.
- Second statement specifies 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.

2. 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 rnicrooperations is shown in the below table.

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

3. Arithmetic shift:

An arithmetic shift is a rnicrooperation that shifts a signed binary number to the left or right.

- An arithmetic shift-left multiplies a signed binary number by 2 and 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 when it is multiplied or divided by 2.
- ✓ The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right.
- ✓ 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.

Figure (h): Arithmetic shift right

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, can be used to detect an arithmetic shift-left overflow.

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

If V, = 0, there is no overflow,
if V, = I, there is an overflow and a sign reversal after the shift.

Hardware Implementation

A combinational circuit shifter can be constructed with multiplexers as shown in Figure (i). 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 (I_L) and the other for shift right (I_R).

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 given under the Figure (i) 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.

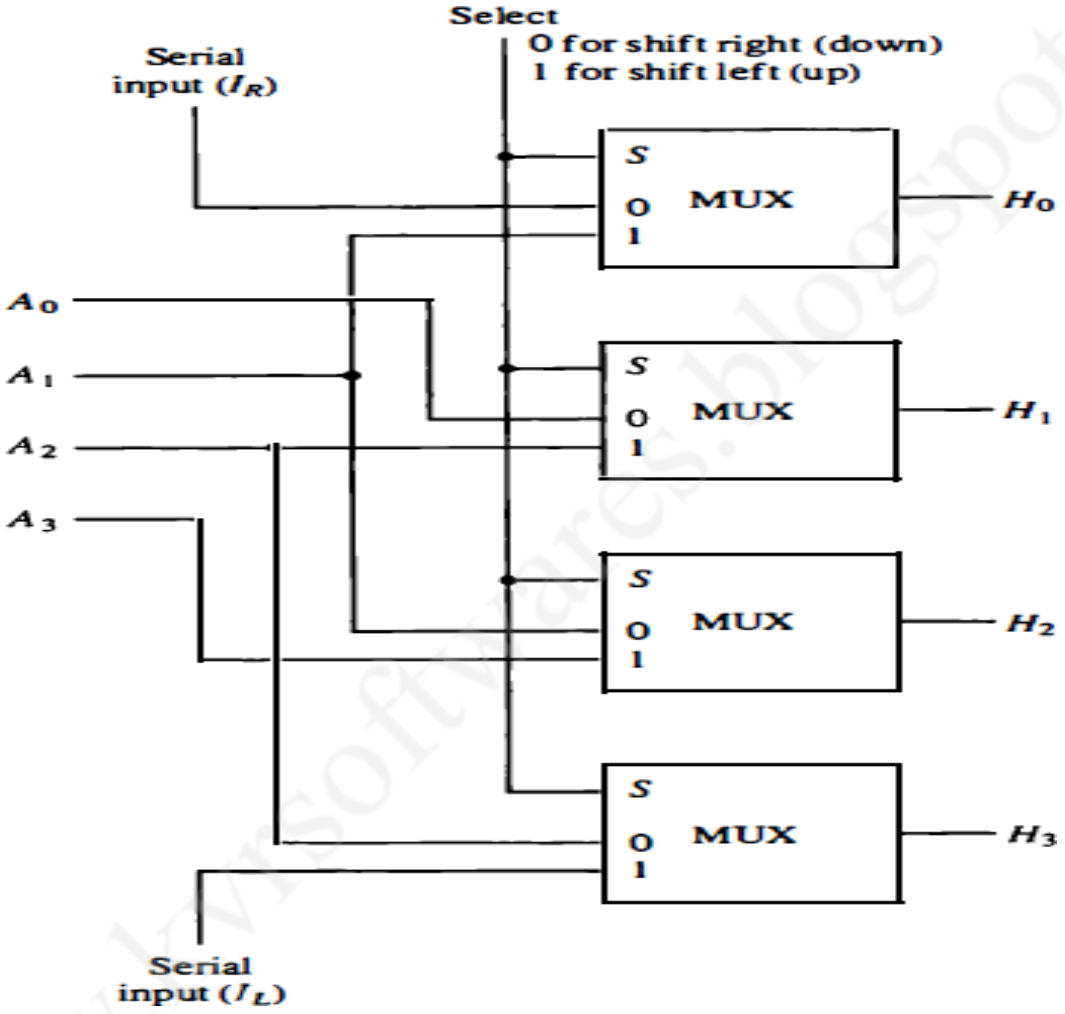


Figure (i): 4-bit combinational Circuit shifter.

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

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.

- 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 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 can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in the below Figure (j).

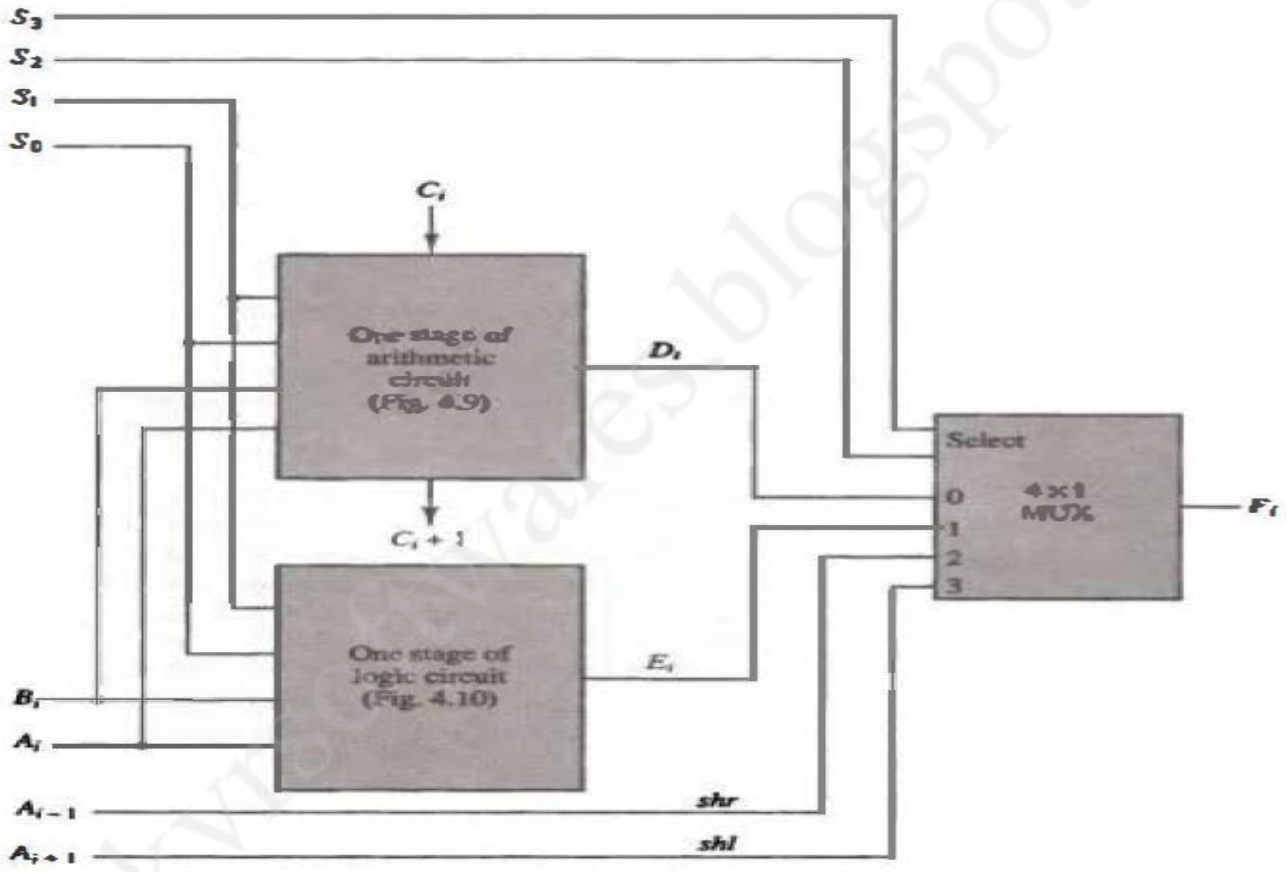


Figure (j): One stage of arithmetic logic shift unit

The above shown one stage ALSU provides 8 arithmetic operations, 4 logic operations, and 2 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 (e) lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with $S_3S_2 = 00$. The next four are logic operations 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 x ' 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.

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	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

Table (e): Function table for arithmetic logic shift unit

Basic Computer Organization and Design

Instruction Codes:

The general purpose digital computer is capable of executing various micro-operations and also can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by using a program.

- A **program** is 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. 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.
- 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 part. The **operation code** of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- 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.

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.

The below figure shows this type of organization.

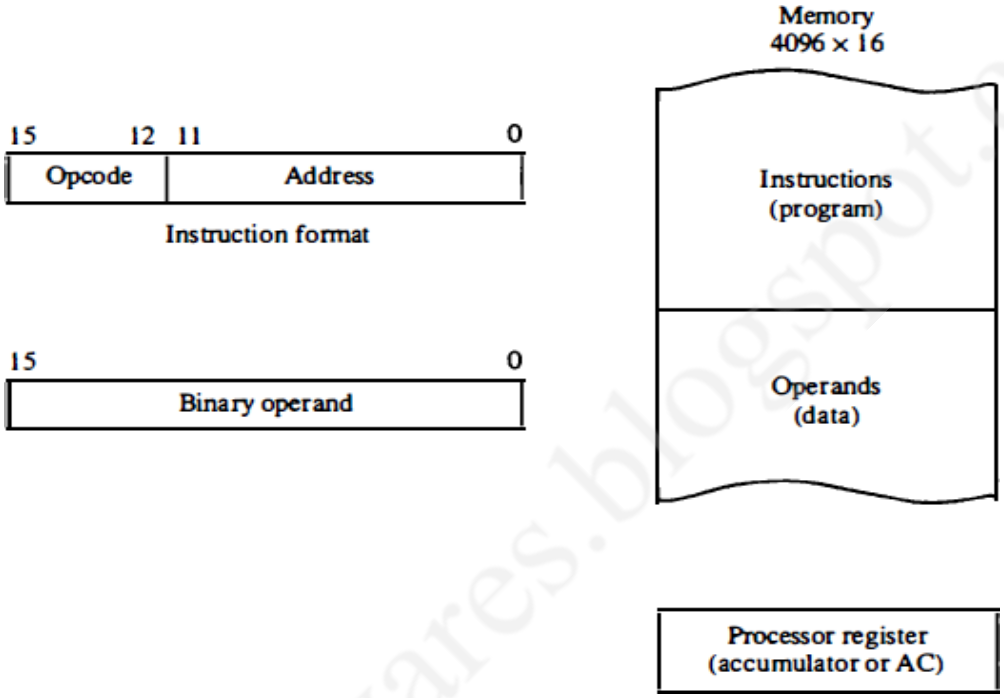


Figure (k): Stored program organization

Instructions are stored in one section of memory and data in another. **EX:** 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 (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.

- 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.

Indirect Address

- 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**.

- When 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, the instruction is said to **an indirect address**. One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- An **effective address** is the address of the operand.

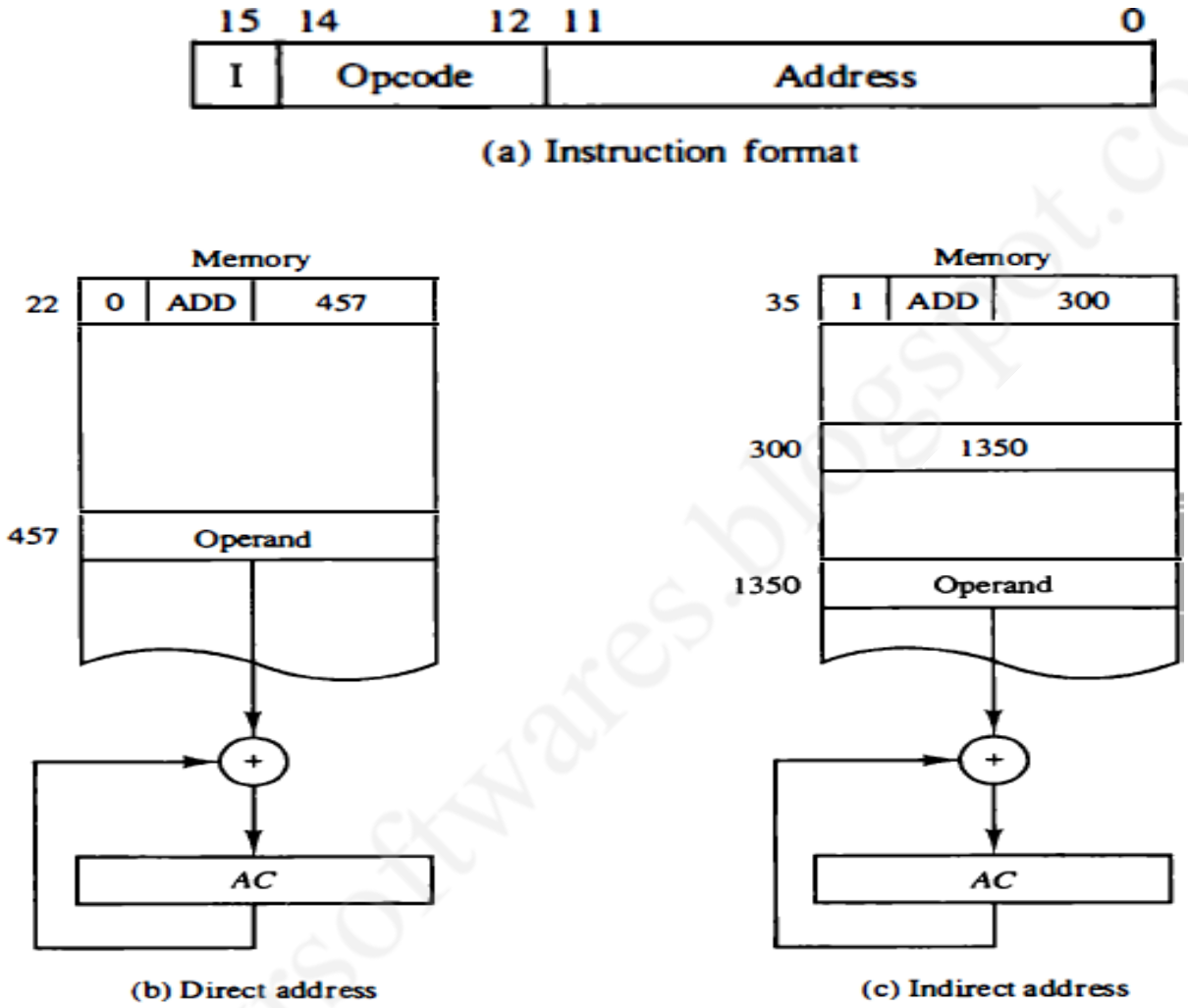


Figure (l): Demonstration of direct and indirect address.

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.

The registers available in the computer are shown in the below figure (m) and table (f), a brief description of their function and the number of bits that they contain also given.

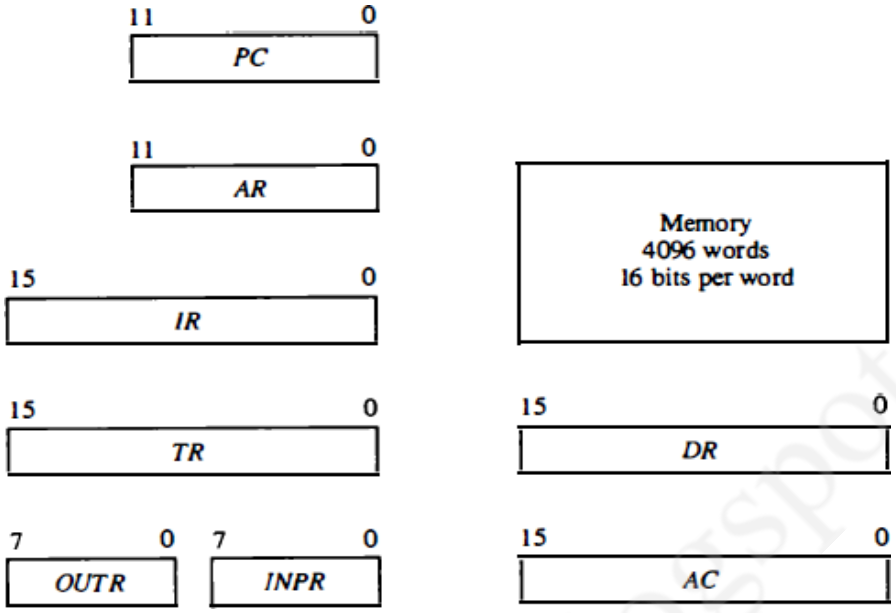


Figure (m): Basic computer registers and memory.

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

Table (f): List of Registers for the Basic computer.

Common Bus System:

- The basic computer 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.
- 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**.

The connection of the registers and memory of the basic computer to a common bus system is shown in the below figure (n).

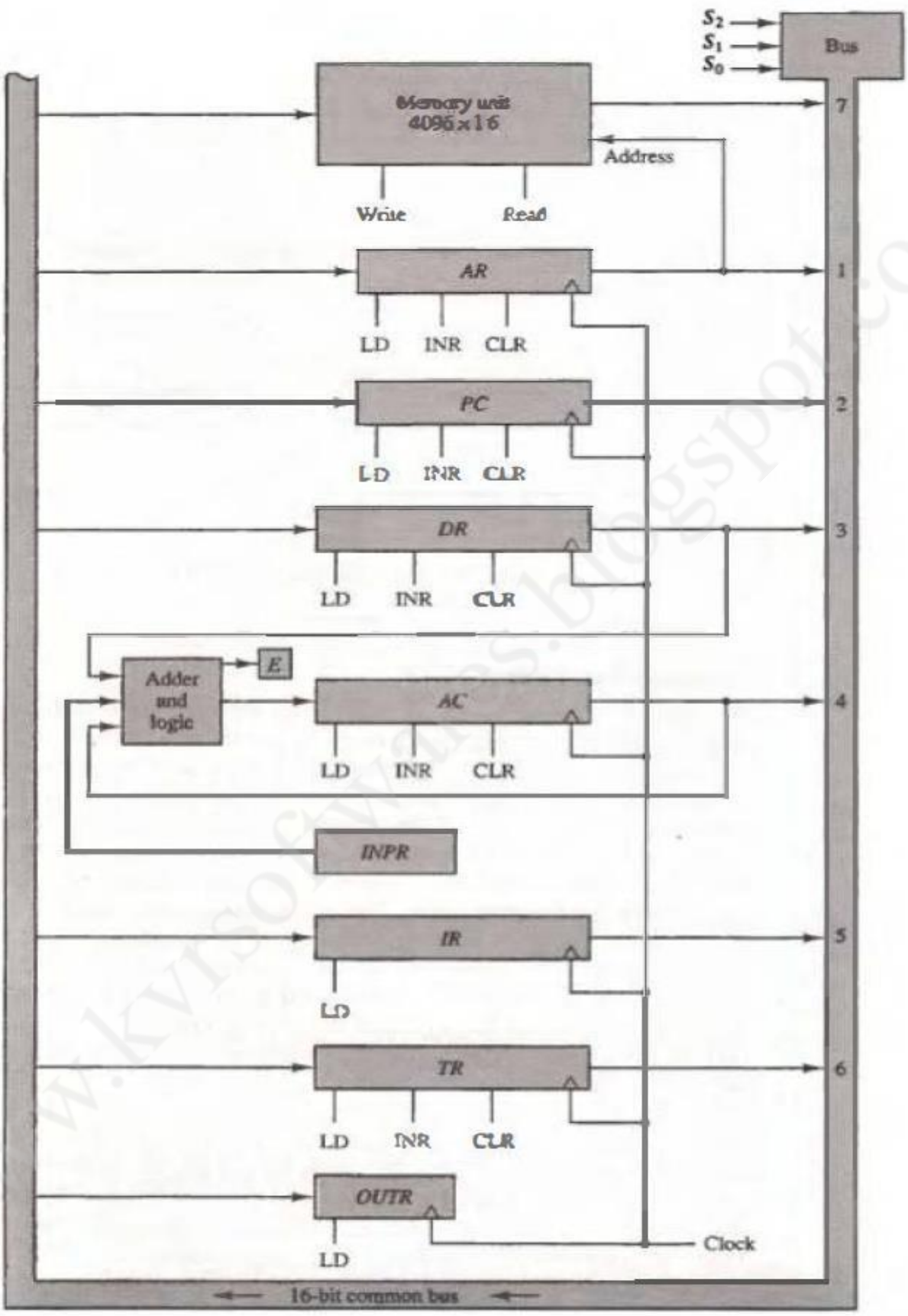


Figure (n): Basic computer registers connected to a common bus.

- 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.

For example1, 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.

For example2. The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

- 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 rnicrooperations

DR ← AC and AC ←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, 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.

Computer Instructions:

The basic computer has **three types of instruction code formats**,

1. Memory-reference instruction.
2. Register-reference instruction.
3. An input-output instruction.

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.

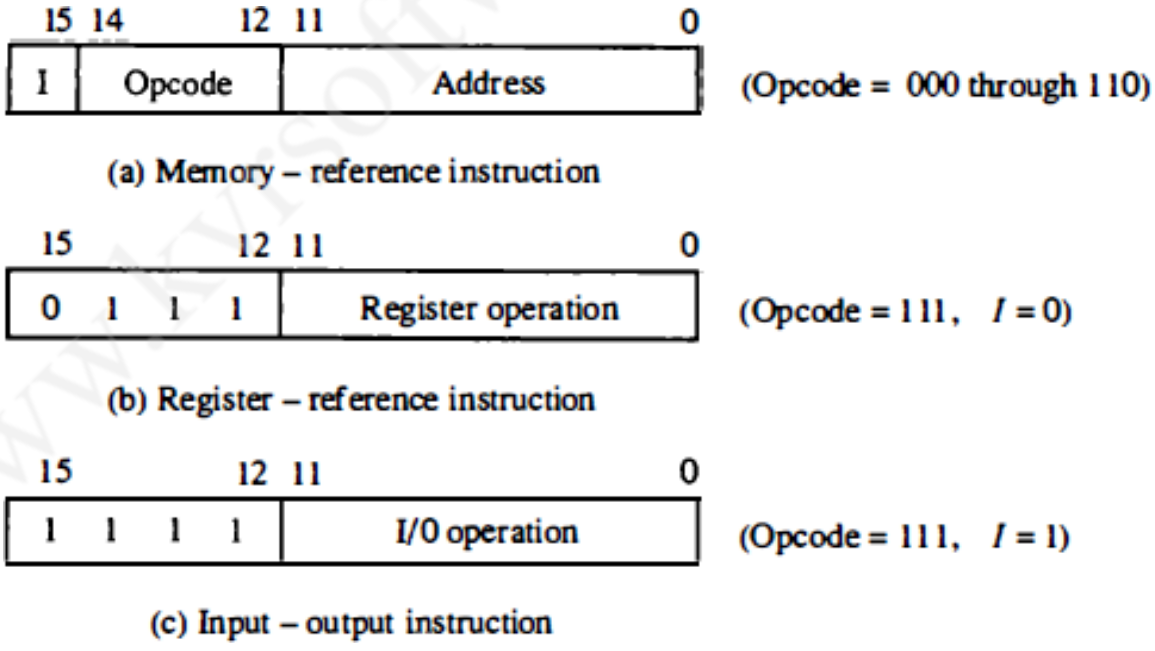


Figure (n): Basic computer instruction formats

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 to 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. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I = 0 for direct address and I = 1 for indirect address.
- If the 3-bit opcode = 111, control then inspects the bit in position 15. If this bit = 0, the instruction is a **register-reference type**. These instructions use 16 bits to specify an operation.
- If the bit I = 1, the instruction is **an input-output type**. These instructions also use all 16 bits to specify an operation.

The instructions for the computer are listed in Table (g, h, i).

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	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

Table (g): Memory-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 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

Table (h): Register-reference instructions

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

Table (i): Input-output instructions

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.

i. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 (000) to 6 (110) since the last bit is 0.
ii. When I = I, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 (1000) to E (1110) since the last bit is I.

B) 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.

C) 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

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.

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 rnicrooperations for the fetch and decode phases can be specified by the following register transfer statements.

T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← 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 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.

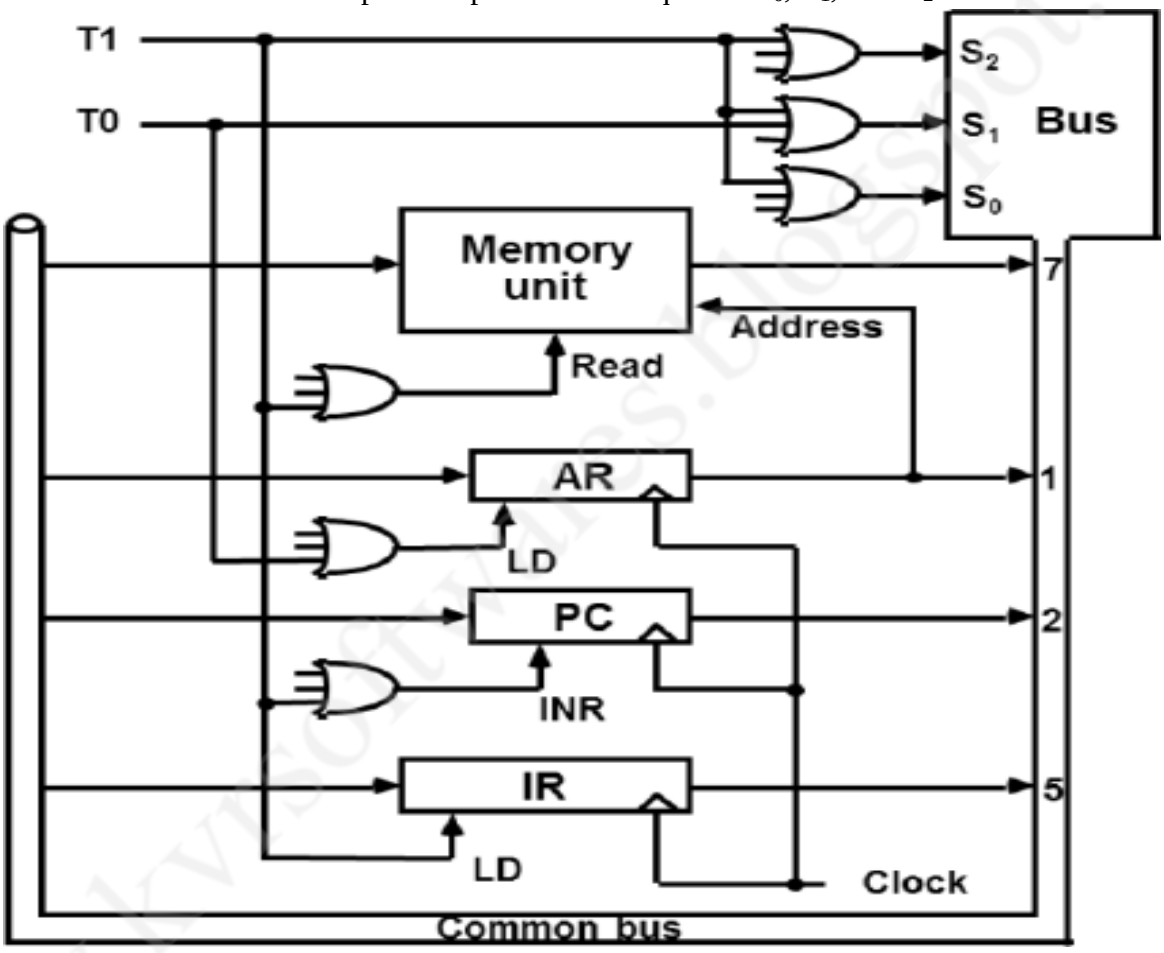


Figure (o): Register transfers for the fetch phase

The above Figure (o) 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_2 S_1 S_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

T1: IR←M[AR], PC ← 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 $S_2 S_1 S_0 = 111$.
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.

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.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111.

- If $D_7 = 1$, the instruction must be a register-reference or input-0utput type.
- If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.

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:

$D_7' I T_3$: $AR \leftarrow M[AR]$
$D_7' I' T_3$: Nothing
$D_7 I' T_3$: Execute a register-reference instruction
$D_7 I T_3$: 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 T4. 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 $T_0 = 1$.

The flowchart of Figure (p) presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding

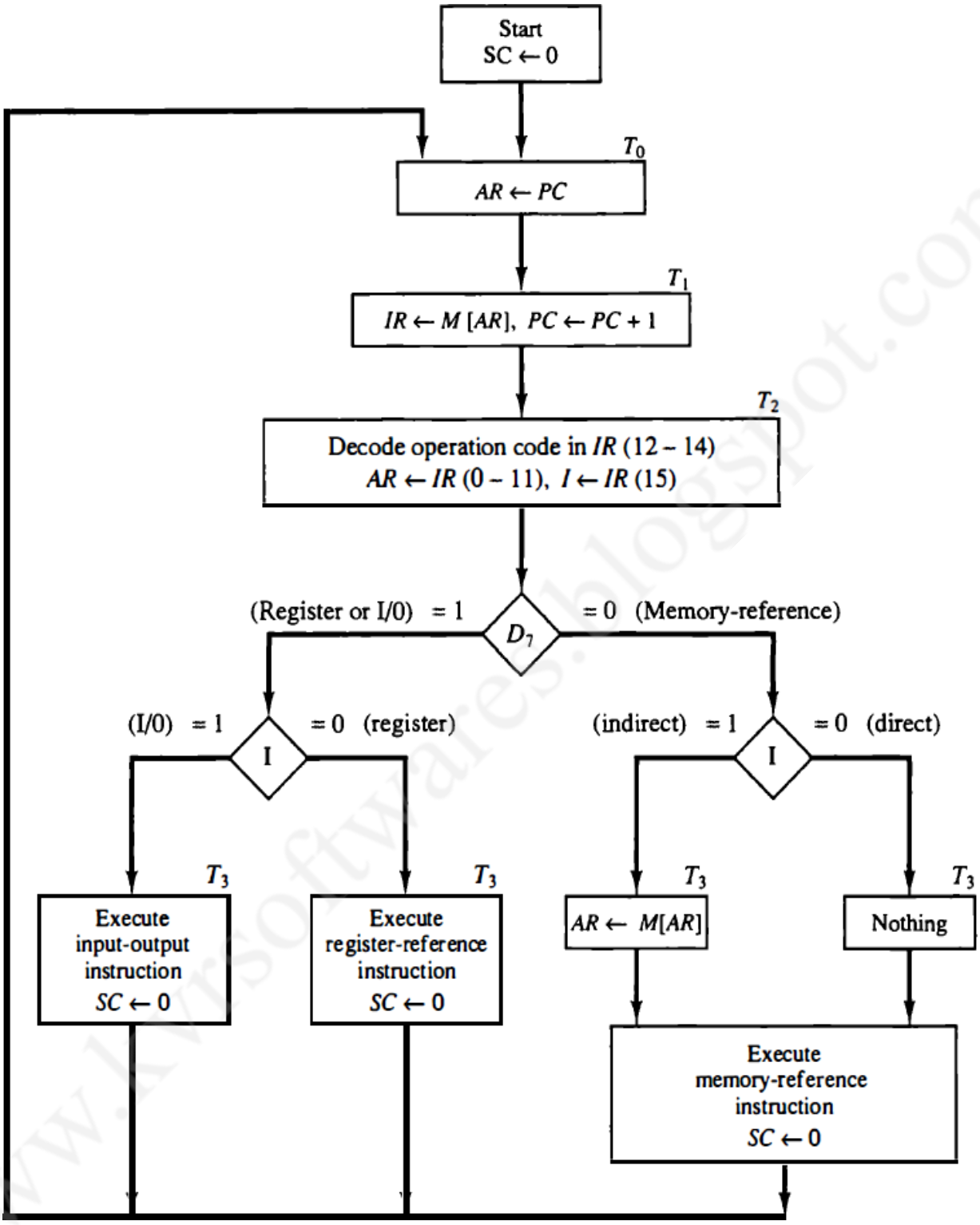


Figure (p): Flowchart for instruction cycle (initial configuration).

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 .

- Each control function needs the Boolean relation D_7 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 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, which gives the binary equivalent 0111 1000 0000 0000.
  - i. The first bit is a zero and is equivalent to I'.
  - ii. The next three bits constitute the operation code and are recognized from decoder output D7.
  - iii. Bit 11 in IR is 1 and is recognized from B_{11}.

The control function that initiates the rnicrooperation for this instruction is $D_7 I' T_3 B_{11} = r B_{11}$

$D_7 I' T_3 = r$ (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:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ 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

Table (j): Execution of Register-Reference Instructions

Memory-Reference Instructions:

- ✓ The below Table (k) 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 T2 when I = 0, or during timing signal T3 when I = 1. The execution of the memory-reference instructions starts with timing signal T4.

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 $PC \leftarrow PC + 1$

Table (k): Memory-Reference Instructions

AND : 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 rnicrooperations that execute this instruction are:

D_0T_4: DR ← M[AR]
D_0T_5: AC ← AC Λ DR, SC ← 0

ADD : 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,., is transferred to the E (extended accumulator) flip-flop. The rnicrooperations needed to execute this instruction are:

D_1T_4 : DR ← M[AR]
D_1T_5: AC ← AC + DR, E ← Cout, SC ← 0

LDA: Load to AC

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

D_2T_4: DR ← M[AR]
D_2T_5: AC ← DR, SC ← 0

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.
- 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 rnicrooperation:

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.

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

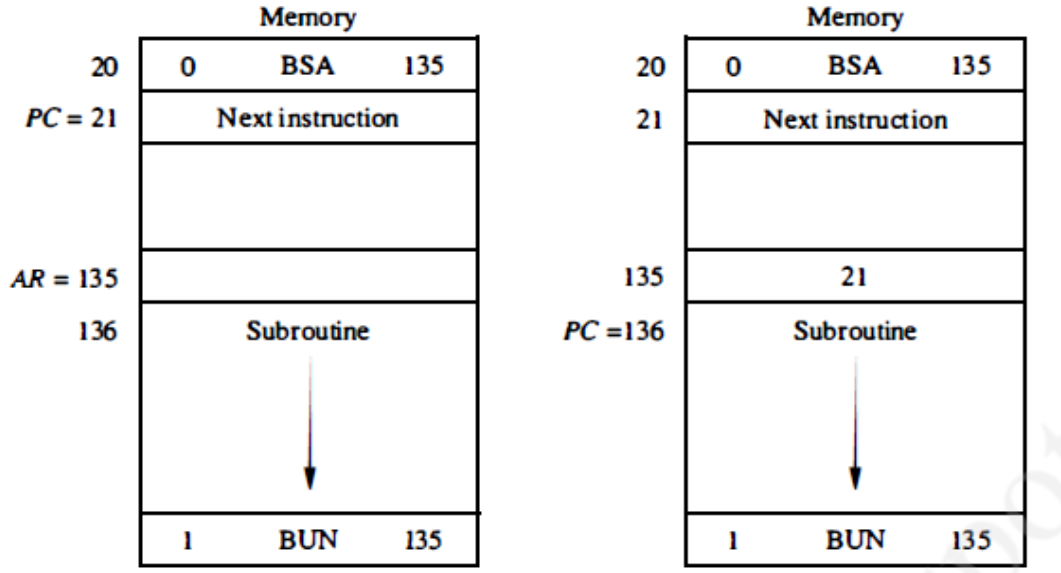
BSA: Branch and Save Return Address

EX:

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:

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

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.



(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 T4 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 T5 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.

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

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Figure (q). The control functions are indicated on top of each box. The microoperations that are performed during time T4, T5, or T6, depend on the operation code value. 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 T0 to start the next instruction cycle.

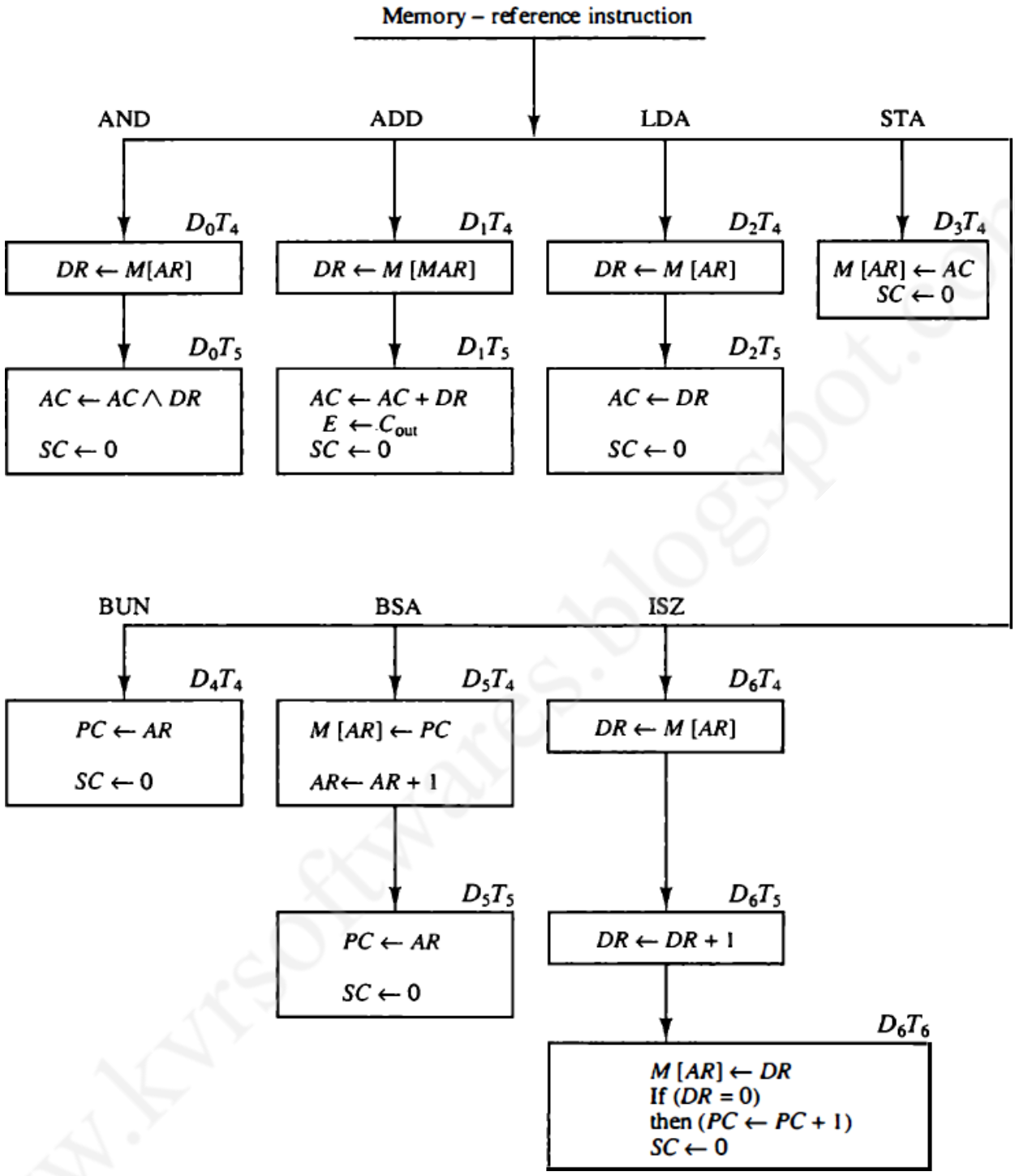


Figure (g): Flowchart for Memory-reference instructions

Input-Output and Interrupt:

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.

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.

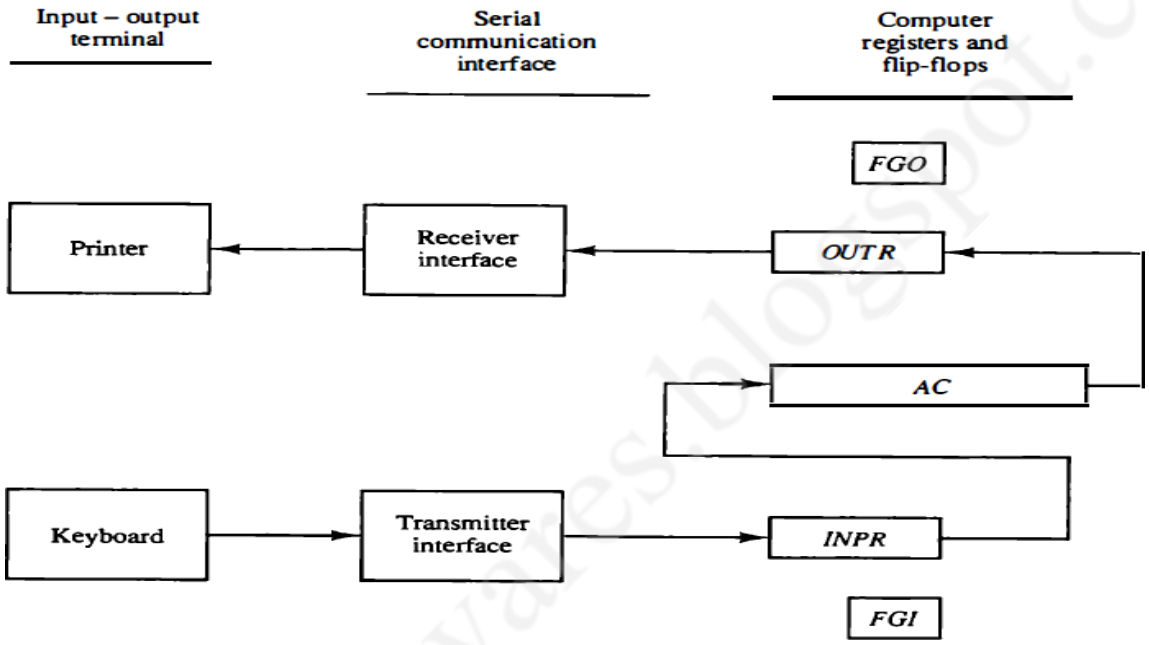


Figure (r): Input-Output configuration

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.

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 D7 = 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 (l).

$D_7IT_3 = p$ (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

Table (l): Input-Output instructions

Program Interrupt

The process of communication discussed so far 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 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 1μs. 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 lEN can be set and cleared with two instructions (IOF and ION instructions).

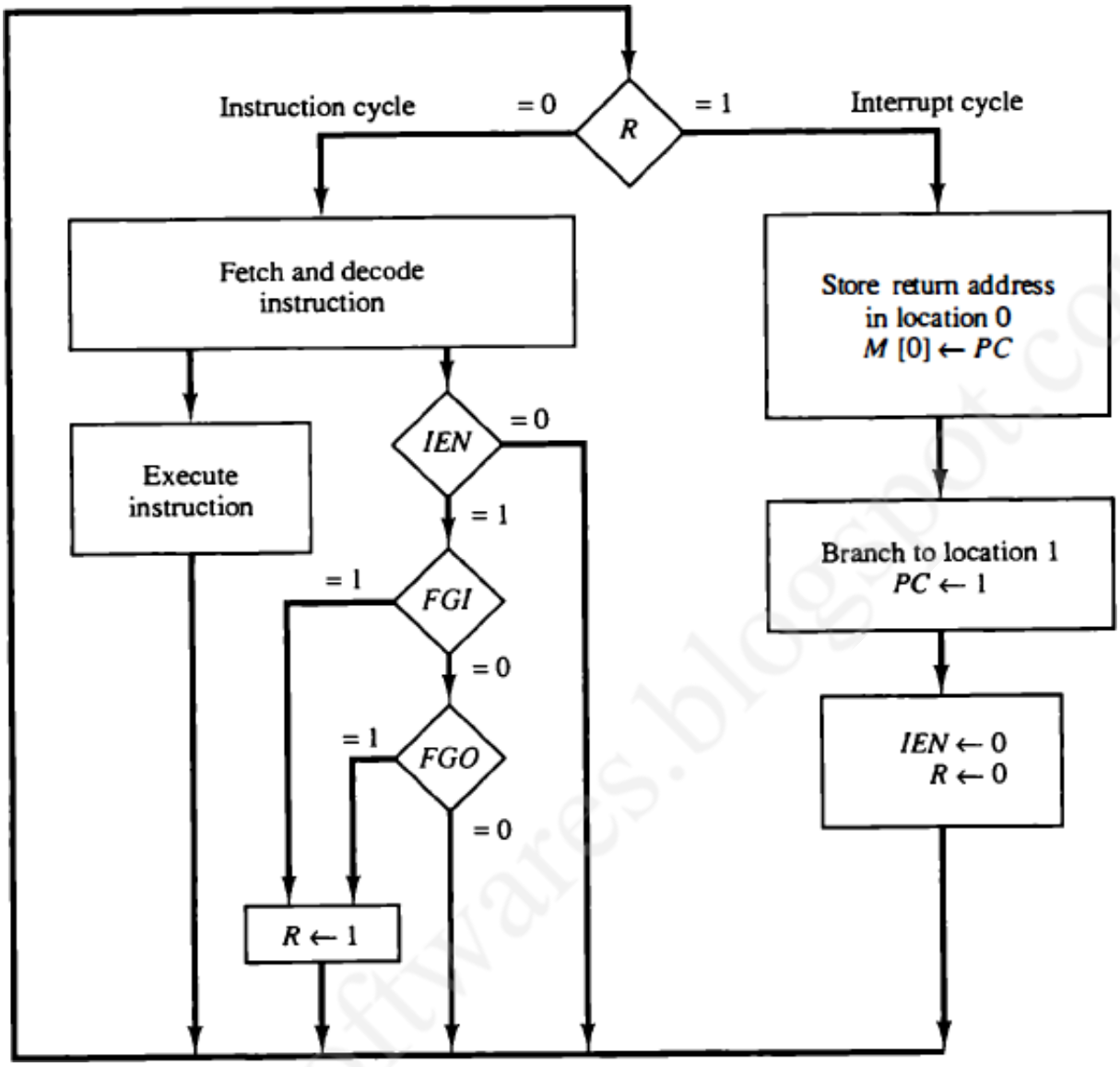


Figure (s): Flowchart for interrupt cycle

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Figure (s).

- 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 lEN 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 lEN 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 lEN = 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.

The interrupt cycle is a hardware implementation of a **branch and save return address(BSA)** operation.

EX:

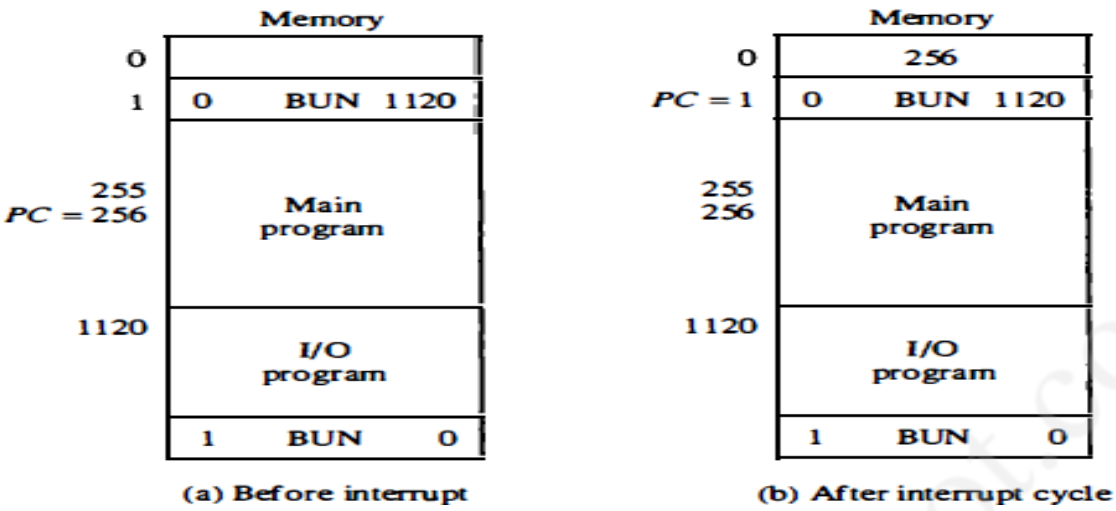


Figure (t): Demonstration of Interrupt Cycle

An example that shows what happens during the interrupt cycle is shown in Figure (t). 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 Figure (a).

When control reaches timing signal T0 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 program returns to the location where it was interrupted. This is shown in Figure (b).

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 lEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T0, T1 or T2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

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

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 lEN and R, and control goes back to T0 by clearing SC to 0. The beginning of the next instruction cycle has the condition R' T0 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.

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

Complete Computer Description:

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in the below figure (u). 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. If the instruction is one of the register-reference instructions, it will be executed. If it is an input-output instruction, it will be executed.

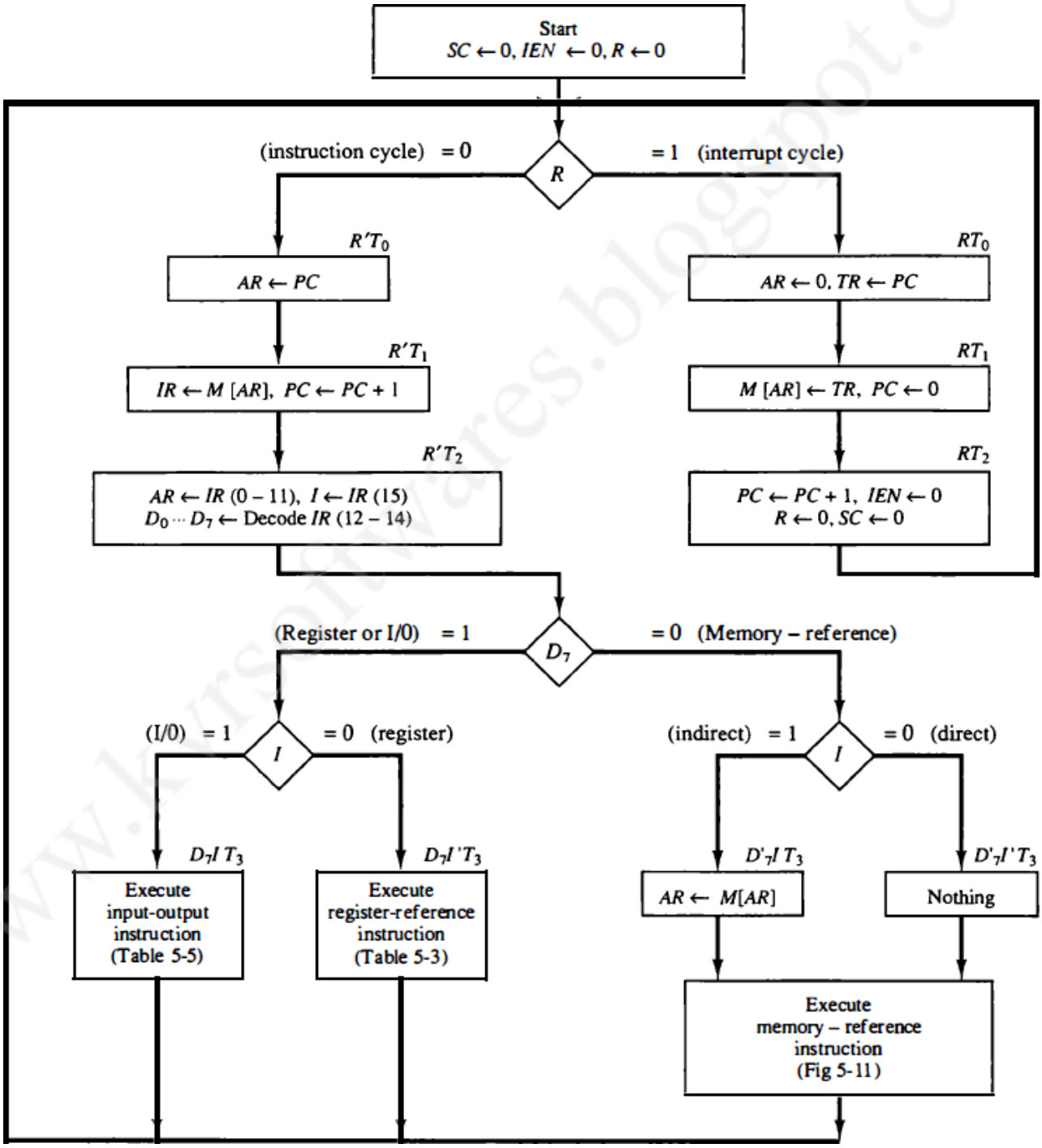


Figure (u): Flowchart for computer operation

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 a ll 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:	$IEN \leftarrow 0$

Table (m): Control functions and microoperations for the Basic computer

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, as shown in the below Table (m).

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.

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.