

# UNIT I

## Digital Computers

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations.

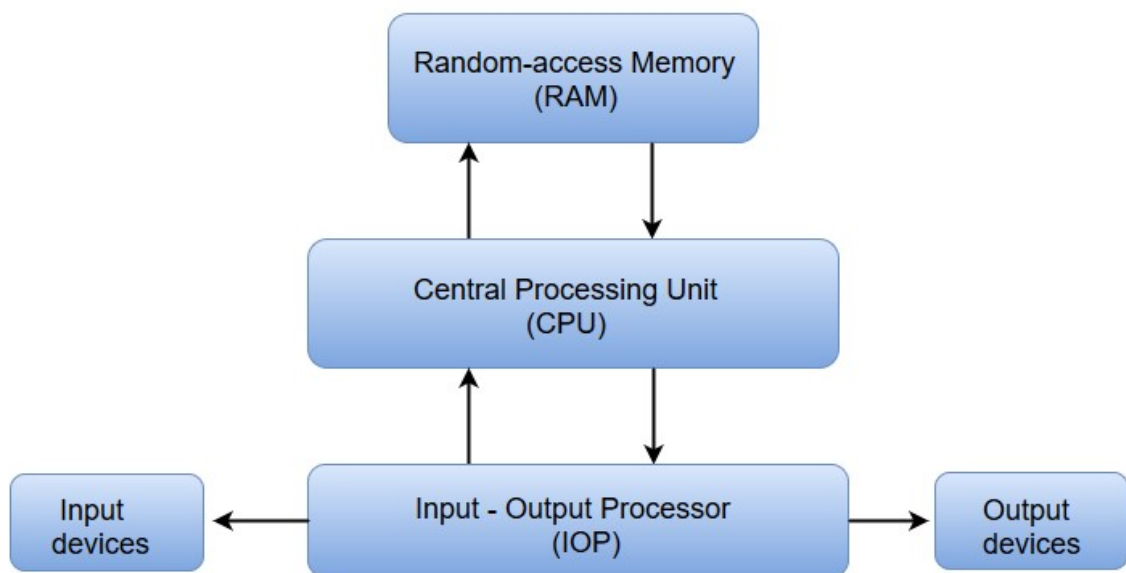
By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit.

A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

### Block diagram of a digital computer:



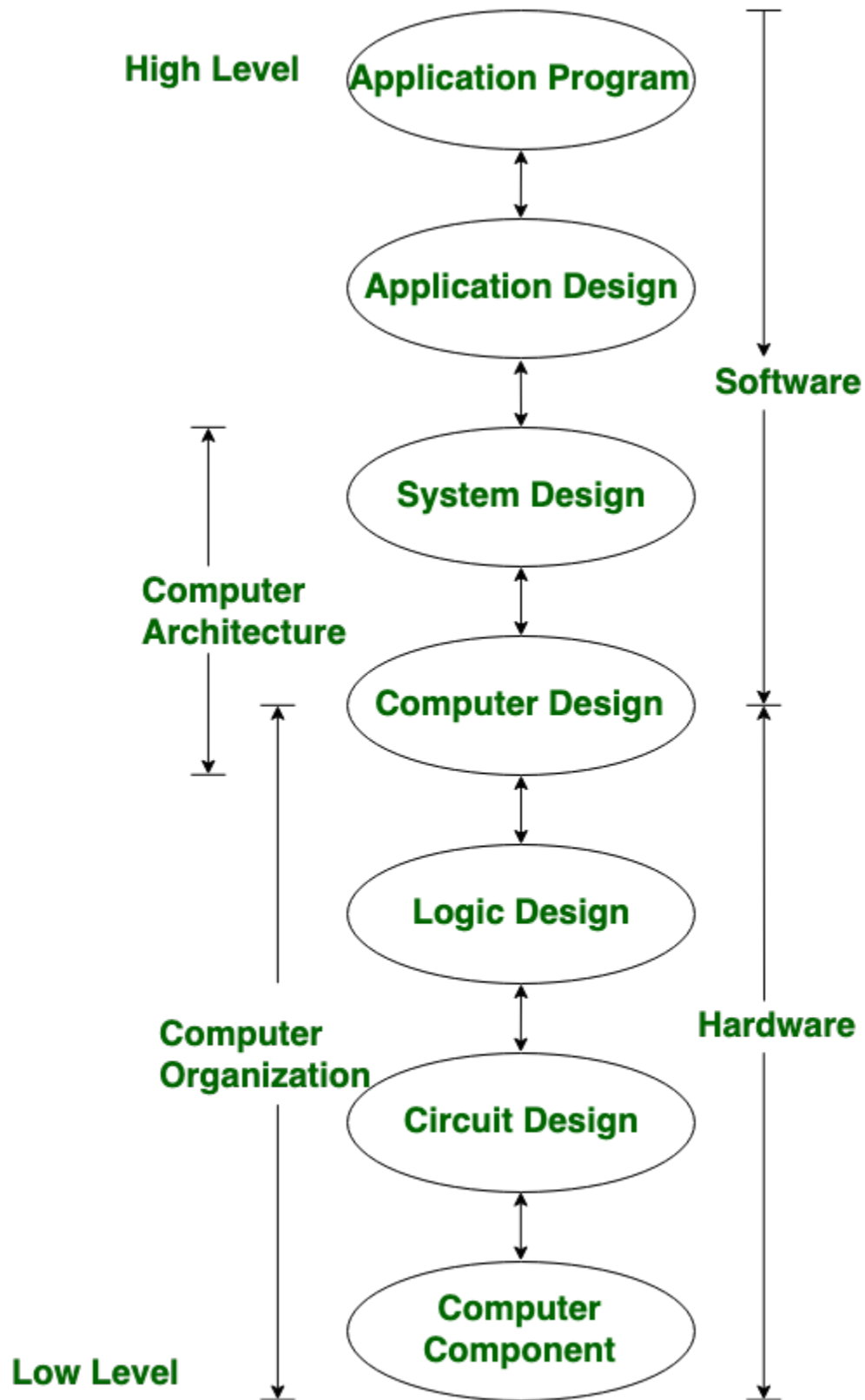
- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.
- The memory unit of a digital computer contains storage for instructions and data.
- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

## Definition of Computer Organization

Computer Architecture	Computer Organization
Computer Architecture is concerned with the way hardware components are connected together to form a computer system.	Computer Organization is concerned with the structure and behaviour of a computer system as seen by the user.
It acts as the interface between hardware and software.	It deals with the components of a connection in a system.
Computer Architecture helps us to understand the functionalities of a system.	Computer Organization tells us how exactly all the units in the system are arranged and interconnected.
A programmer can view architecture in terms of instructions, addressing modes and registers.	Whereas Organization expresses the realization of architecture.
While designing a computer system architecture is considered first.	An organization is done on the basis of architecture.
Computer Architecture deals with high-level design issues.	Computer Organization deals with low-level design issues.
Architecture involves Logic (Instruction sets, Addressing modes, Data types, Cache optimization)	Organization involves Physical Components (Circuit design, Adders, Signals, Peripherals)

## Computer Design and Computer Architecture

Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as computer implementation. Computer architecture is concerned with the structure and behavior of the computer as seen by the user.



**Computer Organization** comes after the decision of Computer Architecture first. Computer Organization is how operational attributes are linked together and contribute to realizing the architectural specification. Computer Organization deals with a structural relationship.

## **Register Transfer Language and Micro operations**

Register transfer language is a symbolic notation for describing micro-operation transfers between registers. The availability of hardware logic circuits that can perform a specified micro-operation and transfer the outcome of the operation to the same or another register is referred to as register transfer.

### **Register Transfer language**

In symbolic notation, it is used to describe the micro-operations transfer among registers. It is a kind of intermediate representation (IR) that is very close to assembly language, such as that which is used in a compiler. The term “Register Transfer” can perform micro-operations and transfer the result of operation to the same or other register.

### **Micro-operations:**

The operation executed on the data store in registers are called micro-operations. They are detailed low-level instructions used in some designs to implement complex machine instructions.

### **Register Transfer:**

The information transformed from one register to another register is represented in symbolic form by replacement operator is called Register Transfer.

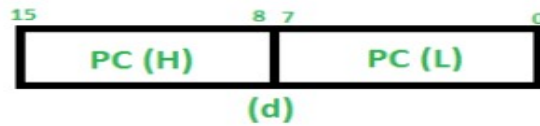
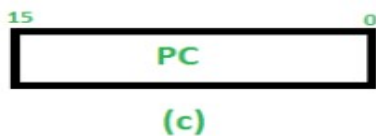
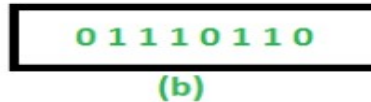
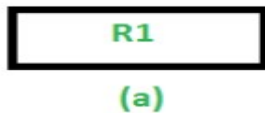
### **Replacement Operator:**

In the statement,  $R2 \leftarrow R1$ ,  $\leftarrow$  acts as a replacement operator. This statement defines the transfer of content of register R1 into register R2.

There are various methods of RTL –

1. General way of representing a register is by the name of the register enclosed in a rectangular box as shown in (a).

2. Register is numbered in a sequence of 0 to (n-1) as shown in (b).
3. The numbering of bits in a register can be marked on the top of the box as shown in (c).
4. A 16-bit register PC is divided into 2 parts- Bits (0 to 7) are assigned with lower byte of 16-bit address and bits (8 to 15) are assigned with higher bytes of 16-bit address as shown in (d).



Basic symbols of RTL :

Symbol	Description	Example
Letters and Numbers	Denotes a Register	MAR, R1, R2
( )	Denotes a part of register	R1(8-bit) R1(0-7)
<-	Denotes a transfer of information	R2 <- R1
,	Specify two micro-operations of Register Transfer	R1 <- R2 R2 <- R1
:	Denotes conditional operations	P : R2 <- R1 if P=1
Naming Operator (:=)	Denotes another name for an already existing register/alias	Ra := R1

## Register Transfer Operations:

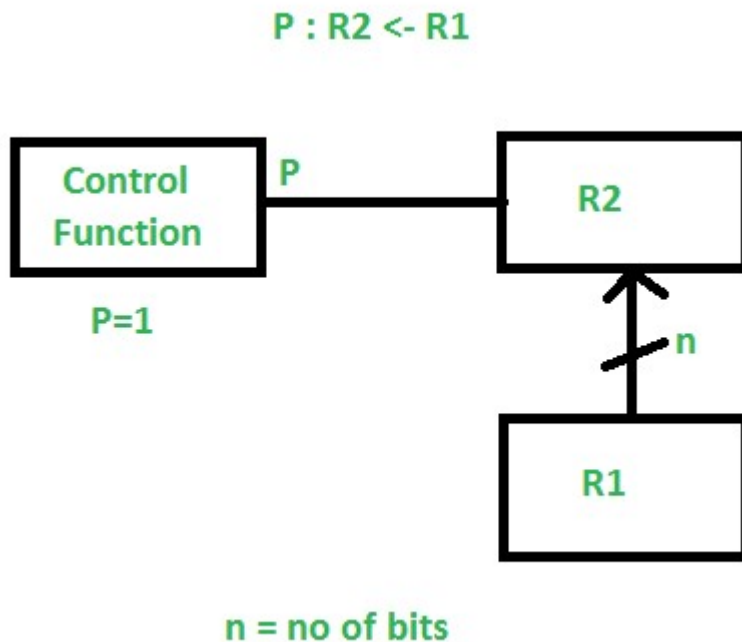
The operation performed on the data stored in the registers are referred to as register transfer operations.

There are different types of register transfer operations:

### 1. Simple Transfer – $R2 \leftarrow R1$

The content of R1 are copied into R2 without affecting the content of R1. It is an unconditional type of transfer operation.

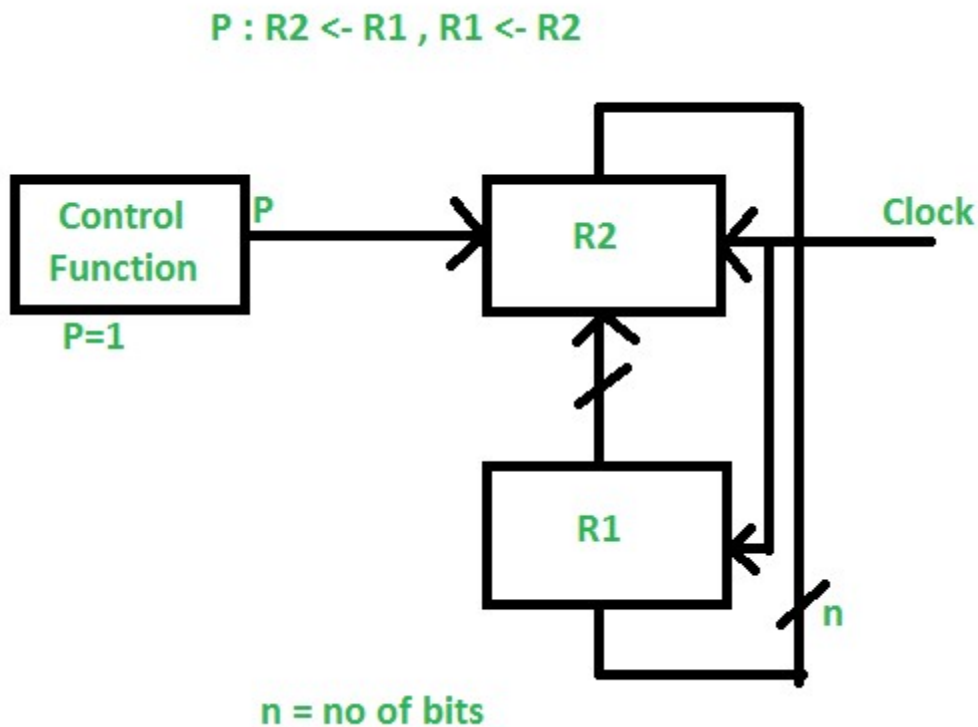
### 2. Conditional Transfer –



It indicates that if  $P=1$ , then the content of R1 is transferred to R2. It is a unidirectional operation.

### 3. Simultaneous Operations

If 2 or more operations are to occur simultaneously then they are separated with comma (,).



If the control function  $P=1$ , then load the content of R1 into R2 and at the same clock load the content of R2 into R1.

## Register Transfer

The term Register Transfer refers to the availability of hardware logic circuits that can perform a given micro-operation and transfer the result of the operation to the same or another register.

Most of the standard notations used for specifying operations on various registers are stated below.

- The memory address register is designated by **MAR**.
- Program Counter **PC** holds the next instruction's address.
- Instruction Register **IR** holds the instruction being executed.
- **R1** (Processor Register).
- We can also indicate individual bits by placing them in parenthesis. For instance, PC (8-15), R2 (5), etc.

- Data Transfer from one register to another register is represented in symbolic form by means of replacement operator. For instance, the following statement denotes a transfer of the data of register R1 into register R2.

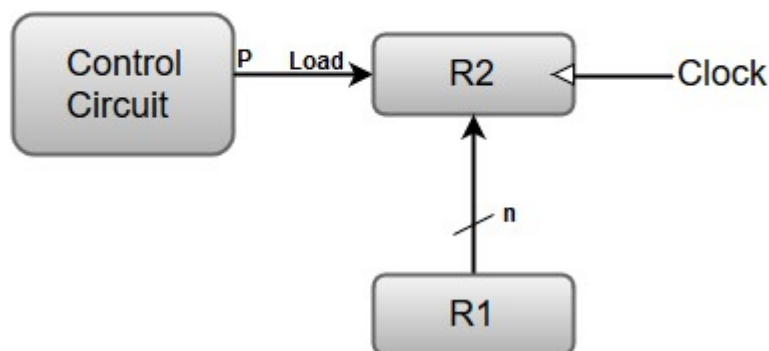
1.  $R2 \leftarrow R1$

- Typically, most of the users want the transfer to occur only in a predetermined control condition. This can be shown by following if-then statement: If (P=1) then ( $R2 \leftarrow R1$ ); Here P is a control signal generated in the control section.
- It is more convenient to specify a control function (P) by separating the control variables from the register transfer operation. For instance, the following statement defines the data transfer operation under a specific control function (P).

1. P:  $R2 \leftarrow R1$

The following image shows the block diagram that depicts the transfer of data from R1 to R2.

### Transfer from R1 to R2 when P = 1:



Here, the letter 'n' indicates the number of bits for the register. The 'n' outputs of the register R1 are connected to the 'n' inputs of register R2.

### Bus and memory transfers

A digital system composed of many registers, and paths must be provided to transfer information from one register to another. The number of wires connecting all of the registers will be excessive if separate lines are used between each register and all other registers in the system.



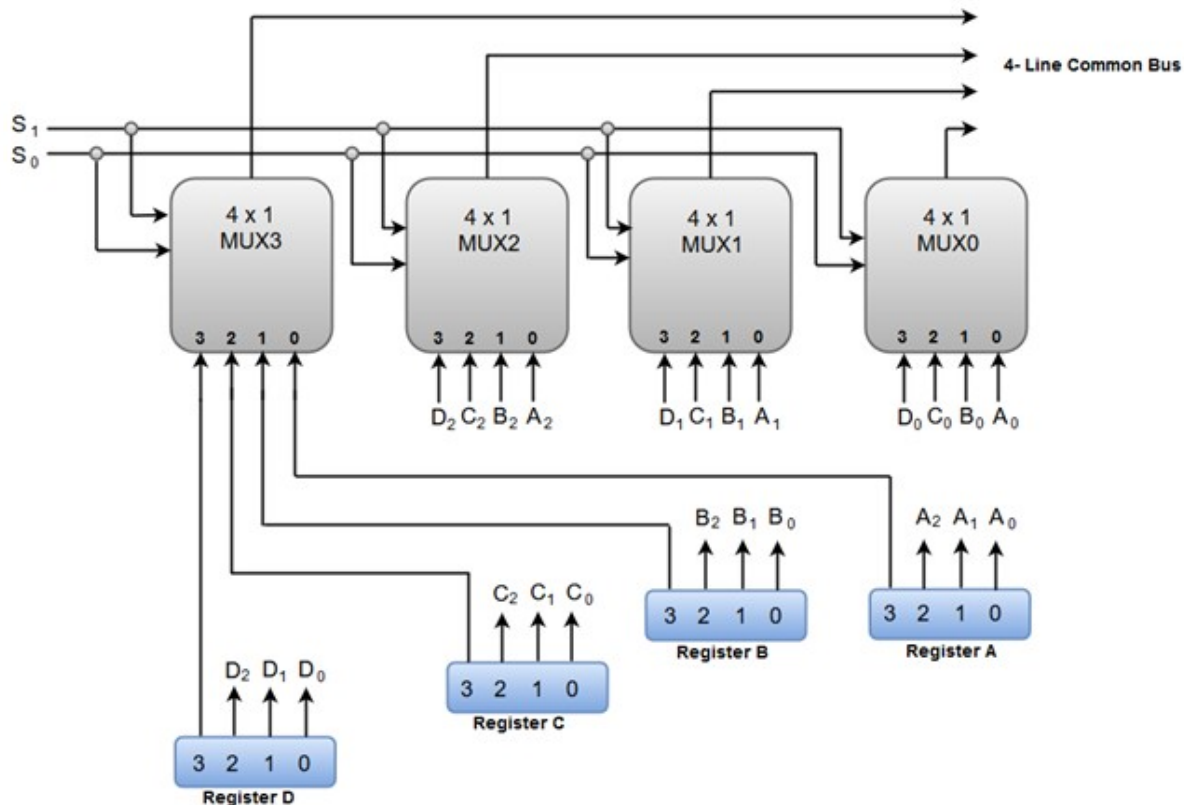
A bus structure, on the other hand, is more efficient for transferring information between registers in a multi-register configuration system.

A bus consists of a set of common lines, one for each bit of register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during a particular register transfer.

The following block diagram shows a Bus system for four registers. It is constructed with the help of four  $4 \times 1$  Multiplexers each having four data inputs (0 through 3) and two selection inputs ( $S_1$  and  $S_0$ ).

We have used labels to make it more convenient for you to understand the input-output configuration of a Bus system for four registers. For instance, output 1 of register A is connected to input 0 of MUX1.

**Bus System for 4 Registers:**



The two selection lines S1 and S2 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 both of the select lines are at low logic, i.e.  $S_1S_0 = 00$ , the 0 data inputs of all four multiplexers are selected and applied to the outputs that forms the bus. This, in turn, causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.

Similarly, when  $S_1S_0 = 01$ , register B is selected, and the bus lines will receive the content provided by register B.

The following function table shows the register that is selected by the bus for each of the four possible binary values of the Selection lines.

S1	S0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

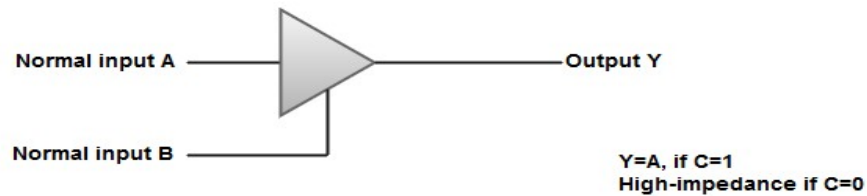
Note: The number of multiplexers needed to construct the bus is equal to the number of bits in each register. The size of each multiplexer must be ' $k * 1$ ' since it multiplexes ' $k$ ' data lines. For instance, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

A bus system can also be constructed using **three-state gates** instead of multiplexers.

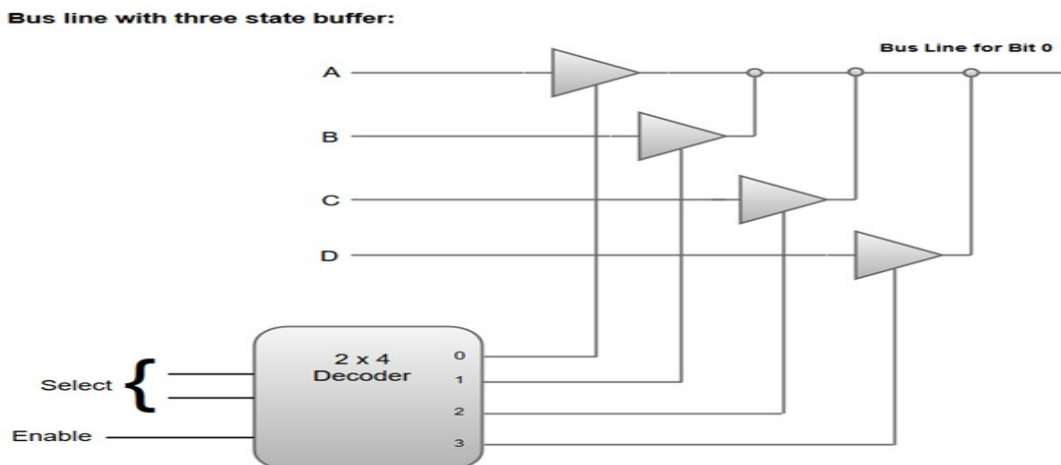
The **three state gates** can be considered as a digital circuit that has three gates, two of which are signals equivalent to logic 1 and 0 as in a conventional gate. However, the third gate exhibits a high-impedance state.

The most commonly used three state gates in case of the bus system is a **buffer gate**.

The graphical symbol of a three-state buffer gate can be represented as:



The following diagram demonstrates the construction of a bus system with three-state buffers.



- The outputs generated by the four buffers are connected to form a single bus line.
- Only one buffer can be in active state at a given point of time.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- A 2 \* 4 decoder ensures that no more than one control input is active at any given point of time.

## Memory Transfer

Most of the standard notations used for specifying operations on memory transfer are stated below.

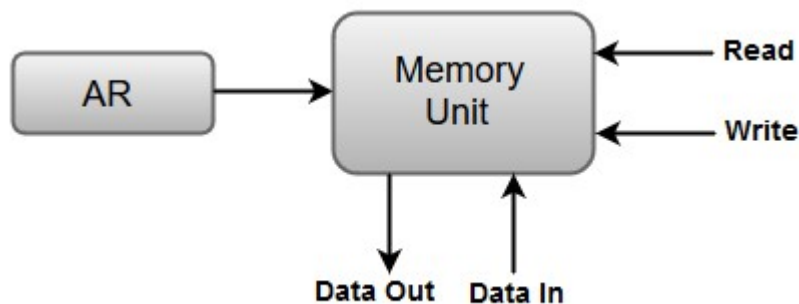
- The transfer of information from a memory unit to the user end is called a **Read** operation.
- The transfer of new information to be stored in the memory is called a **Write** operation.
- A memory word is designated by the letter **M**.
- We must specify the address of memory word while writing the memory transfer operations.
- The **address register** is designated by **AR** and the **data register** by **DR**.
- Thus, a read operation can be stated as:

Read:  $DR \leftarrow M[AR]$

- The **Read** statement causes a transfer of information into the data register (DR) from the memory word (M) selected by the address register (AR).
- And the corresponding write operation can be stated as:

Write:  $M[AR] \leftarrow R1$

- The Write statement causes a transfer of information from register R1 into the memory word (M) selected by address register (AR).



## Arithmetic Micro operations

Arithmetic micro-operations perform operations on the numeric data stored in the registers.

The basic arithmetic micro-operations are-

1. Addition
2. Subtraction
3. Increment

4. Decrement
5. 1's complement
6. 2's complement

Let's discuss these arithmetic micro-operations one by one.

### Addition

The Add arithmetic micro-operation adds the values of the two registers and stores the output in the desired register.

The symbolic notation for the Add arithmetic micro-operation is-

$$R3 \leftarrow R1 + R2$$

Here, R1 and R2 are the registers whose contents we want to add and,

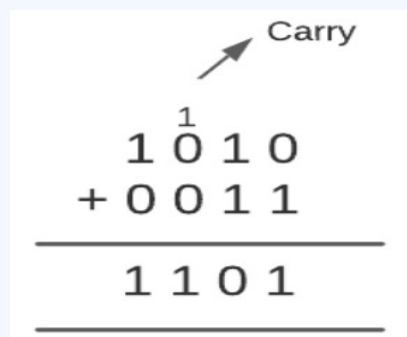
R3 is the desired register for storing the output.

**Note:** We can either store the output in another register or the same register, i.e. R1 or R2.

For example, consider the value of register R1 as 1010 and the value of register R2 as 0011. For performing the add arithmetic micro-operation remember the following rules:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 10$  (here, 0 is placed in the result and 1 is transferred as carry to the next column)

If we add R1 and R2, the output will be-



$$\begin{array}{r}
 \text{Carry} \nearrow 1 \\
 1010 \\
 + 0011 \\
 \hline
 1101 \\
 \hline
 \end{array}$$

### Subtraction

The Subtract arithmetic micro-operation subtracts the values of the two registers and stores the output in the desired register.

The symbolic notation for the Subtract arithmetic micro-operation is-

$$R3 \leftarrow R1 - R2$$

Here, R1 and R2 are the registers whose contents we want to subtract and,

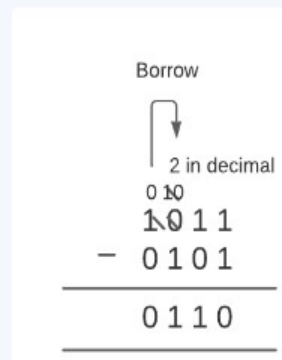
R3 is the desired register for storing the output.

**Note:** We can either store the output in another register or the same register, i.e. R1 or R2.

For example, consider the value of register R1 as 1011 and register R2 as 0101. For performing the subtract arithmetic micro-operation remember the following rules:

- $0 - 0 = 0$
- $0 - 1 = 1$  (because 10 is borrowed from next high order digit which is equal to 2 in decimal so  $2 - 1 = 1$ )
- $1 - 0 = 1$
- $1 - 1 = 0$

If we subtract R2 from R1, the output will be-



Besides the above way, there is also an alternate way of doing the arithmetic subtraction. This way includes the use of the 2's complement.

To subtract the values of two registers, we need to add the first register, the complemented value of the second register and one.

The symbolic notation is-

$$R3 \leftarrow R1 + R2' + 1$$

Here, R1 and R2 are the registers whose contents we want to subtract and,

R3 is the desired register for storing the output.

Using this method, we get the same output as  $R1 - R2$ .

**Note:** We can either store the output in another register or the same register, i.e. R1 or R2.

For example, consider the value of register R1 as 1011 and register R2 as 0101 (same as we take firstly). Now, we will perform subtraction using the alternate method.

First, we will complement the value of the register R2. 0 will be converted to 1 and 1 to 0.

Therefore, the content of R2 will become 1010.

Second, we will add R2 and 1.

$$\begin{array}{r} 1010 \\ + \quad 1 \\ \hline 1011 \\ \hline \end{array}$$

Finally, we will add R1 and R2.

$$\begin{array}{r} \text{carry} \nearrow \\ 11 \\ 1011 \\ + 1011 \\ \hline 10110 \\ \hline \end{array}$$

We will ignore the overflow bit (1 in this case). So, our output will be 0110.

### Increment

The Increment arithmetic micro-operation increments the value of a register by 1. This means this operation adds 1 to the value of the given register and stores the output in the desired register.

The symbolic notation for the Increment arithmetic micro-operation is-

$$R1 \leftarrow R1 + 1$$

Here, R1 is the register whose value we want to increment and,

R1 is also the desired register for storing the output.

**Note:** We can store the output in another register or the same register.

For example, consider the value of register R1 as 0101. For performing the increment arithmetic micro-operation, we will add 1 to R1.

$$\begin{array}{r} \text{carry} \nearrow \\ 0101 \\ + \quad 1 \\ \hline 0110 \\ \hline \end{array}$$

The Increment arithmetic micro-operations is carried out with the help of a combinational circuit or a binary up-down counter.

## Decrement

The Decrement arithmetic micro-operation decreases the value of a register by 1. This means this operation subtracts one from the value of the given register and stores the output in the desired register.

The symbolic notation for the Increment arithmetic micro-operation is-

$$R1 \leftarrow R1 - 1$$

Here, R1 is the register whose value we want to decrement and,

R1 is also the desired register for storing the output.

**Note:** We can store the output in another register or the same register.

For example, consider the value of register R1 as 0101. For performing the decrement arithmetic micro-operation, we will subtract one from R1.

$$\begin{array}{r} 0101 \\ - \quad 1 \\ \hline 0100 \\ \hline \end{array}$$

The Decrement arithmetic micro-operation is carried out with the help of a combinational circuit or a binary up-down counter.

## 1's Complement

The 1's complement arithmetic micro-operation complements the contents of a register. In this micro-operation, 0 is converted to 1 and 1 is converted to 0.

The symbolic notation for the 1's complement arithmetic micro-operation is-

$$R1 \leftarrow R1'$$

Here, R1 is the register whose value we want to complement and,

R1 is also the desired register for storing the output.

**Note:** We can store the output in another register or the same register.

For example, consider the value of register R1 as 0101. For performing the 1's complement arithmetic micro-operation, we will just convert 0 to 1 and 1 to 0.

Therefore, 1's complement of R1 will be 1010.

## 2's Complement

The 2's complement arithmetic micro-operation first complements the contents of the given register and then adds 1 to it. This micro-operation is also known as **Negation**.



The symbolic notation for the 2's complement arithmetic micro-operation is-

$$R2 \leftarrow R2' + 1$$

Here, R2 is the register on whose value we want to perform 2's complement and, R2 is also the desired register for storing the output.

**Note:** We can store the output in another register or the same register.

For example, consider the value of register R2 as 0101. For performing the 2's complement arithmetic micro-operation first, we will find the 1's complement of R2.

The 1's complement of R2 will be 1010. Then we will add 1 to it.

$$\begin{array}{r} 1010 \\ + \quad 1 \\ \hline 1011 \end{array}$$

The signals that implement these operations propagate through gates in this case, and the result of the process can be transferred into a destination register via a clock pulse immediately after the output signal propagates through the combinational circuit.

Besides the above-described arithmetic micro-operations, there are two more arithmetic micro-operations- **multiply** and **divide**. These two operations are valid arithmetic operations, but they are not part of the required set of micro-operations.

A series of add and shift micro-operations are used to perform the multiply micro-operation.

A series of subtracting and shifting micro-operations are used to complete the divide micro-operation.

The following table shows the symbolic representation of various Arithmetic Micro-operations.

Symbolic Representation	Description
$R3 \leftarrow R1 + R2$	The contents of R1 plus R2 are transferred to R3.
$R3 \leftarrow R1 - R2$	The contents of R1 minus R2 are transferred to R3.
$R2 \leftarrow R2'$	Complement the contents of R2(1's complement).
$R2 \leftarrow R2' + 1$	2's complement the contents of R2(negate).
$R3 \leftarrow R1 + R2' + 1$	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.

## Logic micro operations:

Logic micro-operations are used on the bits of data stored in registers. These micro-operations treat each bit independently and create binary variables from them.

There are a total of 16 micro-operations available. These are-

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = x.y$	$F \leftarrow A \wedge B$	AND
$F_2 = x.y'$	$F \leftarrow A \wedge \bar{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'.y$	$F \leftarrow \bar{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	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
$F_{10} = y'$	$F \leftarrow \bar{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \bar{B}$	
$F_{12} = x'$	$F \leftarrow \bar{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \bar{A} \vee B$	
$F_{14} = (x.y)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

Before discussing these logic micro-operations, let's discuss their truth tables.

The below diagram shows the truth table for all the 16 logic micro-operations mentioned above.

Here, x and y are the variables or registers in which the data is stored and F0, F1, ....., F15 are the outputs that occur after performing these logic micro-operations.

x	y	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>	F <sub>6</sub>	F <sub>7</sub>	F <sub>8</sub>	F <sub>9</sub>	F <sub>10</sub>	F <sub>11</sub>	F <sub>12</sub>	F <sub>13</sub>	F <sub>14</sub>	F <sub>15</sub>
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

Now, we will discuss these logic micro-operations one by one.

## 1. Clear

The Clear logic micro-operation is used to clear the register or set the bits of the register to 0. To use this micro-operation, we need to feed 0 to the register. In the above truth table, F0 represents the truth table of Clear logic micro-operation.

For example,  $F \leftarrow 0$  means the value of the register F is set to 0 or is cleared. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Clear logic micro-operation is  $F0 = 0$

## 2. AND

The AND logic micro-operation performs the logical AND between the bits of the data stored in the two registers. The symbol to represent the logical AND is  $\wedge$ .

### Case 1: Both x and y values are true.

In the first case, if the values of both two registers are true then the result of AND operation is 1; else, it is 0. F1 represents the truth table of AND logic micro-operation in the above truth table.

For example,  $F \leftarrow A \wedge B$  means the registers A and B value will undergo AND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the AND logic micro-operation will be  $F1 = x.y$

### Case 2: x is true, and y is false.

The logical AND operation we discussed above gives output 1 when both x and y are true. There is also another AND operation which includes x but not y. Also known as **inhibition**, here for performing the AND operation, the first value is taken from the x variable or register. The second value is taken as the **complement** of the y variable or register. If the value of the x register is true and of the y register is false, then the result of AND operation is 1; else, it is 0.

F2 represents the truth table of inhibition AND logic micro-operation in the above truth table.

For example,  $F \leftarrow A \wedge B'$  means the value of the registers A and complement B will undergo AND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the AND logic micro-operation will be  $F2 = x.y'$

### Case 3: x is false, and y is true.

The third case of logical AND operation includes y but not x. Also known as **inhibition**, here for performing the AND operation, the first value is taken as the **complement** of the x variable or register, and the second value is taken from the y variable or register. If the value of the x register is false and of the y register is true, then the result of AND operation is 1; else, it is 0.

F4 represents the truth table of inhibition AND logic micro-operation in the above truth table.

For example,  $F \leftarrow A' \wedge B$  means the value of the complement register A and as it is B will undergo AND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the AND logic micro-operation will be  $F4 = x'.y$

### 3. Transfer A

The Transfer A logic micro-operation transfers the contents of register A (first register) to the output register.

F3 represents the truth table of Transfer A logic micro-operation in the above truth table. Since there is a transfer of data from the first register to the output register in this micro-operation, its truth table is the same as the taken values of the x variable (0, 0, 1, 1).

For example,  $F \leftarrow A$  means the value of register A is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Transfer A logic micro-operation is  $F3 = x$

### 4. Transfer B

The Transfer B logic micro-operation transfers the contents of register B (second register) to the output register.

F5 represents the truth table of Transfer B logic micro-operation in the above truth table. Since there is a transfer of data from the second register to the output register in this micro-operation, its truth table is the same as the taken values of the y variable (0, 1, 0, 1).

For example,  $F \leftarrow B$  means the value of register B is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Transfer B logic micro-operation is  $F5 = y$

## 5. Exclusive OR

Also known as XOR, this logic micro-operation performs the logical XOR between the data bits stored in the two registers. The logical XOR means either x should be true or y but not both. The symbol to represent the Exclusive OR is  $\oplus$ .

F6 represents the truth table of Exclusive OR logic micro-operation in the above truth table. The output will be 1 when either  $x = 1$  and  $y = 0$  or  $x = 0$  and  $y = 1$ .

For example,  $F \leftarrow A \oplus B$  means the registers A and B value will undergo XOR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the Exclusive OR logic micro-operation will be  $F6 = x.y' + x'.y$

## 6. OR

The OR logic micro-operation performs the logical OR between the data bits stored in the two registers. The symbol to represent the logical OR is  $\vee$ .

**Case 1: Either x or y or both x and y values are true.**

In the first case, if either the value of x register is true and y register is false, or the value of x register is false, and y register is true, or both the values of x and y registers are true, then the result of OR operation is 1 else it is 0. F7 represents the truth table of OR logic micro-operation in the above truth table.

For example,  $F \leftarrow A \vee B$  means the registers A and B value will undergo OR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the OR logic micro-operation will be  $F7 = x + y$

**Case 2: If y, then x else not.**

In the second case, the output for 1 follows the condition that

- If the value of the y register is true, then the value of the x register must be true. If this condition is satisfied, then the output is 1.
- If the value of the y register is false, then we don't need to look for the value of the x register, and the output is 1.
- Else the output is 0.

To perform this logic micro-operation, we need to perform the logical OR of the values of the x register and the complement value of the y register.

In the above truth table, F11 represents the truth table of this logic micro-operation.

For example,  $F \leftarrow A \vee B'$  means the value of the registers A and complement B will undergo OR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for this OR logic micro-operation will be  $F11 = x + y'$

### Case 3: If x, then y else not.

In the second case, the output for 1 follows the condition that

- If the value of the x register is true, then the y register's value must be true. If this condition is satisfied, then the output is 1.
- If the value of the x register is 0, then we don't need to look for the value of the y register, and the output is 1.
- Else the output is 0.

To perform this logic micro-operation, we need to perform the logical OR of the complemented value of the x register and the value of the y register.

In the above truth table, F13 represents the truth table of this logic micro-operation.

For example,  $F \leftarrow A' \vee B$  means the complemented register A and B value will undergo OR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for this OR logic micro-operation will be  $F13 = x' + y$

## 7. NOR

The NOR logic micro-operation is simply the opposite of OR logic micro-operation. As the name suggests, it is Not OR. The output of OR micro-operation is 1 when the value of either x register or y register or both x and y registers are true. In contrast, in NOR, the output is 0 when the value of either x register or y register or both x and y registers are true, and it is 1 when both x and y registers are false. In the above truth table, F8 represents the truth table of NOR logic micro-operation.

For example,  $F \leftarrow (A \vee B)'$  means the registers A and B value will undergo NOR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the Transfer A logic micro-operation is  $F8 = (x + y)'$

## 8. Exclusive NOR

If we perform the Exclusive NOR micro-operation, the output will be 1 when the values of both the x and y registers will be the same. They can be true or false, but they have to be the same.

F9 represents the truth table of Exclusive NOR logic micro-operation in the above truth table.

The output will be 1 when either  $x = 0$  and  $y = 0$  or  $x = 1$  and  $y = 1$ .

For example,  $F \leftarrow (A \oplus B)'$  means the registers A and B value will undergo Exclusive NOR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the Exclusive NOR logic micro-operation will be  $F9 = x.y + x'.y'$

## 9. Complement B

The Complement B logic micro-operation transfers the complemented contents of register B (second register) to the output register. First, the content of the register is complemented and then moved to the desired register.

In the above truth table, F10 represents the truth table of Complement B logic micro-operation. Since there is a transfer of complemented data from the second register to the output register in this micro-operation, its truth table is just the opposite of the taken values of the y variable (1, 0, 1, 0).

For example,  $F \leftarrow B'$  means the complemented value of register B is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Complement B logic micro-operation is  $F10 = y'$

## 10. Complement A

The Complement A logic micro-operation transfers the complemented contents of register A (first register) to the output register. First, the content of the register is complemented and then moved to the desired register.

F12 represents the truth table of Complement A logic micro-operation in the above truth table. Since there is a transfer of complemented data from the first register to the output register in this micro-operation, its truth table is just the opposite of the taken values of the x variable (1, 1, 0, 0).

For example,  $F \leftarrow A'$  means the complemented value of register A is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Complement A logic micro-operation is  **$F12 = x'$**

### 11. NAND

The NAND logic micro-operation is simply the opposite of AND logic micro-operation. As the name suggests, it is Not AND. The output of AND micro-operation is 1 when the value of both the x register and y register is true. In contrast, in NAND, the output is 0 when the value of both x register and y register is true, and it is 1 when either x is false, or y is false, or both are false.

In the above truth table, F14 represents the truth table of NAND logic micro-operation.

For example,  **$F \leftarrow (A \wedge B)'$**  means the registers A and B value will undergo NAND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the NAND logic micro-operation is  **$F14 = (x.y)'$**

### 12. Set to all 1's

The set to all 1's logic micro-operations is used to set all the register bits to 1. To use this micro-operation, we just need to feed 1 to the register. In the above truth table, F15 represents the truth table of Set to all 1's logic micro-operation.

For example,  **$F \leftarrow 1$**  means the value of the register F is set to 1. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Clear logic micro-operation is  **$F15 = 1$**

## Shift micro operations

Shift micro-operations are used when the data is stored in registers. These micro-operations are used for the serial transmission of data. Here, the data bits are shifted from left to right. These micro-operations are also combined with arithmetic and logic micro-operations and data-processing operations.

There are three types of shift micro-operations-

1. Logical Shift
2. Arithmetic Shift
3. Circular Shift

Let's start with logical shift micro-operation.



## Logical Shift

The logical shift micro-operation moves the 0 through the serial input. There are two ways to implement the logical shift.

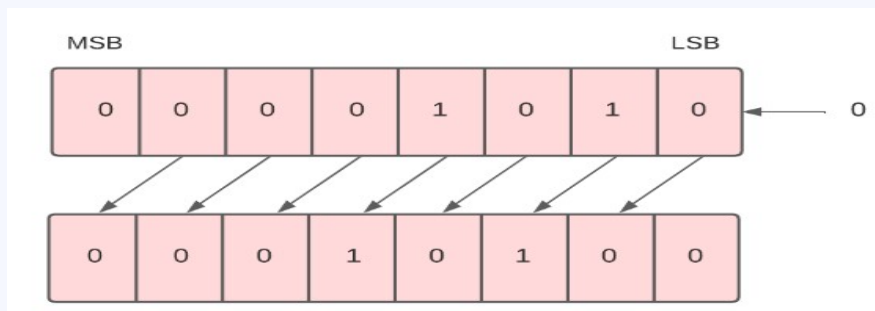
1. Logical Shift Left
2. Logical Shift Right

Let's discuss both of them one by one.

### Logical Shift Left

Each bit in the register is shifted to the left one by one in this shift micro-operation. The most significant bit (MSB) is moved outside the register, and the place of the least significant bit (LSB) is filled with 0.

For example, in the below data, there are 8 bits 00001010. When we perform a logical shift left on these bits, all these bits will be shifted towards the left. The MSB or the leftmost bit i.e. 0 will be moved outside, and at the rightmost place or LSB, 0 will be inserted as shown below.



To implement the logical shift left micro-operation, we use the **shl** symbol.

For example, R1 -> shl R1.

This command means the 8 bits present in the R1 register will be logically shifted left, and the result will be stored in register R1.

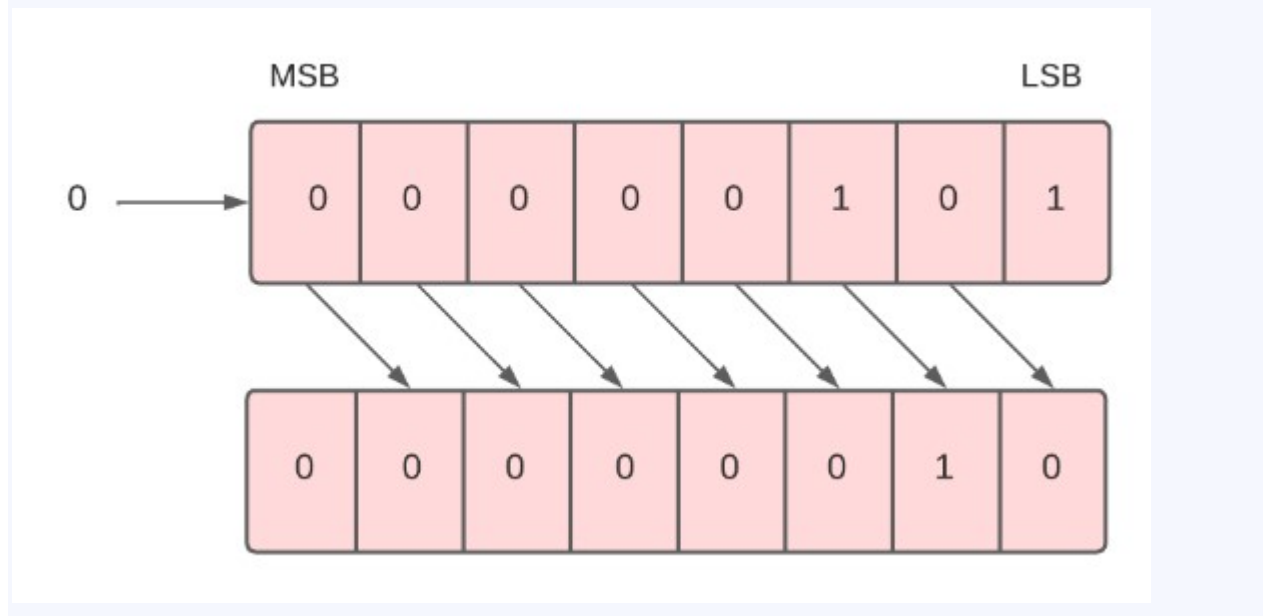
Moreover, the logical shift left microoperation denotes the multiplication of 2. The example we've taken above when converted into decimal forms the number 10. And the result after the logical shift operation when converted to decimal forms the number 20.

Next, we will see the logical shift right micro-operation.

### Logical Shift Right

Each bit in the register is shifted to the right one by one in this shift micro-operation. The least significant bit (LSB) is moved outside the register, and the place of the most significant bit (MSB) is filled with 0.

For example, in the below data, there are 8 bits 00000101. When we perform a logical shift right on these bits, all these bits will be shifted towards the right. The LSB or the rightmost bit i.e. 1 will be moved outside, and at the leftmost place or MSB, 0 will be inserted as shown below.



## Arithmetic logic shift unit

Arithmetic Logic Shift Unit (ALSU) is a part of Arithmetic Logic Unit (ALU) of a computer system. It is a digital circuit that performs arithmetic calculations, logical and shift operations. 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. ALSU consists of arithmetic, logical and shift circuits. In each stage, the circuit can perform 8 arithmetic operations, 16 logical operations and 2 shift operations.

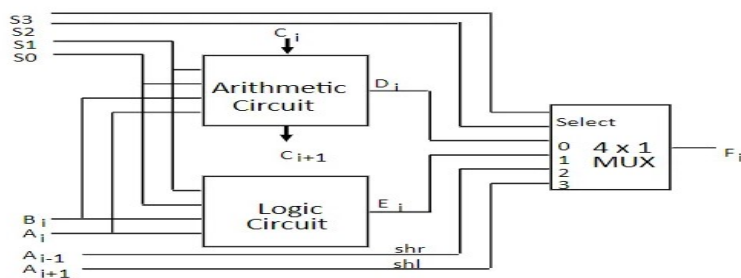


Figure 1 — Arithmetic Logic Shift Unit

## Characteristics of the ALSU

1. Arithmetic logic shift unit in digital circuit performs all the logical and arithmetic operations.
2. The operations like addition, subtraction, multiplication and division are referred as Arithmetic operations.
3. The logical operations refer to operations on numbers and special character operations.

The ALSU is responsible for the conditions given below:

1. Equal to
2. Less than
3. Greater than

## Functions of the ALSU

A particular microoperation is selected with inputs  $S_1$  and  $S_0$ . A 4 x 1 multiplexer at the output chooses between an arithmetic output in  $E_i$  and a logic output in  $H_i$ . The data in the multiplexer are selected with inputs  $S_3$  and  $S_2$ . The other two data inputs to the multiplexer receive inputs  $A_i$  — 1 for the shift-right operation and  $A_i + 1$  for the shift-left operation.

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

Table 1 — Functions of Arithmetic Logic Shift Unit

The circuit of Figure 1 must be repeated  $n$  times for an  $n$ -bit ALU. The output carry  $C_{i+1}$  of a given arithmetic stage must be connected to the input carry  $C_i$  of the next stage in sequence. The input carry to the first stage is the input carry  $C_{in}$ , which provides a selection variable for the arithmetic operations.

## Basic Computer Organization and Design: Instruction codes

### Instruction Codes

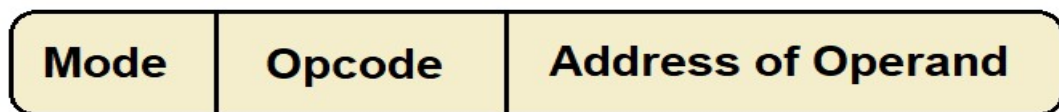
Instruction codes in computer architecture are concise bit groups directing computers to execute specific operations efficiently. Instruction codes and addresses are unique to each computer.

Generally, we can categorize instruction codes and addresses into operation codes (Opcodes) and addresses. Opcodes specify how to perform a specific instruction. An address specifies which we should use- a register or an area, for a particular action. Operands are precise computer instructions that show what information the computer requires to function.

Instruction codes are bits that instruct the computer to execute a specific operation. An instruction comprises groups called fields. These fields include:

An instruction comprises groups called fields. These fields include:

- The Operation code (Opcode) field determines the process that needs to perform.
- The Address field contains the operand's location, i.e., register or memory location.
- The Mode field specifies how the operand locates.



### Structure of an Instruction Code

The instruction code is also known as an instruction set. It is a collection of binary codes. It represents the operations that a computer processor can perform. The structure of an instruction code can vary. It depends on the architecture of the processor but generally consists of the following parts:

- **Opcode:** The opcode (Operation code) represents the operation that the processor must perform. It might indicate that the instruction is an arithmetic operation such as addition, subtraction, multiplication, or division.

- **Operand(s):** The operand(s) represents the data that the operation must be performed on. This data can take various forms, depending on the processor's architecture. It might be a register containing a value, a memory address pointing to a location in memory where the data is stored, or a constant value embedded within the instruction itself.
- **Addressing mode:** The addressing mode represents how the operand(s) can be interpreted. It might indicate that the operand is a direct address in memory, an indirect address (i.e. a memory address stored in a register), or an immediate value (i.e. a constant embedded within the instruction).
- **Prefixes or modifiers:** Some instruction sets may include additional prefixes or modifiers that modify the behavior of the instruction. For example, they may specify that the operation should be performed only if a specific condition is met or that the instruction should be executed repeatedly until a specific condition is met.

### Types of Instruction Code

There are various types of instruction codes. They are classified based on the number of operands, the type of operation performed, and the addressing modes used. The following are some common types of instruction codes:

1. **One-operand instructions:** These instructions have one operand and perform an operation on that operand. For example, the "neg" instruction in the x86 assembly language negates the value of a single operand.
2. **Two-operand instructions:** These instructions have two operands and perform an operation involving both. For example, the "add" instruction in x86 assembly language adds two operands together.
3. **Three-operand instructions:** These instructions have three operands and perform an operation that involves all three operands. For example, the "fma" (fused multiply-add) instruction in some processors multiplies two operands together, adds a third operand, and stores the result in a fourth operand.

4. **Data transfer instructions:** These instructions move data between memory and registers or between registers. For example, the "mov" instruction in the x86 assembly language moves data from one location to another.
5. **Control transfer instructions:** These instructions change the flow of program execution by modifying the program counter. For example, the "jmp" instruction in the x86 assembly language jumps to a different location in the program.
6. **Arithmetic instructions:** These instructions perform mathematical operations on operands. For example, the "add" instruction in x86 assembly language adds two operands together.
7. **Logical instructions:** These instructions perform logical operations on operands. For example, the "and" instruction in x86 assembly language performs a bitwise AND operation on two operands.
8. **Comparison instructions:** These instructions compare two operands and set a flag based on the result. For example, the "cmp" instruction in x86 assembly language compares two operands and sets a flag indicating whether they are equal, greater than, or less than.
9. **Floating-point instructions:** These instructions perform arithmetic and other operations on floating-point numbers. For example, the "fadd" instruction in the x86 assembly language adds two floating-point numbers together.

## Opcodes

An opcode is a collection of bits representing the basic operations, including add, subtract, multiply, complement, and shift. The number of bits required for the opcode is determined by the number of functions the computer gives. For '2n' operations, the minimum bits accessible to the opcode should be 'n', where n is the number of bits. We implement these operations on information saved in processor registers or memory.

## Types of Opcodes

There are three different types of instruction codes on the main computer. The instruction's operation code (opcode) is 3 bits long, and the remaining 13 bits are determined by the operation code encountered.

There are three types of formats:

1. **Memory Reference Instruction:** It specifies the address with 12 bits and the addressing mode with 1 bit (I). For direct addresses, I equal 0, while for indirect addresses, I equal 1.
2. **Register Reference Instruction:** The opcode 111 with a 0 in the leftmost bit of the instruction recognizes these instructions. The remaining 12 bits specify the procedure to be carried out.
3. **Input-Output Instruction:** The operation code 111 with a 1 in the leftmost bit of instruction recognizes these instructions. The input-output action is specified using the remaining 12 bits.

## Address

We represent the memory address where a given instruction is built. We use an instruction code's address bits as an operand rather than an address. The instruction in such methods has an immediate operand. The command is directed to a direct address if the second portion contains an address.

In the second half, there is another choice, which includes the operand's address. It points to an oblique address. One bit might indicate whether the instruction code executes the direct or indirect address.

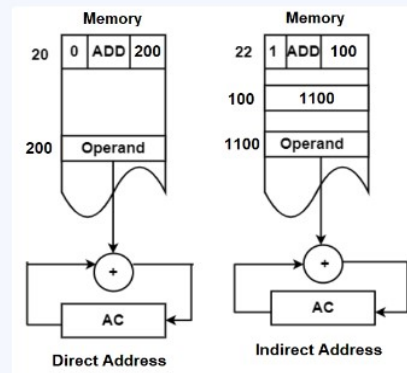
## Addressing Modes

We can mainly describe the address field for instruction in the following ways:

- **Direct Addressing** – Uses the address of the operand.
- **Indirect Addressing** – Enables the address as a pointer to the operand.
- **Immediate operand** – The second part of the instruction code specifies an operand.

**Accumulator Register (AC):** This register is found in single register processors (AC), and it performs all operations with memory operands.

**Effective Address (EA)** is the address of the operand or the target address. It defines the address that we can execute as a target address for a branch-type instruction or the address we can use directly to create an operand for a computation-type instruction without any changes.



## Computer Registers Computer instructions

Computer registers are memory storing units that operate at high speed. It's a component of a computer's processor. It can hold any type of data, including a bit sequence or a single piece of data.

Eight registers, a memory unit, and a control unit make up a basic computer. These devices must be connected on a regular basis.

Following is the list of some of the most common registers in computer:

Register	Symbol	Number of Bits	Function
Accumulator	AC	16	It's a processor register.
Program counter	PC	12	It stores the address of the instruction.
Address Register	AR	12	It is used for storing memory addresses.
Data Register	DR	16	It is a general-purpose register used for storing data during calculations.
Instruction Register	IR	16	It stores the current instruction being executed.
Temporary Register	TR	16	It holds the temporary data.
Input Register	INPR	8	It carries the input character.
Output Register	OUTR	8	It carries the output character

## Types of Registers Used in Computer

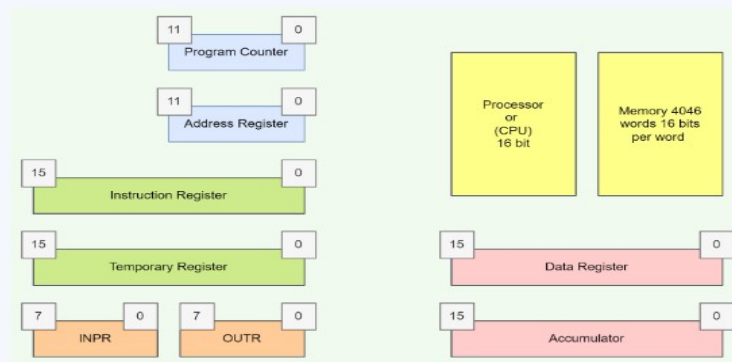
Registers are a type of computer memory used to accept, store, and transfer data and instructions used by the CPU right away. Processor registers refer to the registers used by the CPU.

During the execution of a program, registers are used to store data temporarily.

In most cases, the number of bits that a register can hold is used to determine its size.



The common registers in a computer and the memory are depicted in the diagram below:



Basic computer Registers and memory

The following are the various computer registers and their functions:

- Accumulator Register(AC):** Accumulator Register is a general-purpose Register. The initial data to be processed, the intermediate result, and the final result of the processing operation are all stored in this register. If no specific address for the result operation is specified, the result of arithmetic operations is transferred to AC. The number of bits in the accumulator register equals the number of bits per word
- Address Register(AR):** The Address Register is the address of the memory location or Register where data is stored or retrieved. The size of the Address Register is equal to the width of the memory address is directly related to the size of the memory because it contains an address. If the memory has a size of  $2^n * m$ , then the address is specified using  $n$  bits
- Data Register(DR):** The operand is stored in the Data Register from memory. When a direct or indirect addressing operand is found, it is placed in the Data Register. This value was then used as data by the processor during its operation. It's about the same size as a word in memory
- Instruction Register(IR):** The instruction is stored in the Instruction Register. The instruction register contains the currently executed instruction. Because it includes instructions, the number of bits in the Instruction Register equals the number of bits in the

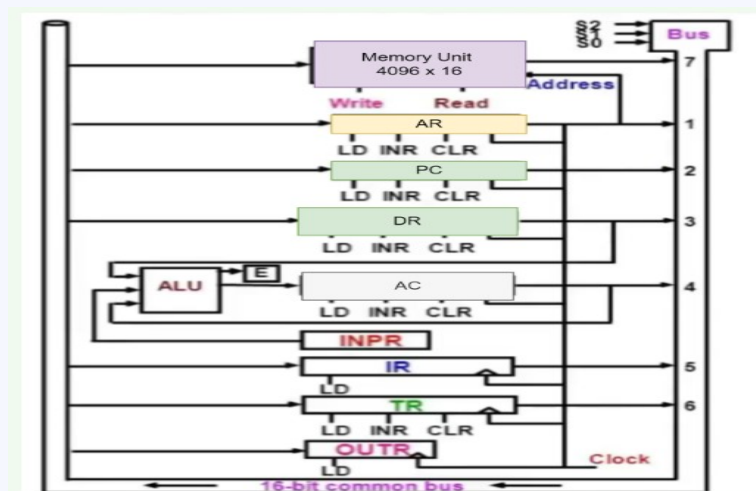
instruction, which is n bits for an n-bit CPU

- **Input Register(INPR):** Input Register is a register that stores the data from an input device. The computer's alphanumeric code determines the size of the input register
- **Program Counter(PC):** The Program Counter serves as a pointer to the memory location where the next instruction is stored. The size of the PC is equal to the width of the memory address, and the number of bits in the PC is equal to the number of bits in the PC
- **Temporary Register(TR):** The Temporary Register is used to hold data while it is being processed. As Temporary Register stores data, the number of bits it contains is the same as the number of bits in the data word
- **Output Register(OUTR):** The data that needs to be sent to an output device is stored in the Output Register. Its size is determined by the alphanumeric code used by the computer

### Common Bus System

A bus is a pair of signal lines that allows multi-bit data to be transferred from one system to another. A common bus is a more efficient method of sending data in a system with multiple registers. The common bus connects the outputs of seven registers and memory. A bus provides a means for people to communicate with one another.

The basic computer registers and memory connection to a common bus system are depicted in the figure below:



## Timing and Control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

1. hardwired control and
2. microprogrammed control.

**In the hardwired organization**, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed.

**In the microprogrammed control**, any required changes or modifications can be done by updating the microprogram in control memory.

The block diagram of the control unit is shown in Fig. 5.6.

It consists of two decoders,

1. a sequence counter, and
2. a number of control logic gates.

An instruction read from memory is placed in the instruction register (IR).position of this register in the common bus system is indicated in Fig 5.4

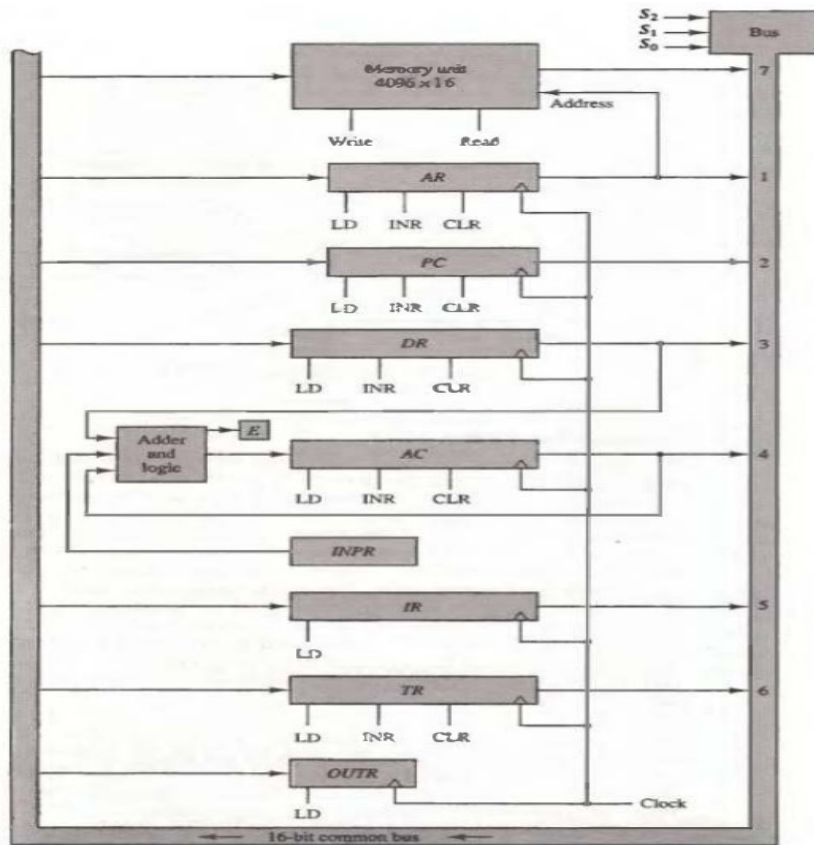


Figure 5-4 Basic computer registers connected to a common bus.

The instruction register is shown again in Fig. 5.6, where it is divided into three parts:

1. the 1 bit,
2. the operation code, and
3. bits 0 through 11.

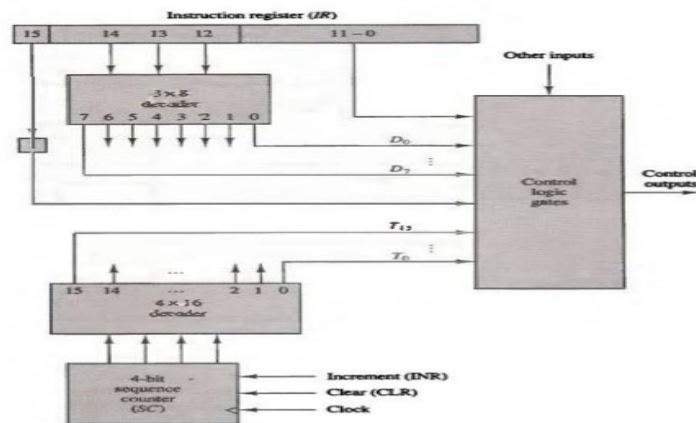


Figure 5-6 Control unit of basic computer.

The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols  $D_0$  through  $D_7$ . The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals  $T_0$  through  $T_{15}$ .

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be  $T_0$ .

As an example, consider the case where SC is incremented to provide timing signals  $T_0$ ,  $T_1$ ,  $T_2$ ,  $T_3$ , and  $T_4$  in sequence. At time  $T_4$ , SC is cleared to 0 if decoder output  $D_3$  is active. This is expressed symbolically by the statement

$D_3T_4: SC \leftarrow 0$

The timing diagram of Fig. 5-7 shows the time relationship of the control signals.

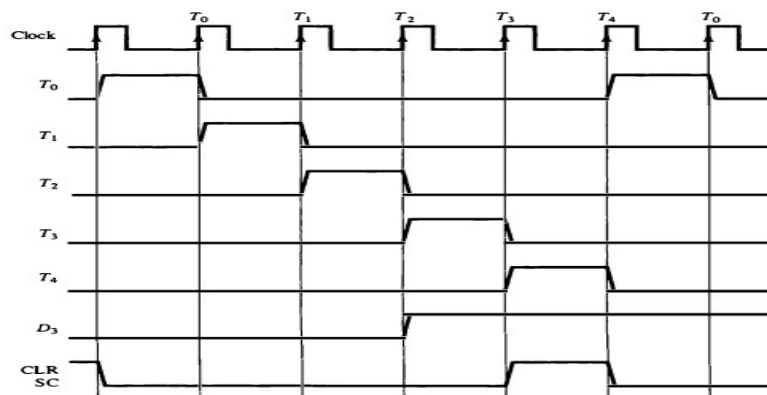


Figure 5-7 Example of control timing signals.

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. The positive clock transition labeled  $T_0$  in the diagram will trigger only those registers whose control inputs are transition, to timing signal  $T_0$ . SC is incremented with every positive clock transition unless its CLR input is active. This produces the sequence of timing signals  $T_0$ ,  $T_1$ ,  $T_2$ ,  $T_3$ ,  $T_4$  and so on, as shown in the diagram. (Note the the relationshuip between the timing signal

and its corresponding positive clock transition.) If SC is not cleared, the timing signals will continue with  $T_5$ ,  $T_6$  up to  $T_{15}$  and back to  $T_0$

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

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

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

$T_0: AR \leftarrow PC$

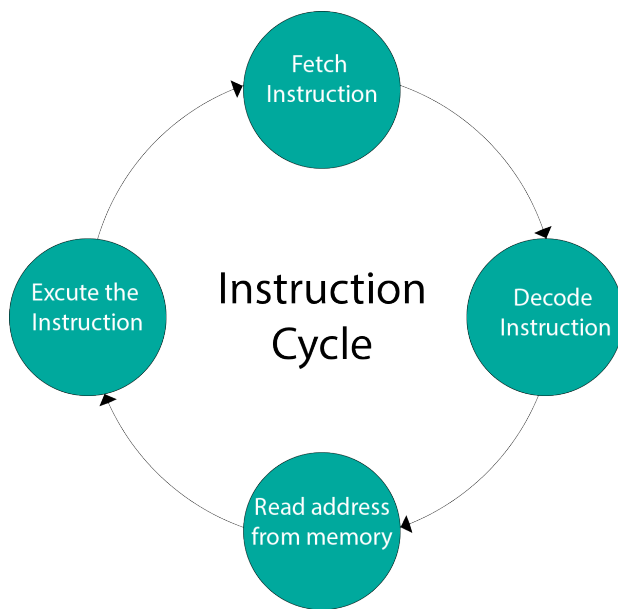
specifies a transfer of the content of PC into AR if timing signal  $T_0$  is active.  $T_0$  is active during an entire clock cycle interval. During this time the content of PC is placed onto the bus (with  $S_2S_1S_0 = 010$ ) and the LD (load) input of AR is enabled. The actual transfer does not occur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

## Instruction Cycle

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.

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

1. Fetch instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory.
4. Execute the instruction.



<https://www.javatpoint.com/instruction-cycle>

