UNIT -1

Functional blocks of a computer:

A computer consists of five functionally independent main parts:

1. input
2. memory
3. arithmetic and logic
4. output
5. control unit

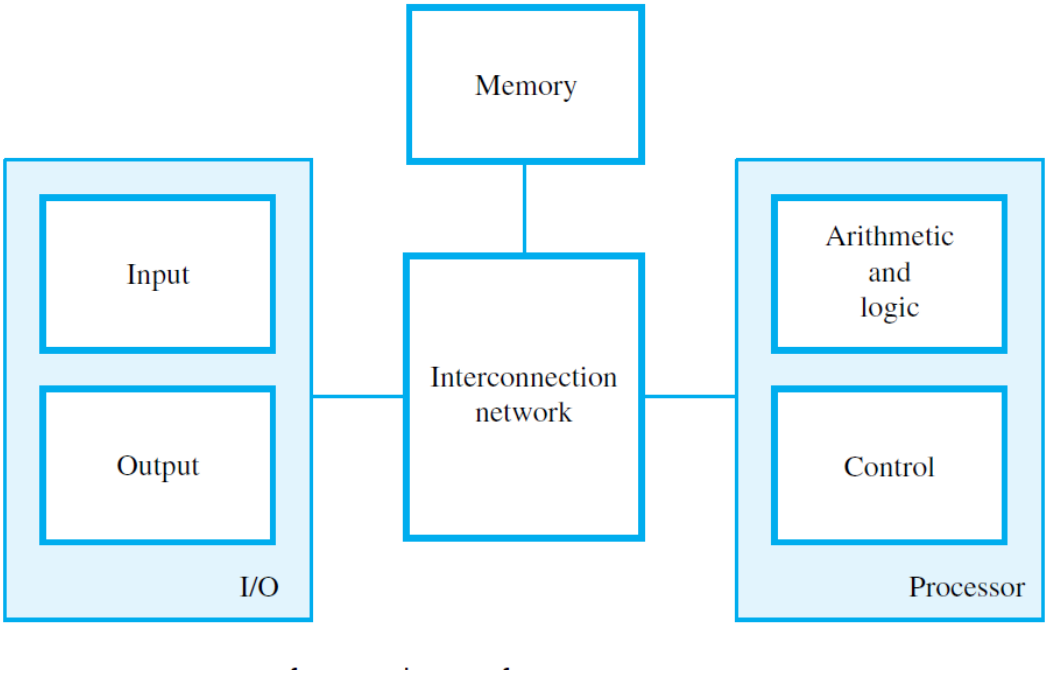


Fig:Basic functional units of a computer.

1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines.
2. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit.
3. The processing steps are specified by a program that is also stored in the memory.
4. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit.
5. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.

Aprogram is a list of instructions which performs a task. Programs are stored in the memory.The processor fetches the program instructions from the memory, one after another, and perform the desired operations. The computer is controlled by the stored program, exceptfor possible external interruption by an operator or by I/O devices.The instructions and data handled by a computer is encoded as a string of binary bits.

- **Input Unit**: Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.

Eg: touchpad, mouse, joystick

- **Memory Unit:**

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory: Primary memory, also called main memory, is a fast memory that operates at electronicspeeds. Programs must be stored in this memory while they are being executed.The memory consists of a large number of semiconductor storage cells, each capable of storingone bit of information.The memory is organized so that one word canbe stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits.To provide easy access to any word in the memory, a distinct address is associatedwith each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing acontrol command to the memory that starts the storage or retrieval process.Instructions and data can be written into or read from the memory under the control ofthe processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The timerequired to access one word is called the memory access time. This time is independent ofthe location of the word being accessed.

Cache Memory: As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates. At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an

instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Secondary Storage: Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondarystorage is used when large amounts of data and many programs have to be stored, particularlyfor information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. Awide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit: Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use. When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data.

Output Unit: The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a printer. Some units, such as graphic displays, provide both an output function, showing textand graphics, and an input function, through touchscreen capability.

Control Unit: The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states. Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place. In practice, much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

Basic Operational Concepts

To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.A typical instruction might be

Load R2, LOC

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.The original contents of location LOC are preserved, whereas those of register R2 are overwritten.Execution of this instruction requires several steps.

1. . First, the instruction is fetched from the memory into the processor.
2. Next, the operation to be performed is determined by the control unit.
3. The operand at LOC is then fetched from the memory into the processor.
4. Finally, the operand is stored in register R2.

Let us consider another example

Add R4, R2, R3

This instruction adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum. After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location and asserting the appropriate control signals.The data are then transferred to or from the memory.Figure 1.1 shows how the memory and the processor can be connected.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.

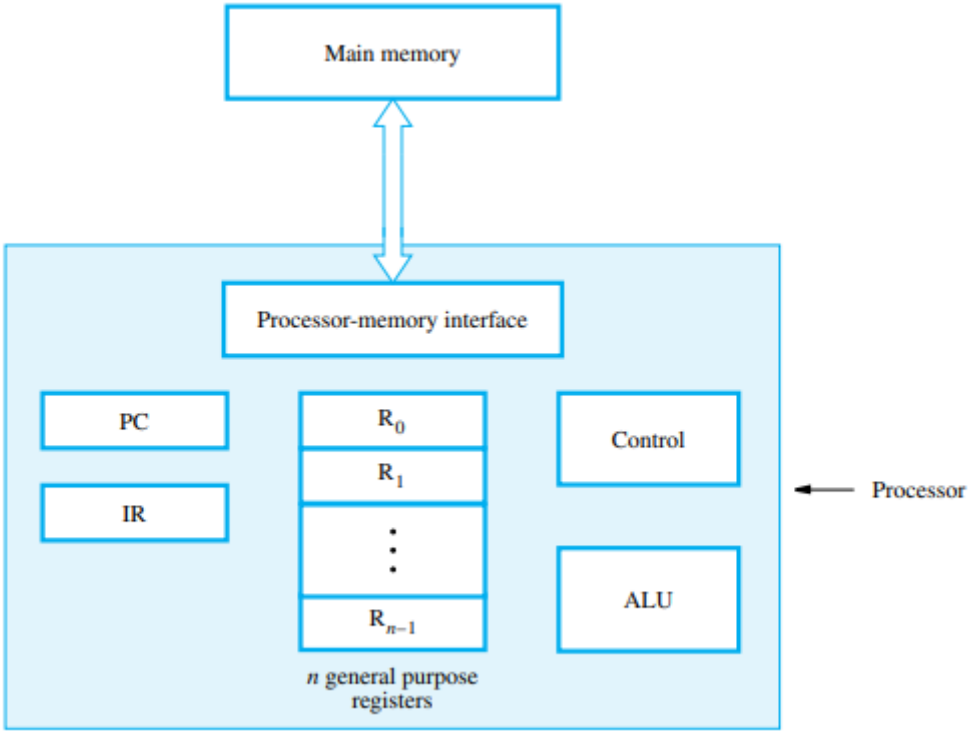


Fig 1.1: ***Connection between the processor and the main memory***

PC: The program counter (PC) is another specialized register.contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed.

General purpose Registers: There are also general-purpose registers R0 through Rn−1, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

Processor Memory Interface: The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Following are typical operating steps:

1) A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage
2) Execution of the program begins when the PC is set to point to the first instruction of the program.
3) The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the

memory it is loaded into register IR. At this point, the instruction is ready to be decoded and executed.

4) If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register "**R**".
5) After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register.
6) If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

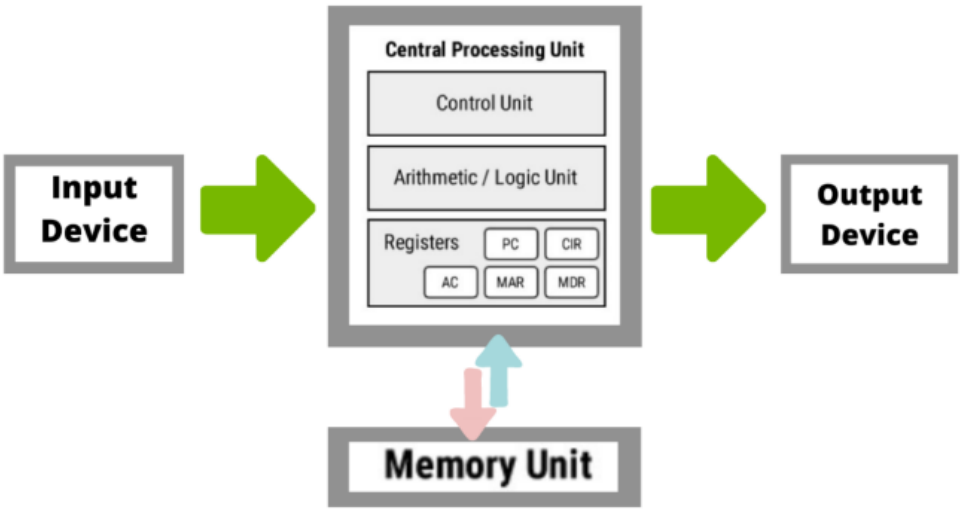
At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an interrupt-service routine. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

Von Neumann Architecture:

The Von-Neumann Architecture or Von-Neumann model is also known as **"Princeton Architecture"**. This architecture was published by the Mathematician **John Von Neumann** in **1945**.

Von Neumann architecture is the design upon which many general purpose computers are based. This architecture implemented the stored program concept in which the data and instructions are stored in the same memory. This architecture consists of a CPU(ALU, Registers, Control Unit), Memory and I/O unit.



Following are the components of Von Neumann Architecture:

1. CPU(Central processing unit)
   - CU(Control Unit)
   - ALU(Arithmetic and logic unit)
   - Registers
     - ✓ PC(Program Counter)
     - ✓ IR(Instruction Register)
     - ✓ AC(Accumulator)
     - ✓ MAR(Memory Address Register)
     - ✓ MDR(Memory Data Register)
2. BUSES
3. I/o Devices
4. Memory Unit

1. CPU: CPU acts as the brain of the computer and is responsible for the execution of instructions.

a) **Control Unit:** A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions, and controls how data moves around the system.

b) **Arithmetic and Logic Unit (ALU) :**
The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic operations.

c) **Registers**: A processor based on von Neumann architecture has five special registers which it uses for processing:

- **Program counter (PC)** holds the memory address of the next instruction to be fetched from primary storage.
- The **Memory Address Register(MAR)** holds the address of the current instruction that is to be fetched from memory, or the address in memory to which data is to be transferred.
- The **Memory Data Register(MDR)** holds the contents found at the address held in MAR or data which is to be transferred to the primary storage.
- The **Current Instruction Register(CIR)** holds the instruction that is currently being decoded and executed.
- The **Accumulator** is a special purpose Register and is used by the ALU to hold the data being processed and the results of calculations.

2.BUSES :A bus is a subsystem that is used to connect computer components and transfer data between them. There are three types of BUSES

a) **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
b) **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
c) **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

3**. I/o Devices**:Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer. If some results are evaluated by CPU and it is stored in the computer, then with the help of output devices, we can present them to the user.

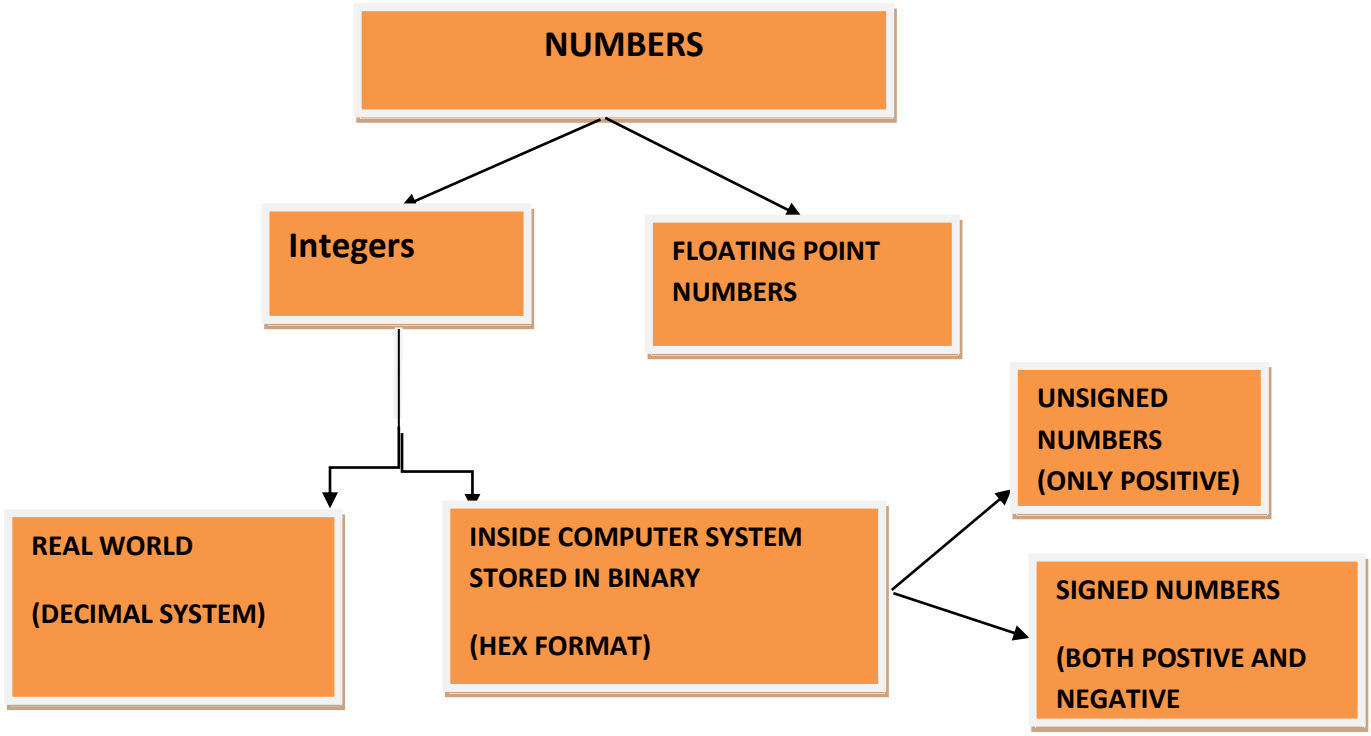
4. **Memory:**A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word(2^Mx N, eg: 128KB).

There are two types of Primary Memory:

1)RAM: VOLATILE MEMORY or temporary Memory(to store the program in execution)

2)ROM: NON-VOLATILE MEMORY or permanent Memory(to store the booting program)

NUMBER REPRESENTATION:



UNSIGNED INTEGERS

These are binary numbers that are always assumed to be positive.Here all available bits of the number are used to represent the magnitude of the number.No bits are used to indicate itssign, hence they are called unsigned numbers.

E.g.: Roll Numbers, Memory addresses etc

SIGNED INTEGERS

These are binary numbers that can be either positive or negative. The MSB of the number indicates whether it is positive or negative. If **MSB is 0 then the number is Positive**. If **MSB is 1 then the number is Negative**. Negative numbers are always stored in 2's complement form.

Three systems are used forrepresenting such numbers:

- **Signed magnitude**
- **1's-complement**
- **2's-complement**

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.Positive values have identical representations in all systems, but negative values have different representations.

In the **signed magnitude system**, negative values are represented by changing the mostsignificant bit from 0 to 1.For example, +5 is represented by 0101, and −5 is represented by 1101.

In **1's-complement representation**, negative values are obtained by complementing eachbit of the corresponding positive number. Thus, the representation for −3 is obtainedby complementing each bit in the vector 0011 to yield 1100.The same operation, bitcomplementing, is done to convert a negative number to the corresponding positive value.

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Fig: Binary signed number Representations

Two's complement gives a unique representation for zero. Any other system gives a separate representation for +0 and for -0. This is absurd. In two's complement system, -(x) is stored as two's complement of (x). Applying the same rule for 0, -(0) should be stored as two's complement of 0. 0 is stored as 000. So –(0) should be stored as two's complement of 000, which again is 000. Hence two's complement gives a unique representation for 0.**It produces an additional number on the negative side.** As two's complement system produces a unique combination for 0, it has a spare combination '1000' in the above case, and can be used to represent –(8).

3 Bit Integer	
2^3 = 8 therefore 8 combinations	
Unsigned	Signed
0 ... 7	-4 ... -1 0 1 ... 3

4 Bit Integer	
2^4 = 16 therefore 16 combinations	
Unsigned	Signed
0 ... 15	-8 ... -1 0 1 ... 7

Fixed and Floating point Representations:

There are two major approaches to store real numbers (i.e., numbers withfractional component) in modern computing. These are

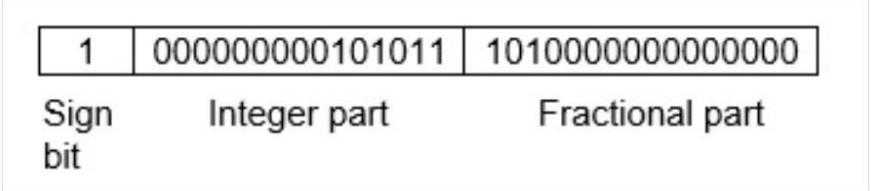
(i) Fixed Point Notation and

(ii) Floating Point Notation.

Fixed Point Notation: In **fixed point notation**, there are a fixed number of digits after the decimal point, whereas **floating point number** allows for avarying number of digits after the decimal point.

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part. Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent -. 000000000101011 is 15-bit binary value for decimal 43 and 1010000000000000 is 16-bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.

Floating Point Representation:

In some numbers, which have a fractional part, the position of the decimal point is not fixed as the number of bits before (or after) the decimal point may vary. **Eg: 0010.01001, 0.0001101, -1001001.01** etc. the position of the decimal point is not fixed, instead it**"floats"** in the number.Such numbers are called Floating Point Numbers. Floating Point Numbers are stored in a "Normalized" form.

NORMALIZATION OF A FLOATING POINTNUMBER:

Normalization is the process of shifting the point, left or right, so that there is only one non-zero digit to the left of the point.

01010.01 $(-1)^{\boldsymbol{0}} \times 1.\boldsymbol{01001} \times 2^3$

11111.01 $(-1)^{\boldsymbol{0}} \times 1.\boldsymbol{111101} \times 2^4$

-10.01 $(-1)^{\boldsymbol{1}} \times 1.\boldsymbol{001} \times 2^1$

A normalized form of a number is:

$$\mathbf{-1^s \ x1.MX2^E}$$

Where: S = Sign, M = Mantissa and E = Exponent.

As Normalized numbers are of the 1.M format, the "1" is not actually stored, it is instead assumed. Also the Exponent is stored in the biased form by adding an appropriate bias value to it so that -ve exponents can be easily represented.

Advantages of Normalization.

1. Storing all numbers in a standard for makes **calculations easier** and **faster**.
2. By **not storing** the **1** (of 1.M format) for a number, considerable **storage space** is **saved**.
3. The **exponent** is **biased** so there is **no need** for **storing** its **sign bit** (as the biased exponent cannot be -ve).

SHORT REAL FORMAT / SINGLE PRECISION FORMAT / IEEE 754: 32 BIT FORMAT:

S	Biased Exponent	Mantissa
(1)	(8) Bias value = 127	(23 bits)

1. **32 bits** are used to store the **number**.
2. **23 bits** are used for the **Mantissa**.
3. **8 bits** are used for the Biased **Exponent**.
4. **1 bit** used for the **Sign** of the number.
5. The **Bias** value is $(\mathbf{127})_{10}$.

Range: $\pm 1 \times 10^{-38}$ to $\pm 3 \times 10^{38}$

LONG REAL FORMAT / DOUBLE PRECISION FORMAT / IEEE 754: 64 BIT FORMAT

1. **64 bits** are used to store the **number**.
2. **52 bits** are used for the **Mantissa**.
3. **11 bits** are used for the Biased **Exponent**.
4. **1 bit** used for the **Sign** of the number.
5. The **Bias** value is $(\mathbf{1023})_{10}$.
6. The range is $\mathbf{+10^{-308}}$ **to** $\mathbf{+10^{308}}$approximately.

s	Biased Exponent	Mantissa
1 bit	11-bits (Bias value:1023)	52-bits

Extreme cases of floating point numbers:

Floating point numbers are represented in IEEE formats.Consider IEEE 754 32-bit format also called Single Precision format or Short real format.

Overflow:

For a value, 1.0 the normalized form will be

$(-1)^0 \times 1.0 \times 2^0$

Herethe True Exponent is 0.

```
If:  TE = 0,     BE = 127    Representation = 0111 1111
If:  TE = 1,     BE = 128    Representation = 1000 0000
If:  TE = 2,     BE = 129    Representation = 1000 0000
...
If:  TE = 127,   BE = 254    Representation = 1111 1110
If:  TE = 128,   BE = 255    Representation = 1111 1111
If:  TE = 129,   BE = 255    Representation = 1111 1111
If:  TE = 130,   BE = 255    Representation = 1111 1111
```

This is because the 8-bit biased exponent cannot hold a value more than 255.Hence, all cases where the TE = 128 or more, the ***BE will be represented as 1111 1111.This indicates as exception (error) called OVERFLOW. The number is called NaN (Not a Number).*** It is identified by Exponent being all 1s (1111

1111).Here, the Mantissa can be anything!The **Extreme case of NaN is Infinity**.It is also an OVERFLOW and hence the Exponent will be 1111 1111.To differentiate Infinity from NaN, the Mantissa in infinity is 0000 0000.Hence **Infinity is identified as Exponent all 1s and Mantissa all 0s.**

Suppose the number is 0.1.It will be normalized as

$(-1)^0 \times 1.0 \times 2^{-1}$

The true exponent here is -1.

```
If:  TE = -1,    BE = 126    Representation = 0111 1110
If:  TE = -2,    BE = 125    Representation = 0111 1101
…
If:  TE = -126,  BE = 1      Representation = 0000 0001
If:  TE = -127,  BE = 0      Representation = 0000 0000
If:  TE = -128,  BE = 0      Representation = 0000 0000
If:  TE = -129,  BE = 0      Representation = 0000 0000
```

Underflow: All cases where the TE = -127 or less, the BE will be represented as 0000 0000.This indicates as exception (error) called UNDERFLOW.

The number is called De-Normal Number.It is identified by Exponent being all 0s (0000 0000).Here, the Mantissa can be anything.The **Extreme case of De-Normal Number is Zero.**

It is also an UNDERFLOW and hence the Exponent will be 0000 0000.To differentiate Zero from De-Normal Number, the Mantissa in Zero is 0000 0000.Hence **Zero is identified as Exponent all 0s and Mantissa all 0s**.This means Zero is represented as all 0s.

Example:Convert 2A3BH into Short Real format.

Soln: Converting the number into binary we get:

0010 1010 0011 1011

Normalizing the number we get:

$(-1)^0$**x** 1.**0101000111011** x 2^{13}

Here S = 0; M = 0101000111011; True Exponent = 13.

Bias value for Short Real format is 127:

Biased Exponent (BE) = True Exponent + Bias

= 13 + 127

= 140.

Converting the Biased exponent into binary we get:

Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**01010001110110 0…**

S Biased Exp Mantissa

(1) (8) (23)

Computer Arithmetic

Integer Addition:

Addition of Unsigned Integers: Addition of 1-bit numbers is illustrated below.The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the sum is 0 and the carry-out is 1. In order to add multiple-bit numbers, We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the carry-in to the next bit pair to the left. The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1, which represents the value 3.

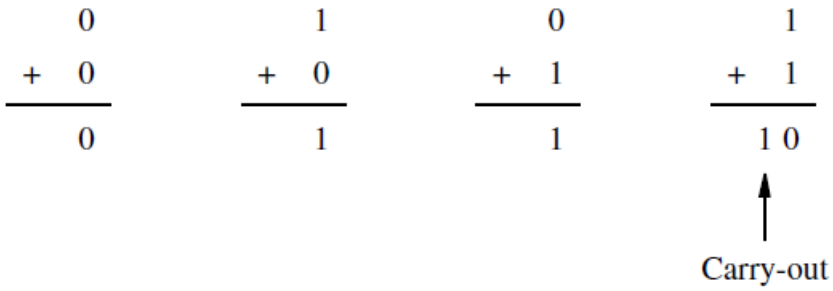


Fig: Addition of 1-bit Numbers

Addition and Subtraction of Signed Integers:

- To add two numbers, add their n-bit representations, ignoring the carry-out bit fromthe most significant bit (MSB) position. The sum will be the algebraically correct value in2's-complement representation if the actual result is in the range$-(2^{n-1})$ through$+2^{n-1}-1$.
- To subtract two numbers X and Y, that is, to perform X − Y , form the 2's-complement of Y , then add it to X using the add rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range $-(2^{n-1})$ through$+2^{n-1}$.

X-Y = X+(-Y) = X+(2'S Complement of Y)

Example: **To perform 7-3 using 2's complement addition**

```
    0 1 1 1
+   1 1 0 1
-------------
1   0 1 0 0
↑
Carry-out
```

If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer.

Few more examples:

(a)	0 0 1 0	(+2)	(b)	0 1 0 0	(+4)
	+ 0 0 1 1	(+3)		+ 1 0 1 0	(–6)
	0 1 0 1	(+5)		1 1 1 0	(–2)
(c)	1 0 1 1	(–5)	(d)	0 1 1 1	(+7)
	+ 1 1 1 0	(–2)		+ 1 1 0 1	(–3)
	1 0 0 1	(–7)		0 1 0 0	(+4)
(e)	1 1 0 1	(–3)	⇒	1 1 0 1	
	– 1 0 0 1	(–7)		+ 0 1 1 1	
				0 1 0 0	(+4)

Sign Extension: We often need to represent a value given in a certain number of bits by using a larger number of bits. For a positive number, this is achieved by adding 0s to the left. For a negative number in 2's-complement representation, the leftmost bit, which indicates the sign of the number, is a 1. A longer number with the same value is obtained by replicating the sign bit to the left as many times as needed.

Overflow in Integer Arithmetic: Using 2's-complement representation, n bits can represent values in the range $-(2^{n-1})$ through$+2^{n-1}$.For example, the range of numbers that can be represented by 4 bits is −8through +7.When the actualresult of an arithmetic operation isoutside the representable range, an arithmetic overflow has occurred.

Introduction to adder circuits:

ONE BIT ADDITION: FULL ADDER

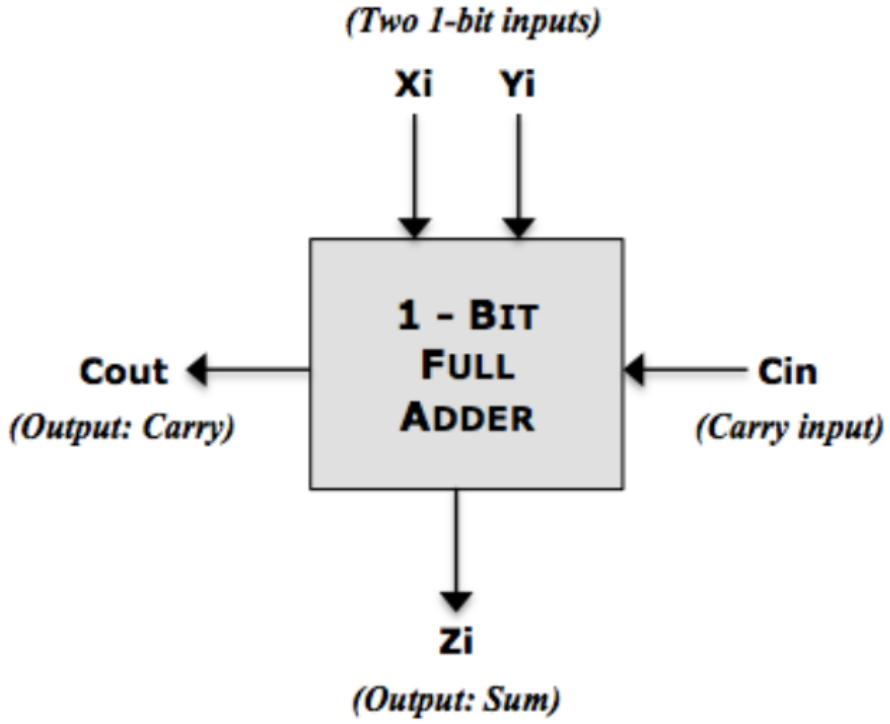
1) It is a 1-bit adder circuit.

2) It adds two 1-bit inputs Xi & Yi, along with a Carry Input Cin.

3) It produces a sum Zi and a Carry output Cout.

4) As it considers a carry input, it can be used in combination to add large numbers.

5) Hence it is called a Full Adder.



Inputs bits: Xi and Yi.
Input Carry: Cin

Output (Sum): Zi
Output (Carry): Cout

Formula for Sum (Zi)

$$Zi = Xi \oplus Yi \oplus Cin$$
$$\therefore Zi = Xi \cdot Yi \cdot Cin + Xi \cdot Yi \cdot \overline{Cin} + Xi \cdot \overline{Yi} \cdot Cin + \overline{Xi} \cdot Yi \cdot Cin$$

Formula for Carry (Cout)

$$Cout = Xi \cdot Yi + Xi \cdot Cin + Yi \cdot Cin$$

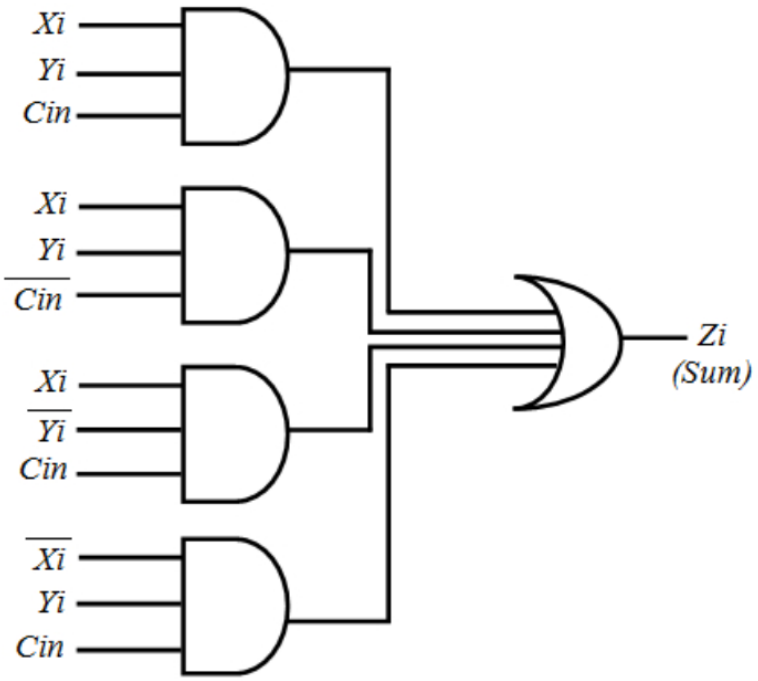


Fig: Circuit for Sum

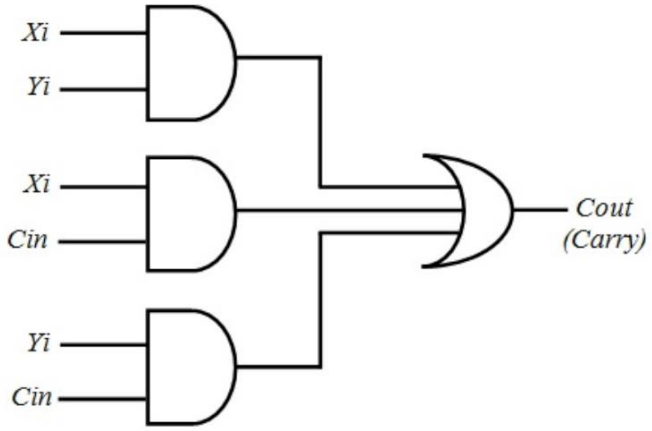


Fig: Circuit for carry

RIPPLE CARRY ADDER(For Multiple bit addition):

1)A Full Adder can add two "1-bit" numbers with a Carry input.

2) It produces a "1-bit" Sum and a Carry output.

3) Combining many of these Full Adders, we can add multiple bits.

4) One such method is called Serial Adder.

5) Here, bits are added one-by-one from Least significant bit(LSB) onwards.

6) The carries are connected in a chain through the full adders. The Carry of each stage is propagated (Rippled) into the next stage.

7) Hence, these adders are also called Ripple Carry Adders.

Advantage: They are very easy to construct.

Drawback: As addition happens bit-by-bit, they are slow.

8) Number of cycles needed for the addition is equal to the number of bits to be added.

Inputs:

Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.

X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB)

Y = Y0 Y1 Y2 Y3 (Y0 is the MSB … Y3 is the LSB)

CIN = Carry Input

Outputs:

Assume Z to be a "4-bit" output, and COUT to be the output Carry

Z = Z0 Z1 Z2 Z3 (Z0 is the MSB … Z3 is the LSB)(Here Z represents the sum)

COUT = Carry Output

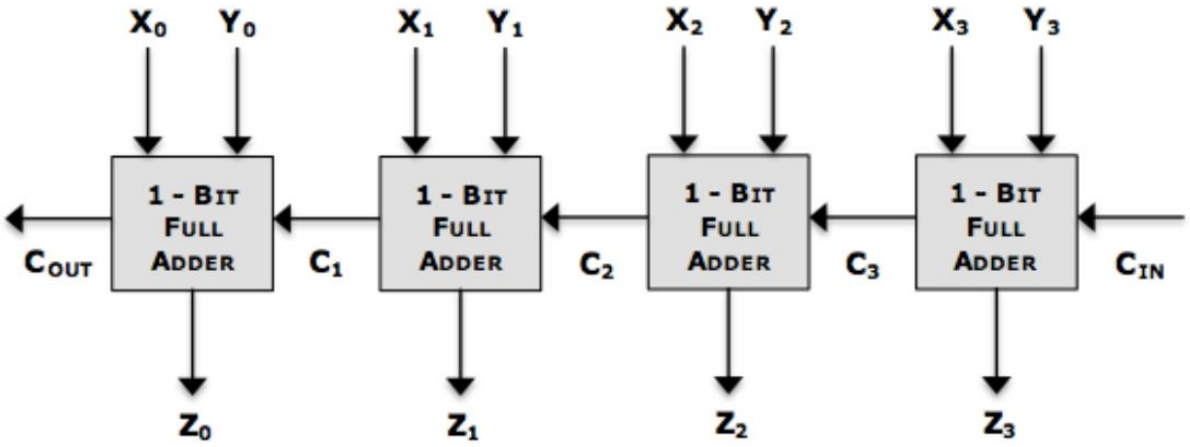


Fig:4-bit Ripple Carry Adder

<u>Carry Look ahead Adder(For multiple bit Addition):</u>

1) This is also called as parallel adder. It is used to add multiple bits simultaneously.
2) While adding multiple bits, the main issue is that of the intermediate carries.
3) In Serial Adders, we therefore added the bits one-by-one.
4) This allowed the carry at any stage to propagate to the next stage.
5) But this also made the process very slow.
6) If we "PREDICT" the intermediate carries, then all bits can be added simultaneously.
7) This is done by the Carry Look Ahead Generator Circuit.
8) Once all carries are determined beforehand, then all bits can be added simultaneously.

Advantage: This makes the addition process extremely fast.

Drawback: Circuit is complex.

Inputs:

Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.

X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB); Y = Y0 Y1 Y2 Y3 & CIN = Carry Input

Outputs:

Assume Z to be a "4-bit" output, and C_{OUT} to be the output Carry

Z = Z0 Z1 Z2 Z3 & C_{OUT} = Carry Output

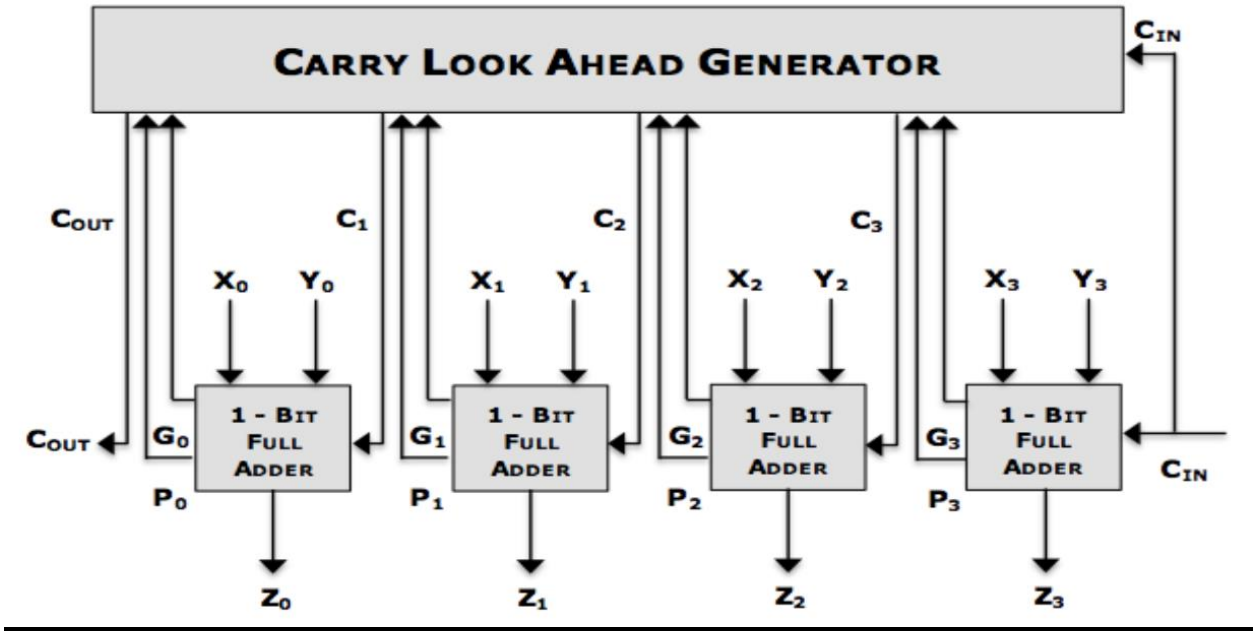


Fig: Circuit for Carry Look ahead Adder

We can "Predict" (Look Ahead) all the intermediate carries in the followingmanner:

The carry at any stage can be calculated as:

$$C_i = X_i.Y_i + X_i.C_{in} + Y_i.C_{in}$$
$$C_i = X_i.Y_i + C_{in}(X_i + Y_i)$$

This implies $C_i = G_i + P_i.C_{IN}$

Here $Gi = X_i.Y_i$ ... (Generate)
And $Pi = X_i + Y_i$ ... (Propagate)

We need to predict the Carries: C3, C2, C1 and C0

C3 = G3 + P3CIN (I)

C2 = G2 + P2C3

Substituting the value of C3, we get:

C2 = G2 + P2G3 + P2P3CIN (II)

C1 = G1 + P1C2

Substituting the value of C2, we get:

C1 = G1 + P1G2 + P1P2G3 + P1P2P3CIN (III)

C0 = G0 + P0C1

Substituting the value of C1, we get:

C0 = G0 + P0G1 + P0P1G2 + P0P1P2G3 + P0P1P2P3CIN (IV)

From the above four equations, it is clear that the values of all the four Carries (C3, C2, C1, C0) can be determined beforehand even without doing the respective additions. To do this we need the values of all G's (Xi.Yi) and all P's (Xi+Yi) and the original carry input CIN. This is done by the Carry Look Ahead Generator Circuit.

Cycle 1: g_1, p_1, g_2, p_2, g_3, p_3, g_0, $p_0$are given to the carry look ahead generator.

Cycle 2: Input carries are given to the adders by the carry generator.

Cycle 3: Results are produced.

Total number of cycles required :3

Multiplication:

1) Shift and Add: This method is used to multiply two unsigned numbers. When we multiply two "N-bit" numbers, the answer is "2 x N" bits. Three registers A, Q and M, are used for this process. Q contains the Multiplier and M contains the Multiplicand. A (Accumulator) is initialized with 0. At the end of the operation, the Result will be stored in (A & Q) combined. The process involves addition and shifting. That is why it is called shift and add method.

Algorithm:

The **number of steps** required is equal to the **number of bits in the multiplier**.

1) At each step, **examine** the current **multiplier bit** starting from the **LSB**.

2) If the current **multiplier bit is "1"**, then the **Partial-Product** is the **Multiplicand** itself.

3) If the current **multiplier bit is "0"**, then the **Partial-Product** is the **Zero**.

4) At each step, **ADD the Partial-Product to the Accumulator**.

5) Now **Right-Shift the Result** produced so far (**A & Q combined**).

Repeat steps 1 to 5 for **all bits** of the multiplier.

The **final answer** will be in **A & Q** combined.

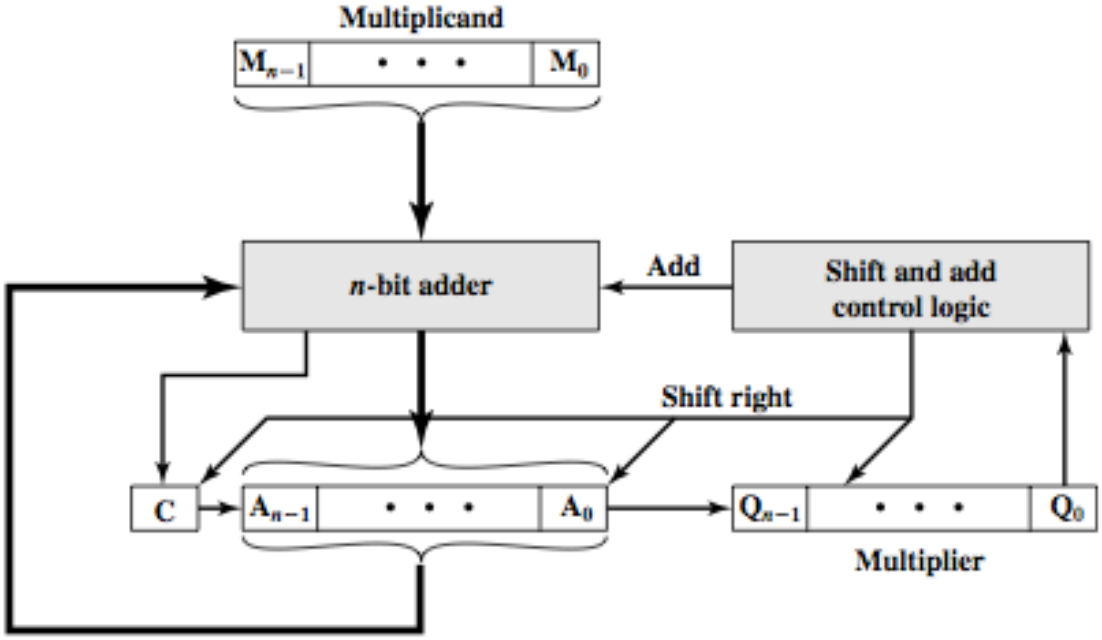


Fig: Shift and Add Multiplication

Example: Let us consider 7X6

```
                    0  1  1  1   … Multiplicand (7)
              X     0  1  1  0   … Multiplier (6)
-------------------------------------------------
                    0  0  0  0   … Partial-Product
                 0  1  1  1  X              ”
              0  1  1  1  X  X              ”
+  0  0  0  0  X  X  X              ”
-------------------------------------------------
   0  1  0  1  0  1  0   … Result (42)
-------------------------------------------------
```

Step	**C** **Carry**	**A** **Accumulator**	**Q** **Multiplier**	**M** **Multiplicand**	**Explanation**
	0	0000	0110	0111	*Initial Value*
1	**0** **0**	0000 0000	0110 0011		*Current Multiplier bit is “0” so ADD “0” to Accumulator and Right-Shift*

2	0 0	0111 0011	0011 1001		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
3	0 0	1010 0101	1001 0100		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
4	0 0	0101 ***0010***	0100 ***1010***		*Current Multiplier bit is "0" so ADD "0" to Accumulator and Right-Shift*

2) Booth Multiplier(For signed Multiplication):

Booth's Algorithm is used to **multiply two SIGNED numbers**. When we multiply two **"N-bit"** numbers, the answer is "**2 x N**" bits. Three registers A, Q and M, are used for this process.**Q** contains the **Multiplier** and **M** contains the **Multiplicand**.**A** (**Accumulator**) is initialized with 0.At the end of the operation, the **Result** will be stored in (**A & Q**) combined.The process involves **addition, subtraction** and **shifting**.

Algorithm:

The **number of steps** required is equal to the **number of bits in the multiplier**.

At the beginning, consider an **imaginary "0" beyond LSB of Multiplier**

1) At each step, **examine two adjacent Multiplier bits** from **Right to Left**.

2) If the transition is from **"0 to 1"** then **Subtract M** from **A** and **Right-Shift** (A & Q) combined.

3) If the transition is from **"1 to 0"** then **ADD M** to **A** and **Right-Shift**.

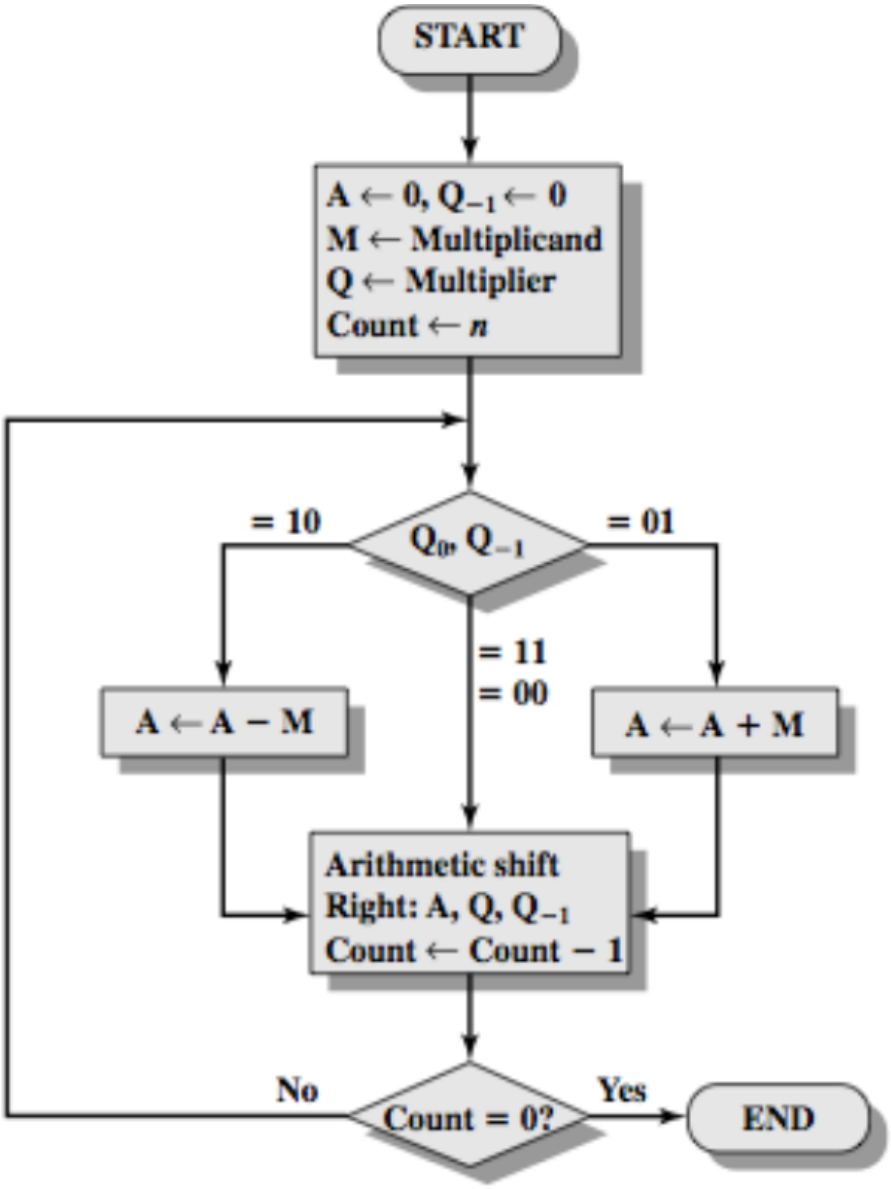
4) If the transition is from **"0 to 0"** then **simply Right-Shift**.

5) If the transition is from **"1 to 1"** then **simply Right-Shift**.

Repeat steps 1 to 5 for **all bits** of the multiplier.

The **final answer** will be in **A & Q** combined.

Flowchart for Booth's Algorithm:



Example: -9x10=-90

Multiplicand (M): **-9 = 10111** **9 = 01001**. (Two's Complement Form)

Multiplier (Q): **10 = 01010**. **-10 = 10110** (Two's Complement Form)

step	**A** **Accumulator**	**Q** **Multiplier**	**Q(-1)**	**M** **Multiplicand**
Initial	**00000**	**01010**	**0**	**10111**
1) (0 ç 0) No Add or Sub Right-Shift	**00000** **00000**	**01010** **00101**	**0** **0**	
2) (1 ç 0)	**01001**	**00101**	**0**	

Perform **(A - M)** Right-Shift	**00100**	**10010**	**1**	
3) (0 ç 1) Perform **(A + M)** Right-Shift	**11011** **11101**	**10010** **11001**	**1** **0**	
4) (1 ç 0) Perform **(A - M)** Right-Shift	**00110** **00011**	**11001** **01100**	**0** **1**	
5) (0 ç 1) Perform **(A + M)** Right-Shift	**11010** **11101**	**01100** **00110**	**1** **0**	

Restoring and Non-Restoring Division:

Non Restoring Division:

1) Let Q register hold the divided, M register holds the divisor and A register is 0.

2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.

1) At each step, left shift the dividend by 1 position.

2) Subtract the divisor from A (perform A - M).

3) If the result is positive then the step is said to be “Successful”. In this case quotient bit will be “1” and Restoration is NOT Required. The Next Step will also be Subtraction.

4) If the result is negative then the step is said to be “Unsuccessful”. In this case quotient bit will be “0”. Here Restoration is NOT Performed. Instead, the next step will be ADDITION in place of subtraction.

As restoration is not performed, the method is called Non-Restoring Division.

Repeat steps 1 to 4 for all bits of the Dividend.

Example: (7) / (5)

Dividend (Q) = 7

Divisor (M) = 5

Accumulator (A) = 0

7 = 0111 **5** = 0101

-7 = 1001-**5** = 1011

	Accumulator A(0)	Dividend Q(7)	Divisor M(5)
Initial Values	**0000**	**0111**	**0101**
Step 1: Left shift A-M Unsuccessful(-ve) Next step: Add	0000 +1011 **1011**	**111_** **1110**	
Step 2: Left shift A+M Unsuccessful(-ve) Next step: Add	0111 +0101 **1100**	**110_** **1100**	
Step 3: Left shift A+M Unsuccessful(-ve) Next step: Add	1001 +0101 **1110**	**100_** **1000**	
Step 4: Left shift A+M successful(+ve)	1101 +0101 **0010**	**000_** **0001**	
	Remainder:2	**Quotient:1**	

RESTORING DIVISION (For unsigned Numbers)

1) Let Q register hold the divided, M register holds the divisor and A register is 0.

2) On completion of the algorithm, Q will get the quotient and A will get theremainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.

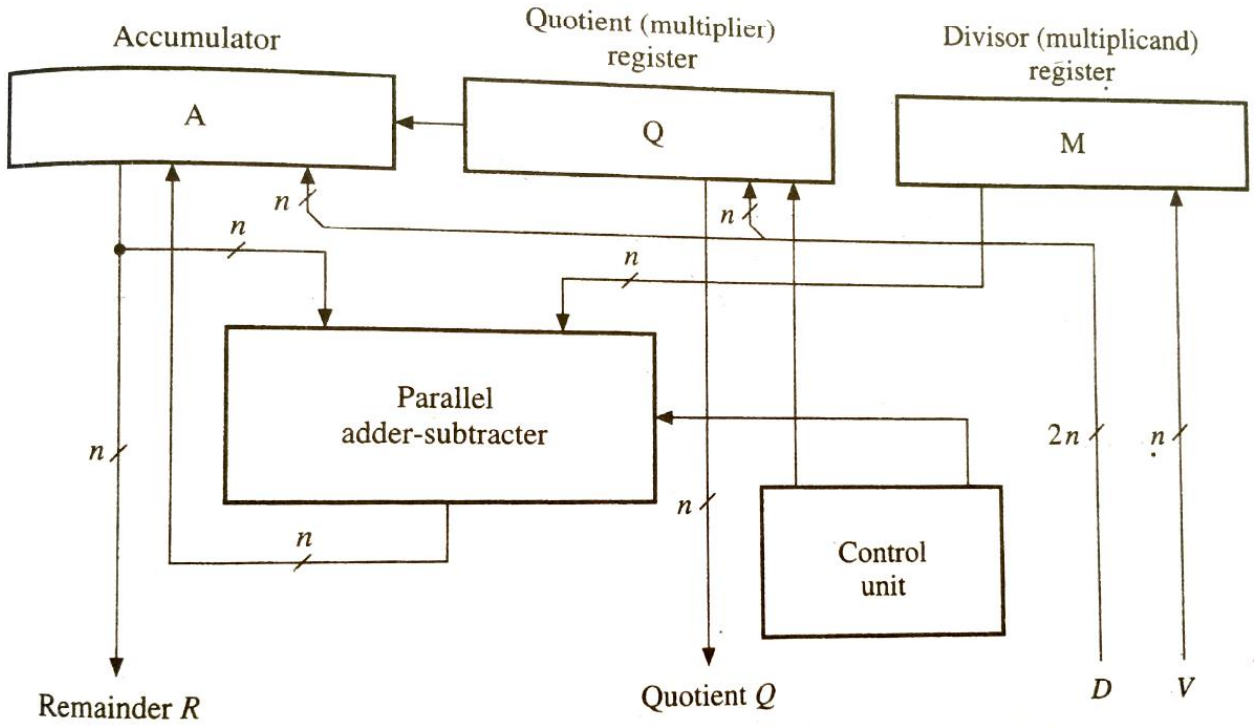
1) At each step, left shift the dividend by 1 position.

2) Subtract the divisor from A (perform A - M).

3) If the result is positive then the step is said to be “Successful”.In this case quotient bit will be “1” and Restoration is NOT Required.

4) If the result is negative then the step is said to be “Unsuccessful”.In this case quotient bit will be “0”.Here Restoration is performed by adding back the divisor.

Hence the method is called Restoring Division.Repeat steps 1 to 4 for all bits of the Dividend.



Example: (6) / (4)

Dividend (Q) = 6

Divisor (M) = 4

Accumulator (A) = 0

6 = 0110 4 = 0100

-6 = 1010 -4 = 1100

	Accumulator A(0)	**Dividend Q(6)**	**Divisor M(4)**
Initial Values	0000	0110	0100
Step 1: Left shift A-M Unsuccessful(-ve) Restoration:	0000 + 1100 **1100** 0000	110_ 1100	
Step 2: Left shift A-M Unsuccessful(-ve) Restoration:	0001 +1100 1101 0001	100_ 1000	
Step 3: Left shift A-M Unsuccessful(-ve) Restoration:	0011 +1100 1111 0011	000_ 0000	

Step 3: Left shift A-M Successful(+ve) No Restoration	0110 +1100 0010	000_ 0001	
	Remainder(2)	Quotient(1)	

RESTORING DIVISION FOR SIGNED NUMBERS:

1) Let M register hold the divisor, Q register hold the divided.

2) A register should be the signed extension of Q.

3) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.

1) At each step, left shift the dividend by 1 position.

2) If Sign of A and M is the same then Subtract the divisor from A (perform A - M),
Else Add M to A

3) After the operation,If Sign of A remains the same or the dividend (in A and Q) becomes zero,then the step is said to be "Successful".In this case quotient bit will be "1" and Restoration is NOT Required.

4) If Sign of A changes, then the step is said to be "Unsuccessful".In this case quotient bit will be "0".Here Restoration is Performed.Hence, the method is called Restoring Division.Repeat steps 1 to 4 for all bits of the Dividend.

Note: The result of this algorithm is such that, the quotient will always bepositive and the remainder will get the same sign as the dividend.

Example: (-19) / (7)

19 = 010011 7 = 000111

-19 = 101101 -7 = 111001

	Accumulator A(Sign Extension)	**Dividend Q(-19)**	**Divisor M(7)**
Initial Values	111111	101101	000111
Step 1: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111111 + 000111 000110 111111	01101_ 011010	
Step 2: Left-shift	111110	11010_	

Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	+ 000111 000101 111110	 110100	
Step 3: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111101 + 000111 000100 111101	10100_ 101000	
Step 4: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111011 + 000111 000010 111011	01000_ 010000	
Step 5: Left-shift Sign(A,M) Different: A+M Sign still same: Successful Restoration not required	110110 + 000111 111101 111101	10000_ 100001	
Step 6: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111011 + 000111 000010 111011	00001_ 000010	
	Remainder(-5)	Quotient(2)	

UNIT-2

Architecture of 8086:

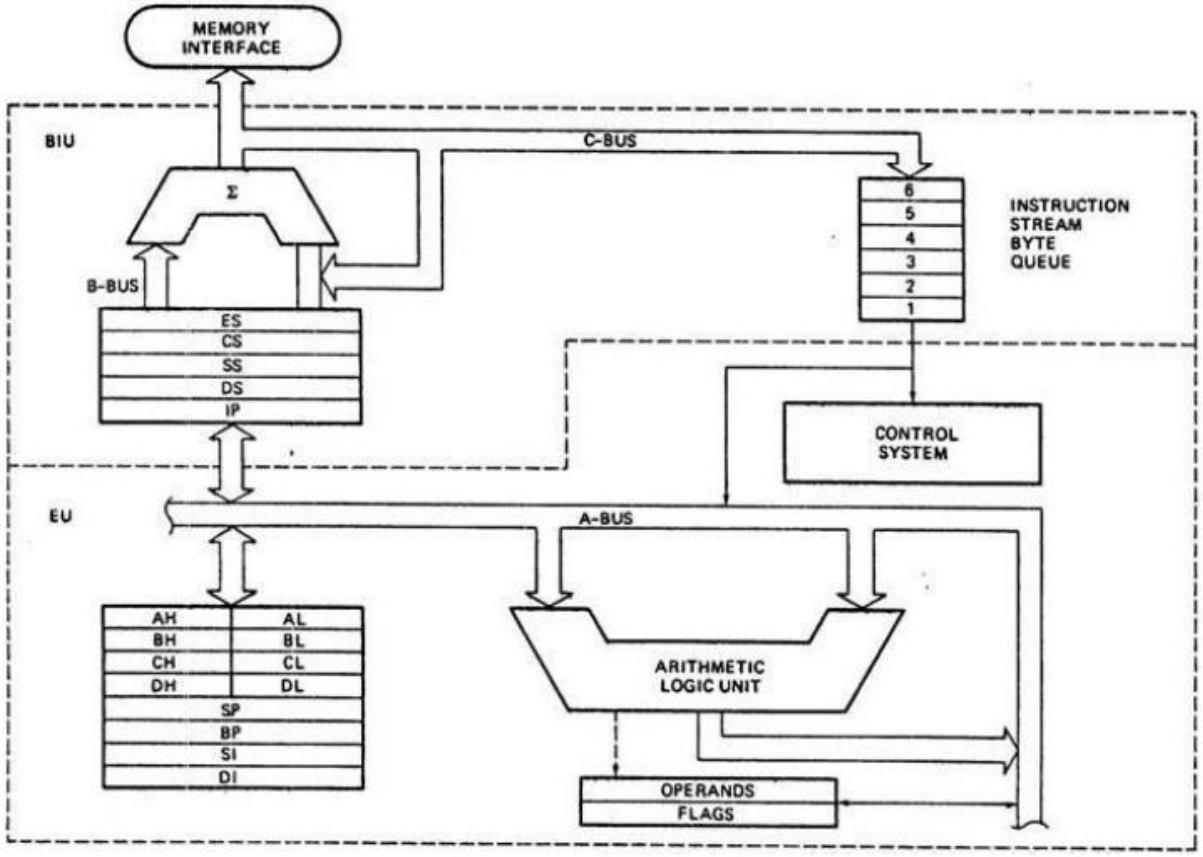


Fig: 8086 internal Architecture

The architecture of 8086 supports a 16-bit ALU , a set of 16-bit registers, and provides segmented memory addressing capability,fetched instruction queue for overlapped fetching and execution.

- Architecture of 8086 is pipeline type of architecture.
- The architecture of 8086 is divided into two functional parts i.e.,

i. **Execution unit (EU)**

ii. **Bus interface unit (BIU)**

These two units work asynchronously.

- Functional division of architecture speeds up the processing, since BIU and EU operate parallelly and independently i.e., EU executes the instructions and BIU fetches another instruction from the memory simultaneously.
- As the whole architecture is divided into two independent functional parts and both the subsystem's operations can be overlapped, hence the architecture is PIPELINING type of architecture.

EXECUTION UNIT

- The execution unit informs the BIU of the processor regarding from where to fetch the instructions from and then executes these instructions.
- The execution unit consists of the following:

- General purpose registers
- Stack pointer
- Base pointer
- Index registers
- ALU
- Flag register(FLAGS/ PSW)
- Instruction decoder
- Timing and control unit

Functions of EU

- Tells BIU regarding from where to fetch instructions or to read data.
- Receives opcode of an instruction from the queue.
- decodes the instructions.
- Executes the instruction.

Functions of various parts of EU

- Control circuitry: Directs internal operations.
- Instruction Decoder: Translates instructions fetched from memory into series of actions.
- ALU: Performs arithmetic and logical operations.
- FLAGS: Reflects the status of program.
- General purpose registers: Used to store Temporary data.
- Index and Pointer registers: Specifies/ informs about offset of operand

BUS INTERFACE UNIT

- The BIU handles transfer of data and address between the processor and memory/ I/O devices by computing address (Physical/ Effective address) and send the computed address to memory / I/O and fetches instruction codes then stores them in FIFO register set called Queue register.
- The BIU consists of the following:
  - Segment Registers
  - Instruction pointer
  - 6-Byte instruction Queue Register

Functions of BIU

- Handles transfer of data and address between processor and memory / I/O devices.
- Compute physical address and send it to memory interfaces.
- Fetches instruction codes and stores it in Queue
- Reads/Writes data from/to memory/ I/O devices

Functions of various parts of BIU

- **Segment registers :** Used to hold the starting address of the segment registers.
- **Queue register:** Used to store pre fetched instructions and inputs it to EU.
- **Instruction Pointer:** Used to point to the next instruction to be executed by EU.
- While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instruction bytes that will be following the present instruction from memory and stores them in the queue register simultaneously.

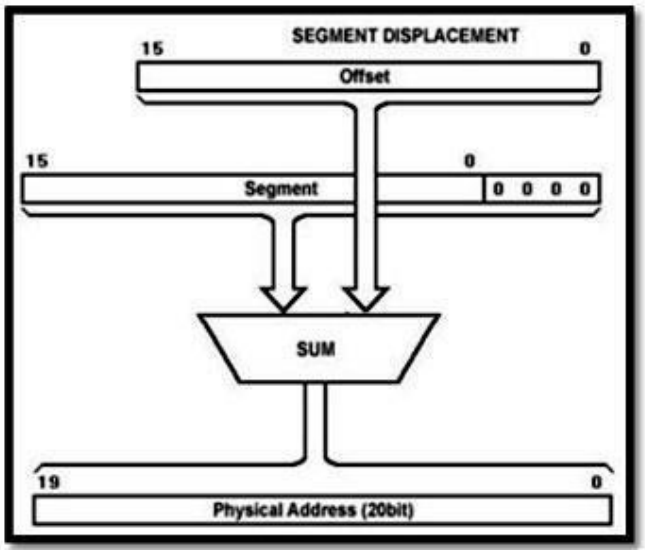
Logical and Physical Address

- Addresses within a segment can range from address 00000h to address 0FFFFh. This corresponds to the 64K-bytelength of the segment. An address within a segment is called an offset or logical address.
- A logical address gives the displacement from the base address of the segment to the desired location within it, as opposed to its "real" address, which maps directly anywhere into the 1 MByte memory space. This "real" address is called the physical address.

Difference between the physical and the logical address:

- The physical address is 20 bits long and corresponds to the actual binary code output by the BIU on the address bus lines. The logical address is an offset from location 0 of a given segment.

Segment address ⟶ 1005H
Offset address ⟶ 5555H
Segment address ⟶ 1005H ⟶ 0001 0000 0000 0101
Shifted by 4 bit positions ⟶ 0001 0000 0000 0101 0000
+
Offset address ⟶ 0101 0101 0101 0101
Physical address ⟶ 0001 0101 0101 1010 0101
1 5 5 A 5



a physical address while addressing memory. The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP, BP or an immediate 16-bit value, depending upon the addressing mode.

In case of 8085, once the opcode is fetched and decoded, the external bus remains free for some time, while the processor internally executes the instruction. This time slot is utilised in 8086 to achieve the over-lapped fetch and execution cycles. While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next instruction and arrange it in a queue known as predecoded instruction byte queue. It is a 6 bytes long, first-in first-out structure. The instructions from the queue are taken for decoding sequentially. Once a byte is decoded, the queue is rearranged by pushing it out and the queue status is checked for the possibility of the next opcode fetch cycle. While the opcode is fetched by the Bus Interface Unit (BIU), the Execution Unit (EU) executes the previously decoded instruction concurrently. The BIU along with the Execution Unit (EU) thus forms a pipeline. The bus interface unit, thus manages the complete interface of execution unit with memory and I/O devices, of course, under the control of the timing and control unit.

The execution unit contains the register set of 8086 except segment registers and IP. It has a 16-bit ALU, able to perform arithmetic and logic operations. The 16-bit flag register reflects the results of execution by the ALU. The decoding unit decodes the opcode bytes issued from the instruction byte queue. The timing and control unit derives the necessary control signals to execute the instruction opcode received from the queue, depending upon the information made available by the decoding circuit. The execution unit may pass the results to the bus interface unit for storing them in memory.

Flag register of 8086

There are total 9 flags in 8086 and the flag register is divided into two types:

(a) Status Flags – There are 6 flags in 8086 microprocessor which become set(1) or reset(0) depending upon condition after either 8-bit or 16-bit operation. These flags are conditional/status flags.

The 6 status flags are:

(b) Sign Flag (S): This flag is set when the result of any computation is negative.
(c) Zero Flag (Z): This flag is set when the result of any computation or comparison performed is zero.
(d) Auxiliary Cary Flag (AC): This flag is set when there is acarry from the lower nibble.
(e) Parity Flag (P): This flag is set when the lower byte of the result contains even number of 1's .
(f) Overflow Flag: This flag will be set (1) if the result of a signed operation is too large to fit in the number of bits available to represent it, otherwise reset (0).(eg:50+32= 82)
(g) Carry Flag (CY)): This flag is set when there is a carry out of the MSB in case of addition or a borrow in case of subtraction.

Control Flags – The control flags enable or disable certain operations of the microprocessor.There are 3 control flags in 8086 microprocessor and these are:

Directional Flag (D) – This flag is specifically used by string manipulation instructions string instructions. If this flag is 0,the string is processed beginning from the lowest address to the highest address. If this flag is 1,the string is processed beginning from the highest address to the lowest address.

Interrupt Flag: If interrupt flag is set (1), the microprocessor will recognize interrupt requests from the peripherals.

If interrupt flag is reset (0), the microprocessor will not recognize any interrupt requests and will ignore them.

Trap Flag (T) –Setting trap flag puts the microprocessor into single step mode for debugging.

INSTRUCTION SET ARCHITECTURE OF CPU

Register Transfer Language:

- A digital computer system exhibits an interconnection of digital modules such as registers, decoders, arithmetic elements, and Control logic. These digital modules are interconnected with some common data and control paths to form a complete digital system. Digital modules are best defined by the registers and the operations that are performed on the data stored in them.
- The **operations performed on the data stored in registers are called Micro-operations**. A microoperation is an elementary operation performed on the information stored in oneor more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examplesof microoperations are shift, count, clear, and load.
- The **Register Transfer Language** is the **symbolic representation of notations used to specify the sequence of micro-operations**.

In a computer system, **data transfer takes place between processor registers and memory and between processor registers and input-output systems**. These data transfer can be represented by standard notations given below:

- Notations R0, R1, R2..., and so on represent processor registers.
- The addresses of memory locations are represented by names such as LOC, PLACE, MEM, etc.
- Input-output registers are represented by names such as DATA IN, DATA OUT and so on.
- The content of register or memory location is denoted by placing square brackets around the name of the register or memory location.

Register Transfer:

Computer registers are denoted by capital letters (sometimes followed by numerals) to denote the function of the register. The register that holds an address for the memory unit is usually called a **memory address register** and is denoted by **MAR**. Other registers are PC (for program counter), IR (for instruction register, and R1 (for processor register). An n-bit register is sequence of n-flipflops numbered from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.

The most common way to represent a register is by a rectangular box with the name of the register inside, as shown in the figure below. The individual bits can be distinguished as shown in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H(for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC (L) refers to the low-order byte and PC(8-15) or PC(H) to the high-order byte.

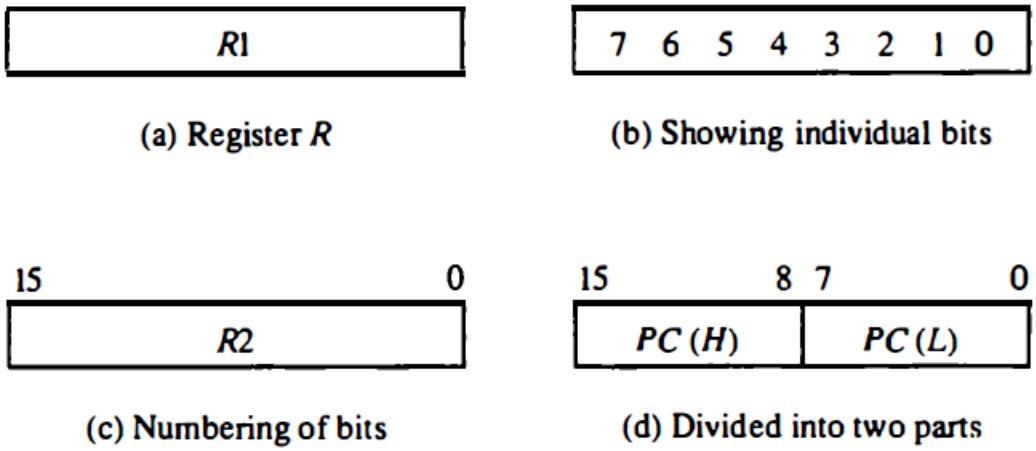


Fig: Block diagram of registers

Information transfer from one register to another is designated in symbolicform by means of a replacement operator as shown below, which denotes a transfer of the contents of register R1 into register R2.Contents of R2 are replaced by the contents of R1.By definition, thecontent of the source register R1 does not change after the transfer.register transfer implies that circuits areavailable from the outputs of the source register to the inputs of the destinationregister.

R2 <--R1

Sometimes, we may want the transfer to occur only under a predetermined controlcondition. This can be shown by means of an if-then statement

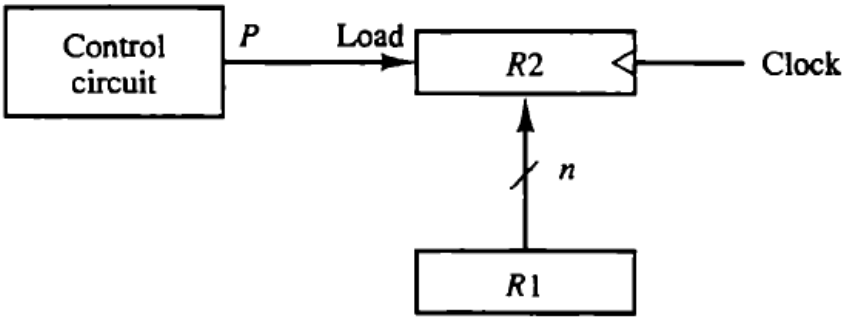
If (P = 1) then (R2 <--R1)

where P is a control signal generated in the control section.A control function is a Boolean variable that isequal to 1 or 0. The control function is included in the statement as follows

P: R2 <--R1

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

Every statement written in a register transfer notation implies a hardwareconstruction for implementing the transfer. Figure below shows the block diagramthat depicts the transfer from R1 to R2. The n outputs of register R1 areconnected to the n inputs of register R2. The letter n will be used to indicateany number of bits for the register. It will be replaced by an actual numberwhen the length of the register is known. Register R2 has a load input that isactivated by the control variable P. It is assumed that the control variable issynchronized with the same clock as the one applied to the register.



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

Note:Even though the control condition such as P becomes active just after time t,the actual transfer does not occur until the register is triggered by the nextpositive transition of the clock at time t + 1 .

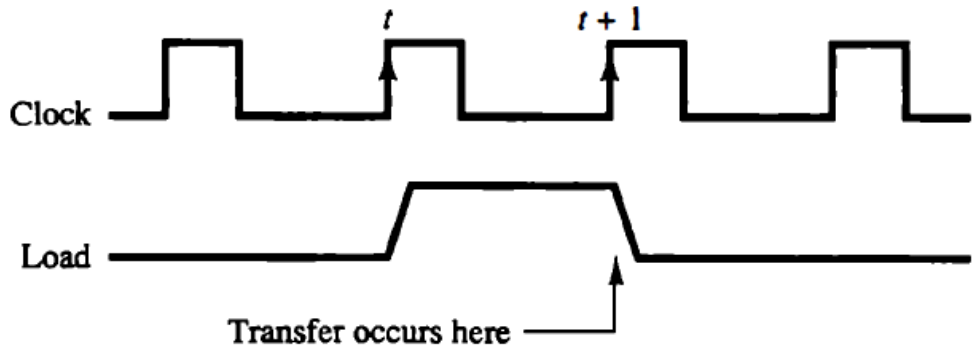


Fig: Timing Diagram

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

The statement

T: R2 <- R1, R1 <- R2

It denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1.

The basic symbols of the register transfer notation are given below:

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*

Fig: Basic symbols of register Transfer

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 thememory address during the transfer. It is necessary to specify the address ofM when writing memory transfer operations. This will be done by enclosingthe address in square brackets following the letter M.

Memory Read: 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.

Memory Write: The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data is in register R1 and the address in AR. The write operation can be stated symbolically as follows:

Write: M[AR] ← R1

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

Instruction cycle:

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 To. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on.

The Micro-operations for the fetch and decode phases can be specified by the following register transfer statements:

T_0: ***AR ← PC***

The address from PC to AR during the clock transition associated with timing signal T0.

T_1: ***IR ←M[AR], PC ←PC + 1***

The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program

T_2: ***D0, …, D7 ← Decode IR(12-14),AR ← IR(0-11), I ← IR(l5)***

At time T2, 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.

Decoding: The timing signal that is active after the decoding is T3. During time T3, the control unit determines the type of instruction that was just read from memory. Decoder output D7, is equal to 1 if the operation code is equal to binary 111. If D7 = 1, the instruction must be a register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 000 through 110,specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a

memoryreferenceinstruction with an indirect address.The microoperation for the indirect addresscondition can be symbolized by the register transfer statement:

AR ← M [AR]

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

When a **memory-reference instruction** with I = 0 is encountered, it is notnecessary to do anything since the effective address is already in AR. However,the sequence counter SC must be incrementedso that theexecution of the memory-reference instruction can be continued with timingvariable T4.After the instruction is executed,SC is cleared to 0 and control returns to the fetch phase with T0 = 1 .

Register-reference instructions are recognized by the control when 07 = 1 andI = 0.The 12 bitsavailable in IR(0-11) are transferred to AR during time T2.These instructions are executed with the clocktransition associated with timing variable T3.The execution of a register-reference instruction is completedat time T3.The sequence counter SC is cleared to 0 and the control goesback to fetch the next instruction with timing signal T0.

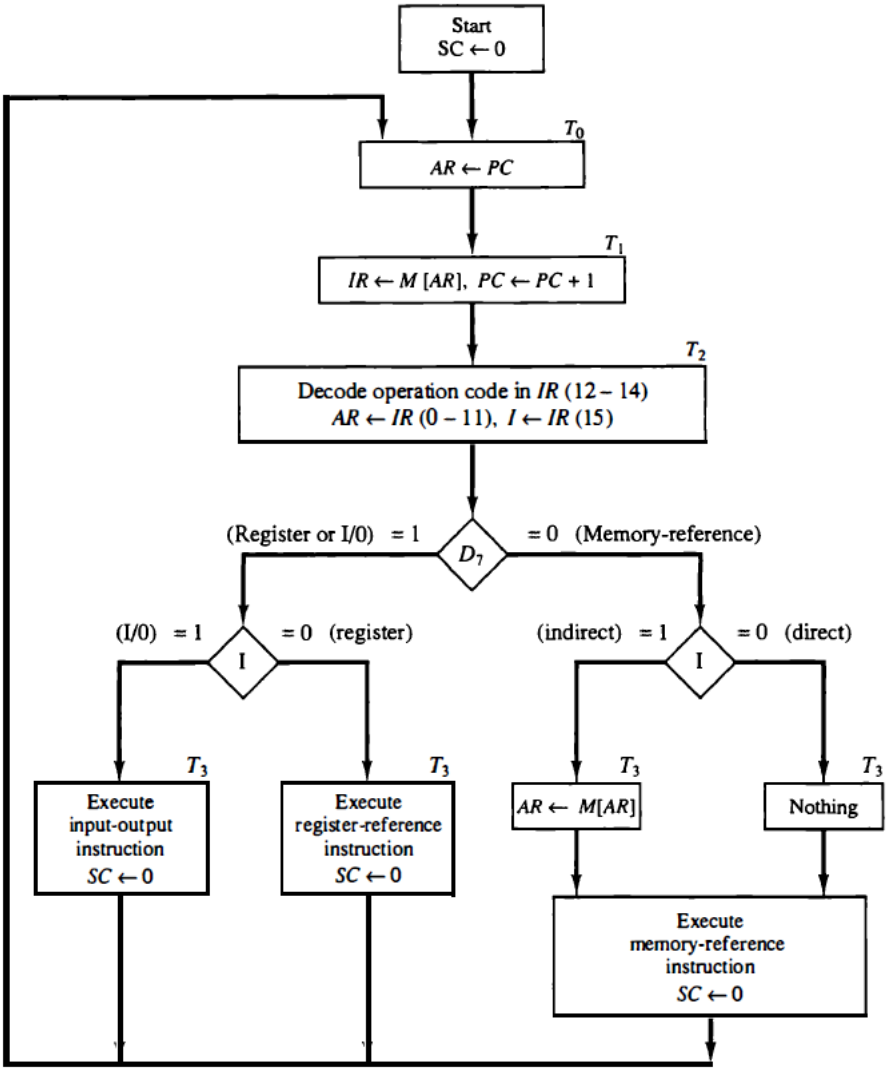


Fig:Flowchart for instruction cycle

Addressing Modes

ADDRESSING MODES OF 8086:

Addressing modes is the manner in which operands are given in an instruction.The addressing modes of 8086 are as follows:

1) **IMMEDIATE ADDRESSING MODE:**In this mode the operand is specified in theinstruction itself. Instructions are longer but the operands are easily identified.
   Eg: MOV CL, 12H ; **Moves 12 immediately into CL register**
   MOV BX, 1234H ; **Moves 1234 immediately into BX register**
2) **REGISTER ADDRESSING MODE**:In this mode operands are specified using registers. Instructions are shorter but operands cant be identified by looking at the instruction.
   Eg: MOV AX,BX
   ADD BX,CX
3) **DIRECT ADDRESSING MODE:**In this mode address of the operand is directly specified in the instruction.
   Eg:MOV CL, [2000H] ; CL Register gets data from memory location 2000H
   CL ←[2000H]
   MOV [3000H], DL ; Memory location 3000H gets data from DL Register
   [3000H] ← DL
4) **INDIRECT ADDRESSING MODE:** In Indirect Addressing modes, address is given by a register. The register can be incremented in a loop to access a series of locations. There are various sub-types of Indirect addressing mode.
   REGISTER INDIRECT ADDRESSING MODE
   This is the most basic form of indirect addressing mode.Here address is simply given by a register.
   Eg: MOV CL, [BX] ; CL gets data from a memory location pointed by BX
   CL ← [BX]. If BX = 2000H, CL ← [2000H]
   Eg: MOV [BX], CL ; CL is stored at a memory location pointed by BX
   [BX] ←CL. If BX = 2000H, [2000H] ← CL.

REGISTER RELATIVE ADDRESSING MODE : Here address is given by a register plus a numeric displacement.

Eg: MOV CL, [BX + 03H] ; CL gets data from a location BX + 03H

CL ← [BX+03H]. If BX = 2000H, then CL ← [2003H]

Eg: MOV [BX + 03H], CL ; CL is stored at location BX + 03H

[BX+03H] ←CL. If BX = 2000H, then [2003H] ← CL.

BASE INDEXED ADDRESSING MODE Here address is given by a sum of two registers. This is typically useful in accessing an array or a look up table. One register acts as the base of the array holding its starting address and the other acts as an index indicating theelement to be accessed.

Eg: MOV CL, [BX + SI] ; CL gets data from a location BX + SI ; CL ← [BX+SI]. ;

If BX = 2000H, SI = 1000H, then CL ç [3000H] Eg: MOV [BX + SI], CL ; CL is stored at location BX + SI ; [BX+SI] ← CL. ; If BX = 2000H, SI = 1000H, then [3000H] ← CL.

BASE RELATIVE PLUS INDEX ADDRESSING MODE Here address is given by a sum of base register plus index register plus a numeric displacement.

Eg: MOV CL, [BX+SI+03H] ; CL gets data from a location BX + SI + 03H ;

CL ←[BX+SI+03H]. ;

If BX = 2000H, SI = 1000H, then CL ← [3003H]

Eg: MOV [BX+SI+03H], CL ; CL is stored at location BX + SI + 03H ;

[BX+SI+03H] ←CL. ;

If BX = 2000H, SI = 1000H, then [3003H] ← CL.

IMPLIED ADDRESSING MODE: In this addressing mode, the operand is not specified at all, as it is an implied operand. Someinstructions operate only on a particular register. In such cases, specifying the register becomesunnecessary as it becomes implied.

Eg: STC ; Sets the Carry flag.; This instruction can only operate on the Carry Flag.

Eg: CMC ; Complements the Carry flag.; This instruction can only operate on the Carry Flag

Instruction Set:

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

1) Data Transfer Instructions:Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.Table below gives a list of eight data transfer instructions used in many computers.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Accompanying each instruction is a mnemonic symbol. Different computers use different mnemonics for the same instruction name.

The **load instruction** has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The **store instruction** designates a transfer from a processor register into memory. The **move instruction** has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The **exchange instruction** swaps information between two registers or a register and a memory word. The **input and output instructions** transfer data among processor registers and input or output terminals. The **push and pop** instructions transfer data between processor registers and a memory stack.

2) Data Manipulation Instructions: The data manipulation instructions in a typical computer are usually divided into three basic types:

- ✓ Arithmetic instructions
- ✓ Logical and bit manipulation instructions

- ✓ Shift instructions

Arithmetic instructions :The four basic arithmetic operations are addition, subtraction, multiplication,and division. Most computers provide instructions for all four operations.Some small computers have only addition and possibly subtraction instructions.

A list of typical arithmetic instructions is given in Table given below:

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

The **increment instruction** adds 1 to the value stored in a register or memory word.The **decrement instruction** subtracts 1 from a value stored in a registeror memory word.The add, subtract, multiply, and divide instructions may be available fordifferent types of data. The data type assumed ·to be in processor registersduring the execution of these arithmetic operations is included in the definitionof the operation code. An arithmetic instruction may specify fixed-point orfloating-point data, binary or decimal data, single-precision or double-precision data.

The mnemonics for three add instructions that specifydifferent data types are shown below:

ADDI Add two binary integer numbers

ADDF Add two floating-point numbers

ADDD Add two decimal numbers in BCD

The instruction "**add with carry**" performs the addition on two operands plus the value of the carry fromthe previous computation. Similarly, the "**subtract with borrow**" instructionsubtracts two words and a borrow which may have resulted from a previoussubtract operation. The **negate instruction** forms the 2' s complement of anumber, effectively reversing the sign of an integer when represented in thesigned-2's complement form.

Logical and Bit Manipulation Instructions: Logical instructions **perform binary operations on strings of bits** stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The **logical instructions consider each bit of the operand separately** and treat it as a Boolean variable. By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some logical and bit manipulation instructions are shown in the figure below:

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

The clear instruction causes the specified operand to be replaced by D's.The complement instruction produces the 1's complement by inverting all thebits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Although theyperform Boolean operations, when used in computer instructions, the logicalinstructions should be considered as performing bit manipulation operations.There are three-bit manipulation operations possible: a selected bit can becleared to 0, or can be set to 1, or can be complemented. The three logicalinstructions are usually applied to do just that.

Shift Instructions: Shifts are operations in which the bits of a word are moved to the left or right. Shift instructions may specify either logicalshifts, arithmetic shifts, or rotate-type operations. In either case the shift maybe to the right or to the left.Table below lists four types of shift instructions:

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

The **logical shift** inserts 0to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

The **arithmetic shift-right** instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus, a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

Program Control Instructions: Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter.

Some program control instructions are listed in Table below:

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. Branch instruction is written as **BR ADR**, where ADR is a symbolic name for an address. Branch and jump instructions may be conditional or unconditional. Anunconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a conditionsuch as branch if positive or branch if zero. If the condition is met, the programcounter is loaded with the branch address and the next instruction is taken.from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

The **skip instruction** does not need an address field and is therefore azero-address instruction. A conditional skip instruction will skip the nextinstruction if the condition is met. If the condition is not met, control proceeds with thenext instruction in sequence.

The **call and return** instructions are used in conjunction with subroutines.

The **compare instruction** performs a subtraction between two operands, but the result of the operation isnot retained. However, certain status bit conditions are set as a result of theoperation. Similarly, the **test instruction** performs the logical AND of twooperands and updates certain status bits without retaining the result or changing the operands. (***Note:***The compare and test instructions do not change the program sequence directly. They are listed in Table because of their application in setting conditions for subsequent conditional branch instructions)

RISC vs CISC Architecture

RISC stands for Reduced Instruction set Computer and **CISC** stands for Complex Instruction Set Computer.RISC and CISC are the two ideologies behind making the processor.

	RISC	CISC
1	**Instructions of a fixed size**	Instructions of **variable size**
2	Most instructions take **same time to fetch**.	Instructions have **different fetching times**.
3	Instruction set **simple and small**.	Instruction set **large and complex**.
4	**Less addressing modes** as most operations are register based.	**Complex addressing modes** as most operations are memory based.
5	**Compiler design is simple**	**Compiler design is complex**
6	**Total size of program is large** as many instructions are required to perform a task as instructions are simple.	**Total size of program is small** as few instructions are required to perform a task as instructions are complex & more powerful.
7	Instructions use a **fixed number of operands.**	Instructions have **variable** number of operands.
8	Ideal for processors performing a **dedicated operation**.	Ideal for processors performing a **verity of operations**.
9	Since instructions are simple, they can be decoded by a **hardwired control unit**.	Since instructions are complex, they require a **Micro-programmed Control Unit**.
10	Execution speed is **faster** as most operations are register based.	Execution speed is **slower** as most operations are memory based.
11	As No. of cycles per instruction is fixed, it gives a **better degree of pipelining**	Since number of cycles per instruction varies, **pipelining has more bubbles** or stalls.
12	E.g.:: **ARM7**, **PIC 18** Microcontrollers.	E.g.: Intel **8085, 8086** Microprocessors.
13	They have **register based operations**.	13. **Memory based** operations.

CPU CONTROL UNIT DESIGN

- **Hardwired CU :**
  In Hardwired CU, control signals are produced by hardware. There are three types of Hardwired Control Units
  1) **STATE TABLE METHOD**
  2) **DELAY ELEMENT METHOD**
  3) **SEQUENCE COUNTER METHOD**

STATE TABLE METHOD:

1) It is the most basic type of hardwired control unit.
2) Here the behaviour of the control unit is represented in the form of a table called the state table.
3) The rows represent the T-states and the columns indicate the instructions.
4) Each intersection indicates the control signal to be produced, in the corresponding T-state of everyinstruction.
4) A circuit is then constructed based on every column of this table, for each instruction.

T-STATES	INSTRUCTIONS			
	I_1	I_2	...	I_N
T_1	$Z_{1,1}$	$Z_{1,2}$	...	$Z_{1,N}$
T_2	$Z_{2,1}$	$Z_{2,2}$	...	$Z_{2,N}$
...	...	...	...	...
T_M	$Z_{M,1}$	$Z_{M,2}$	...	$Z_{M,N}$

$Z_{1,1}$: Control Signal to be produced in T-state (T_1) of Instruction (I_1)

ADVANTAGE: It is the simplest method and is ideally suited for very small instruction sets.

DRAWBACK: As the number of instructions increase, the circuit becomes bigger and hence more complicated. As a tabular approach is used, instead of a logical approach (flowchart), there are duplications of manycircuit elements in various instructions.

Delay Element Method:

1) Here the behaviour of the control unit is represented in the form of a flowchart.

2) Each step in the flowchart represents a control signal to be produced.

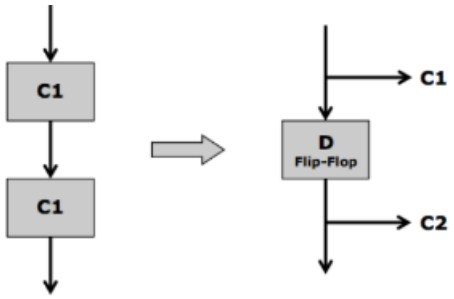
3) Once all steps of a particular instruction, are performed, the complete instruction gets executed.

4) Control signals perform Micro-Operations, which require one T-states each.

5) Hence between every two steps of the flowchart, there must be a delay element.

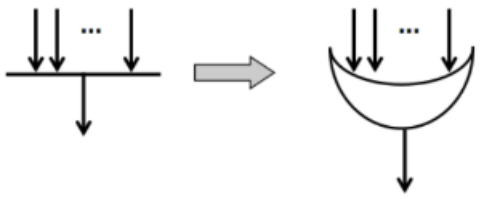
6) The delay must be exactly of one T-state. This delay is achieved by D Flip-Flops.

7) These D Flip-Flops are inserted between every two consecutive control signals.

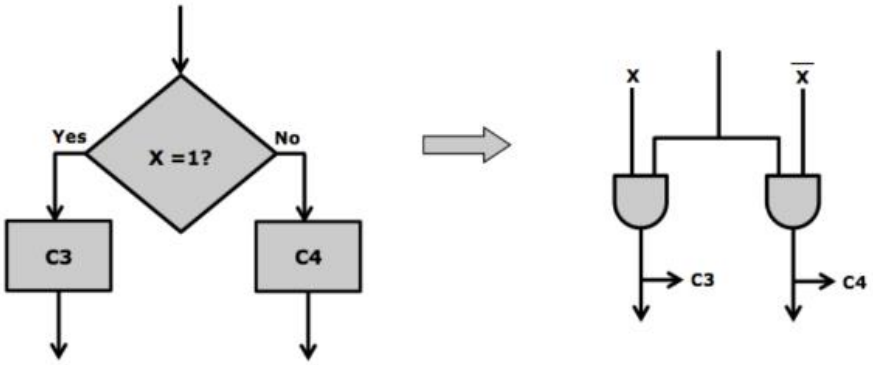


8) Of all D Flip-Flops only one will be active at a time. So the method is also called “One Hot Method”.

9) In a multiple entry point, to combine two or more paths, we use an OR gate.



10)A decision box is replaced by a set of two complementing AND gates



11) A multiple entry point is substituted by an OR gate.

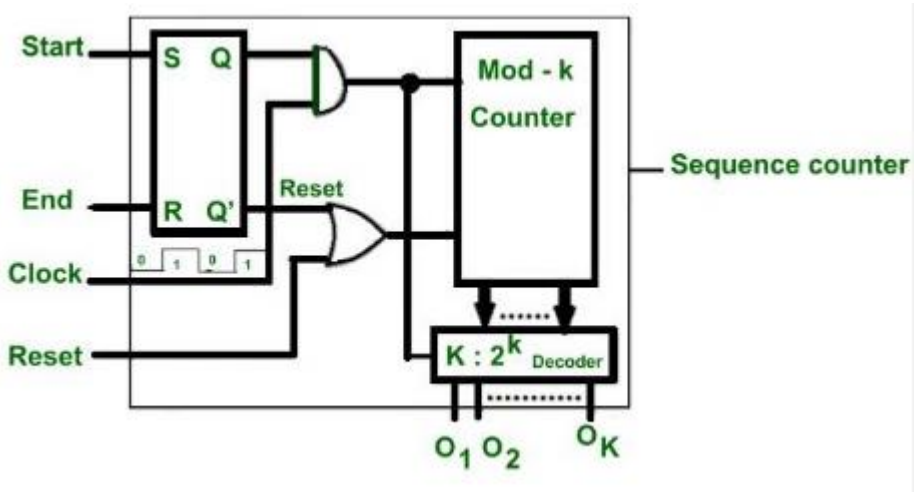
ADVANTAGE:

As the method has a logical approach, it can reduce the circuit complexity.This is done by re-utilizing common elements between various instructions.

DRAWBACK:

As the no of instructions increase, the number of D Flip-Flops increase, so the cost increases.Moreover, only one of those D Flip-Flops are actually active at a time.

SEQUENCE COUNTER METHOD:



1) This is the most popular form of hardwired control unit. The goal of this circuit is to provide triggers to different parts of the circuit after gaps of 1-Tstate.

2) It follows the same logical approach of a flowchart, like the Delay element method, but does not use all those unnecessary D Flip-Flops because at any point of time only one delay element is active and a complex circuitry would involve many delay elements which is very inefficient. The D-Flip-flops are replaced by trigger points which are activated after gaps of one T-state.

Following are the steps involved in designing a CU using Sequence Counter Method.

1) First a flowchart is made representing the behaviour of a control unit.

2) It is then converted into a circuit using the same principle of AND & OR gates.

3) We need a delay of 1 T-state (one clock cycle) between every two consecutive control signals.

4) That is achieved by the above circuit.

5) If there are “k” number of distinct steps producing control signals, we employ a “mod k” and “k”output decoder.

6) The counter will start counting at the beginning of the instruction.

7) The “clock” input via an AND gate ensures each count will be generated after 1 T-state.

8)The count is given to the decoder which triggers the generation of “k” control signals, each aftera delay of 1 T-state.

9)When the instruction ends, the counter is reset so that next time, it begins from the first count.

ADVANTAGE:

Avoids the use of too many D Flip-Flops.

GENERAL DRAWBACKS OF A HARDWIRED CONTROL UNIT

1) Since they are based on hardware, as the instruction set increases, the circuit becomes more and more complex. For modern processors having hundreds of instructions, it is virtually impossible to create Hardwired Control Units.

2) Such large circuits are very difficult to debug.

3) As the processor gets upgraded, the entire Control Unit has to be redesigned, due to the rigid nature of hardware design.

Microprogrammed CU

WILKES’ DESIGN FOR A MICROPROGRAMMED CONTROL UNIT:

1) Microprogrammed Control Unit produces control signals by software, using micro-instructions

2) A program is a set of instructions.

3) An instruction requires a set of Micro-Operations.

4) Micro-Operations are performed by control signals.

5) Instead of generating these control signals by hardware, we use micro-instructions. This means every instruction requires a set of micro-instructions This is called its micro-program.

6) Microprograms for all instructions are stored in a small memory called “Control Memory”. The Control memory is present inside the processor.

7) Consider an Instruction that is fetched from the main memory into the Instruction Register (IR).

8) The processor uses its unique “opcode” to identify the address of the first micro-instruction. That address is loaded into CMAR (Control Memory Address Register). CMAR passes the address to the decoder.

9) The decoder identifies the corresponding micro-instruction from the Control Memory.

10) A micro-instruction has two fields: a control filed and an address field.

Control field: Indicates the control signals to be generated.

Address field: Indicates the address of the next micro-instruction.

11) This address is further loaded into CMAR to fetch the next micro-instruction.

12) For a conditional micro-instruction, there are two address fields.This is because, the address of the next micro-instruction depends on the condition.The condition (true or false) is decided by the appropriate control flag.

13) The control memory is usually implemented using FLASH ROM as it is writable yet non-volatile.

ADVANTAGES

1) The biggest advantage is flexibility.

2) Any change in the control unit can be performed by simply changing the micro-instruction.

3) This makes modifications and up gradation of the Control Unit very easy.

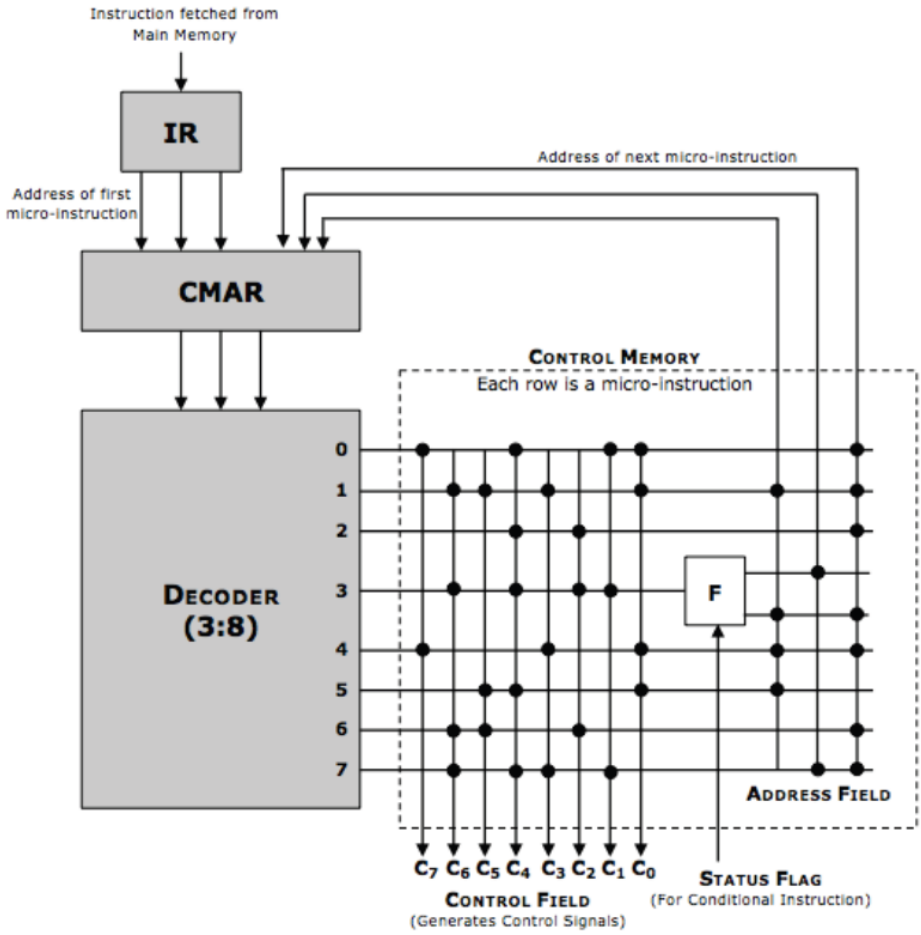
4) Moreover, software can be much easily debugged as compared to a large Hardwired Control Unit.

DRAWBACKS

1) Control memory has to be present inside the processor, increasing its size.

2) This also increases the cost of the processor.

3) The address field in every micro-instruction adds more space to the control memory. This can beeasily avoided by proper micro-instruction sequencing.



<u>TYPICAL MICROPROGRAMMED CONTROL UNIT</u>

1) Microprogrammed Control Unit produces control signals by software, using micro-instructions.

2) A program is a set of instructions.

3) An instruction requires a set of Micro-Operations.

4) Micro-Operations are performed by control signals.

5) Here, these control signals are generated using micro-instructions.

6) This means every instruction requires a set of micro-instructions

7) This is called its micro-program.

8) Microprograms for all instructions are stored in a small memory called “Control Memory”.

9) The Control memory is present inside the processor.

10) Consider an Instruction that is fetched from the main memory into the Instruction Register (IR).

11) The processor uses its unique “opcode” to identify the address of the first micro-instruction.

12) That address is loaded into CMAR (Control Memory Address Register) also called μIR.

13) This address is decoded to identify the corresponding μ-instruction from the Control Memory.

14) There is a big improvement over Wilkes’ design, to reduce the size of micro-instructions.

15) Most micro-instructions will only have a Control field.

16) The Control field Indicates the control signals to be generated.

17) Most micro-instructions will not have an address field.

18) Instead, μPC will simply get incremented after every micro-instruction.

19) This is as long as the μ-program is executed sequentially.

20) If there is a branch μ-instruction only then there will be an address filed.

21) If the branch is unconditional, the branch address will be directly loaded into CMAR.

22) For Conditional branches, the Branch condition will check the appropriate flag.

23) This is done using a MUX which has all flag inputs.

24) If the condition is true, then the MUX will inform CMAR to load the branch address.

25) If the condition is false CMAR will simply get incremented.

26) The control memory is usually implemented using FLASH ROM as it is writable yet non-volatile.

ADVANTAGES

1) The biggest advantage is flexibility.

2) Any change in the control unit can be performed by simply changing the micro-instruction.

3) This makes modifications and up gradation of the Control Unit very easy.

4) Moreover, software can be much easily debugged as compared to a large Hardwired Control Unit.

5) Since most micro-instructions are executed sequentially, they don’t need for an address field.

6) This significantly reduces the size of micro-instructions, and hence the Control Memory.

DRAWBACKS

1) Control memory has to be present inside the processor, increasing its size.

2) This also increases the cost of the processor.