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

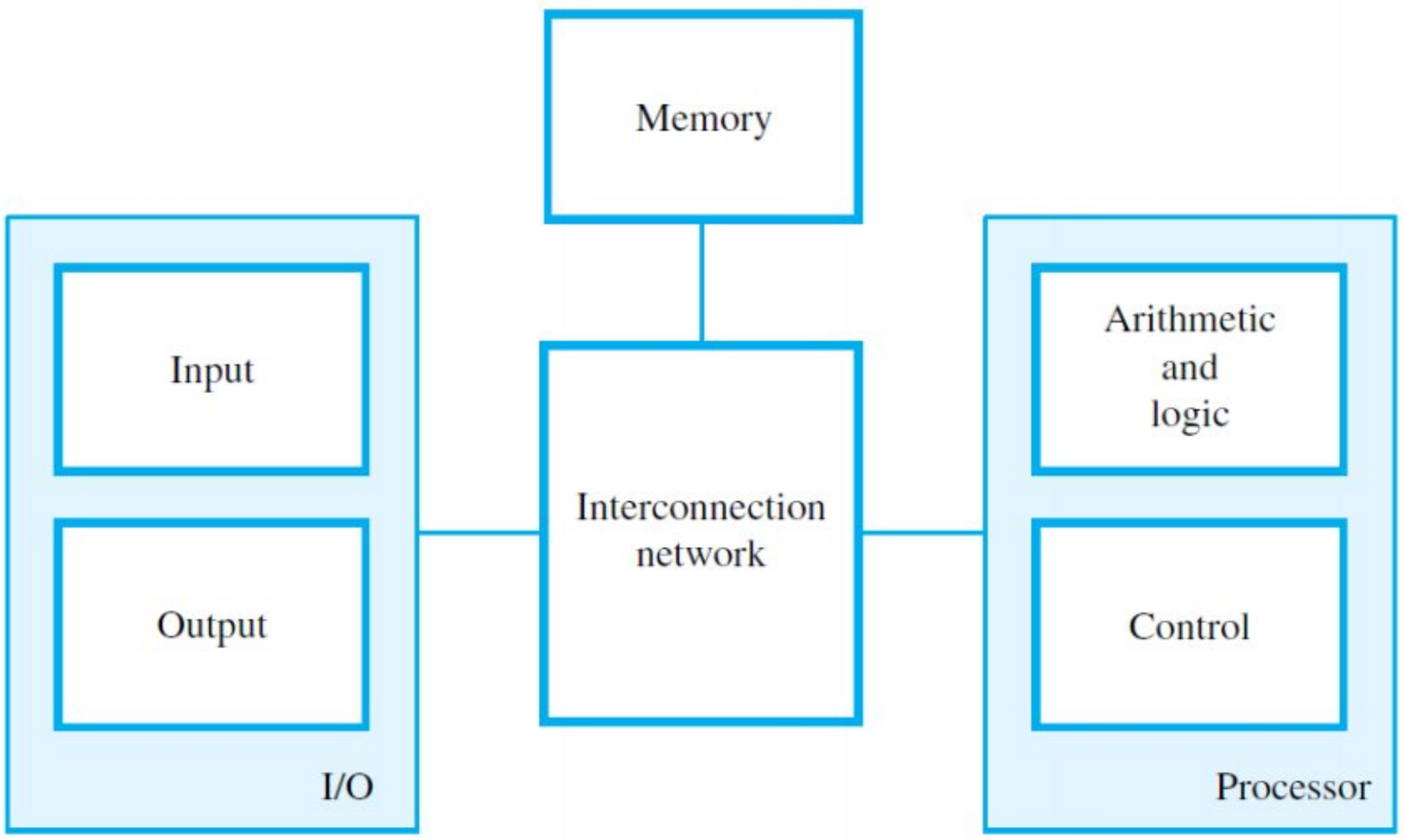


Fig:Basic functional units of a computer.

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.

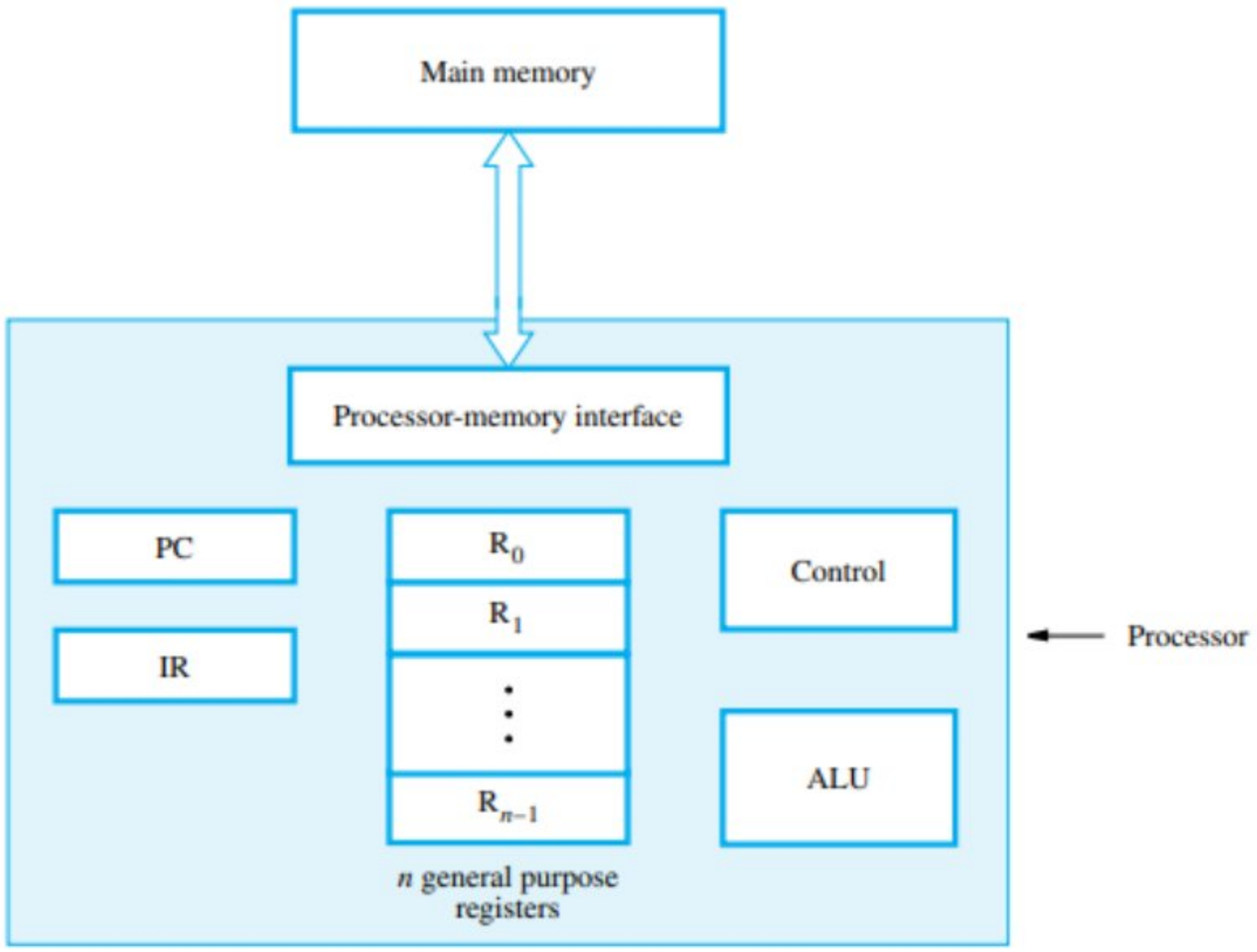


Fig 1.1: ***Connection between the processor and the main memory***

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.

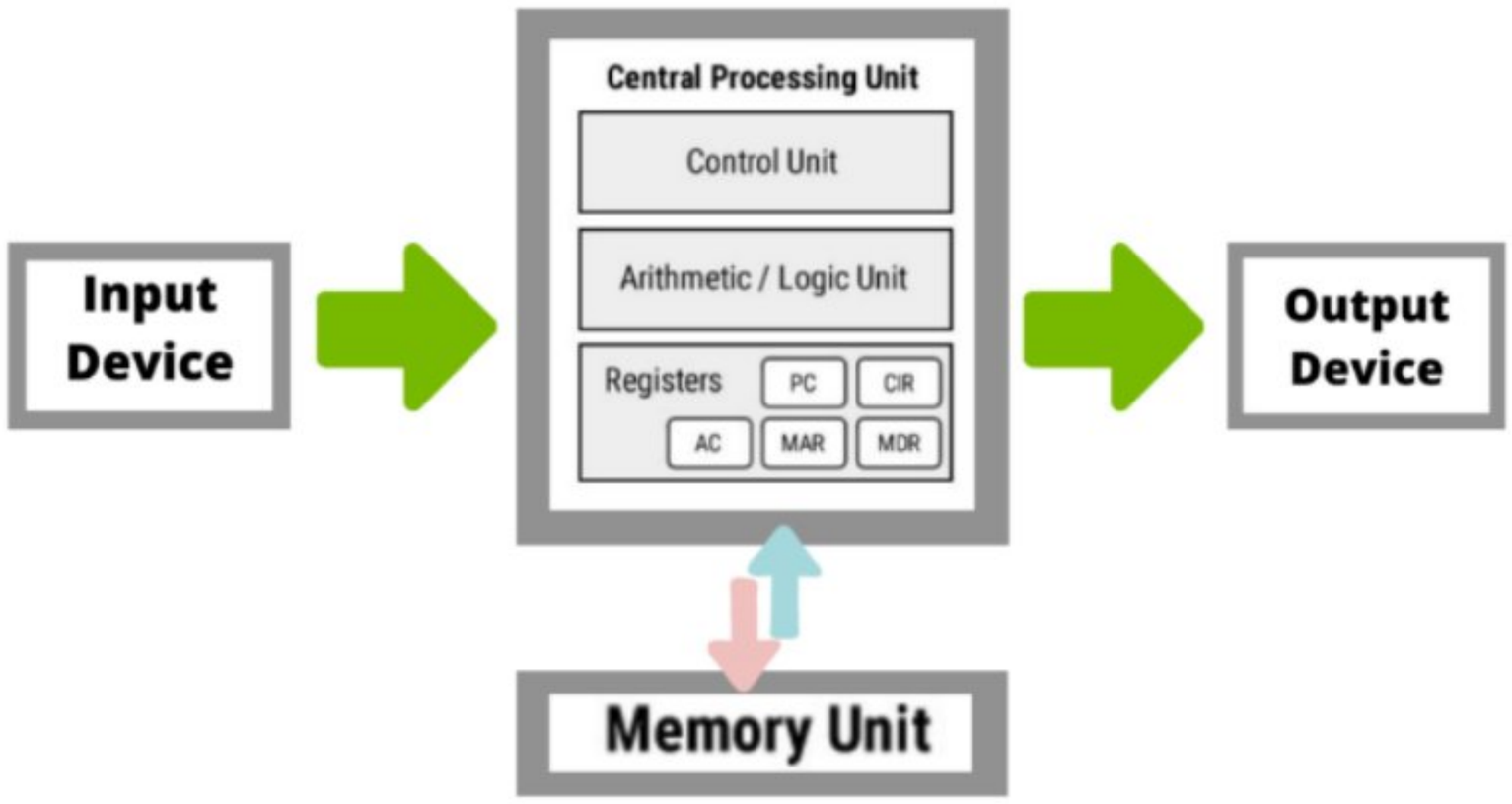
At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

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**. <u>I/o Devices</u>**: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.

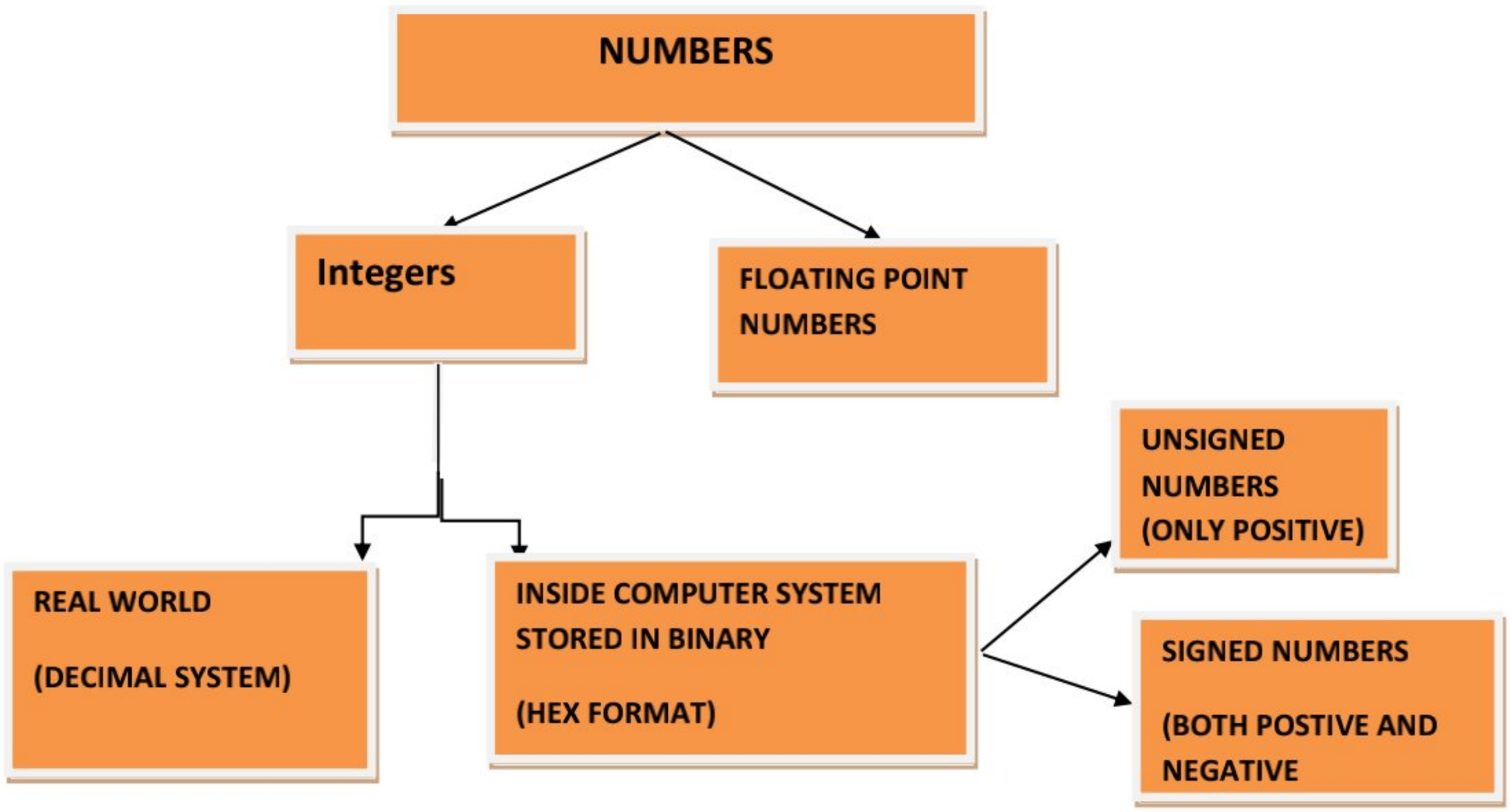
4. **<u>Memory:</u>**A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words 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

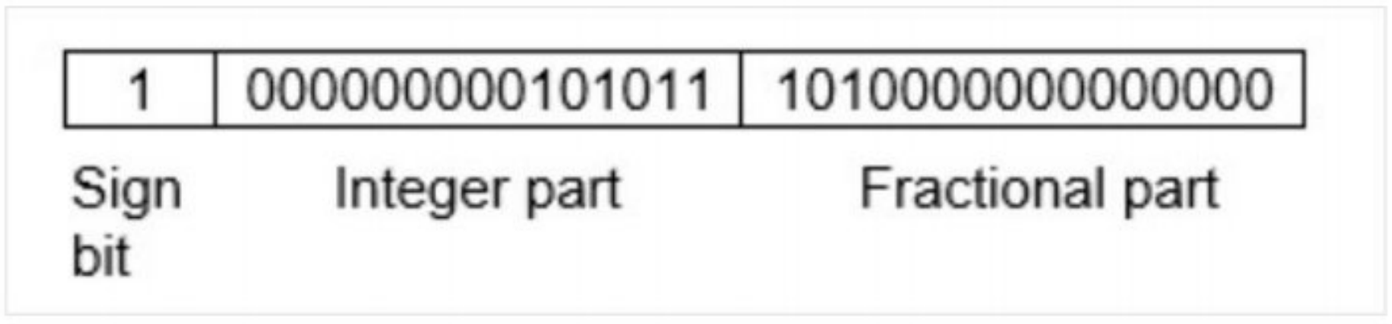
(i) Fixed Point Notation and

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

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

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

```
    0       1       0       1
  + 0     + 0     + 1     + 1
  ---     ---     ---     ---
    0       1       1      10
                           ↑
                       Carry-out
```

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**

$$
\begin{array}{cccccc}
 & & 0 & 1 & 1 & 1 \\
+ & & 1 & 1 & 0 & 1 \\
\hline
1 & & 0 & 1 & 0 & 0 \\
\uparrow & & & & & \\
\text{Carry-out} & & & & &
\end{array}
$$

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

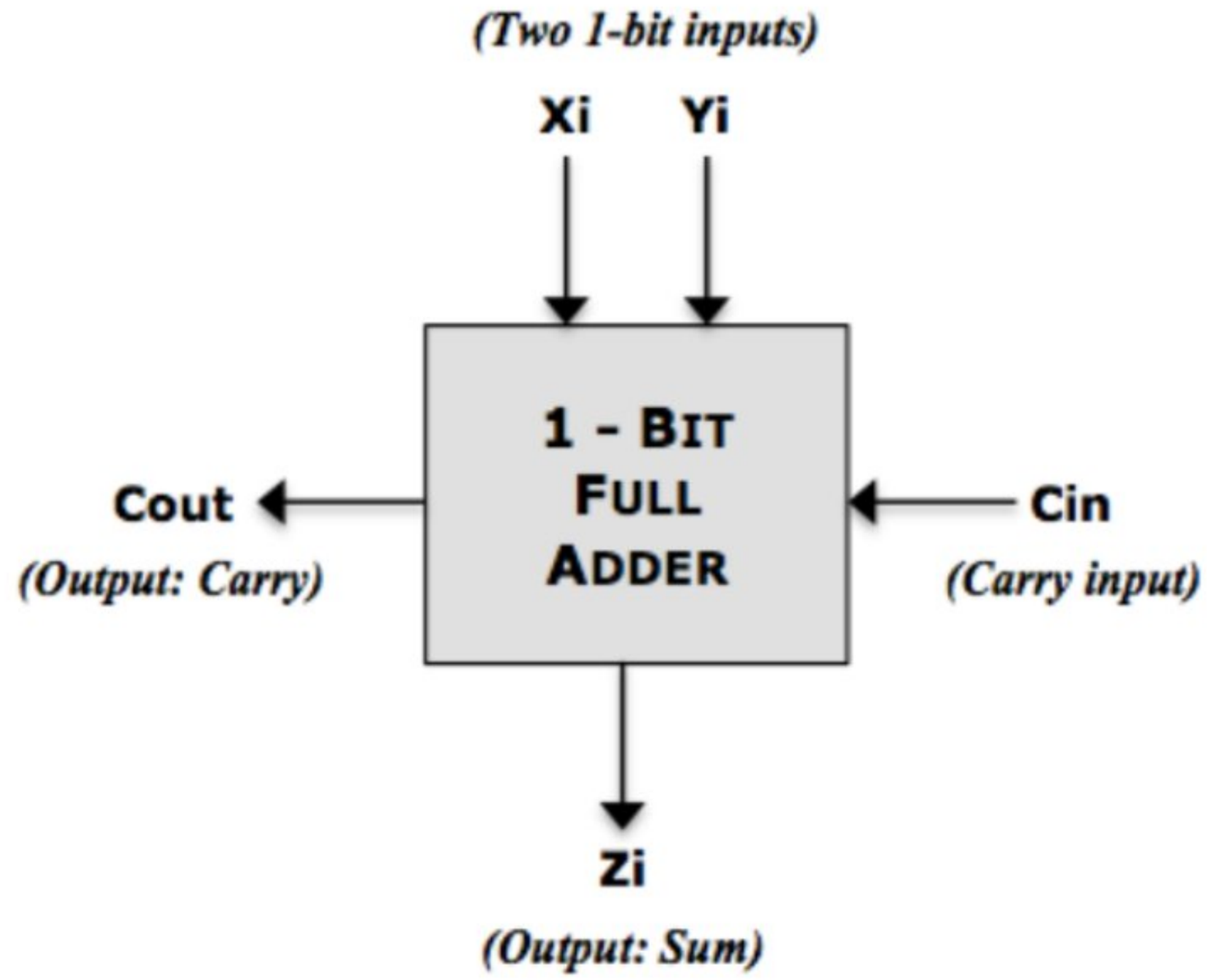
1) It is a 1-bit adder circuit.

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)

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

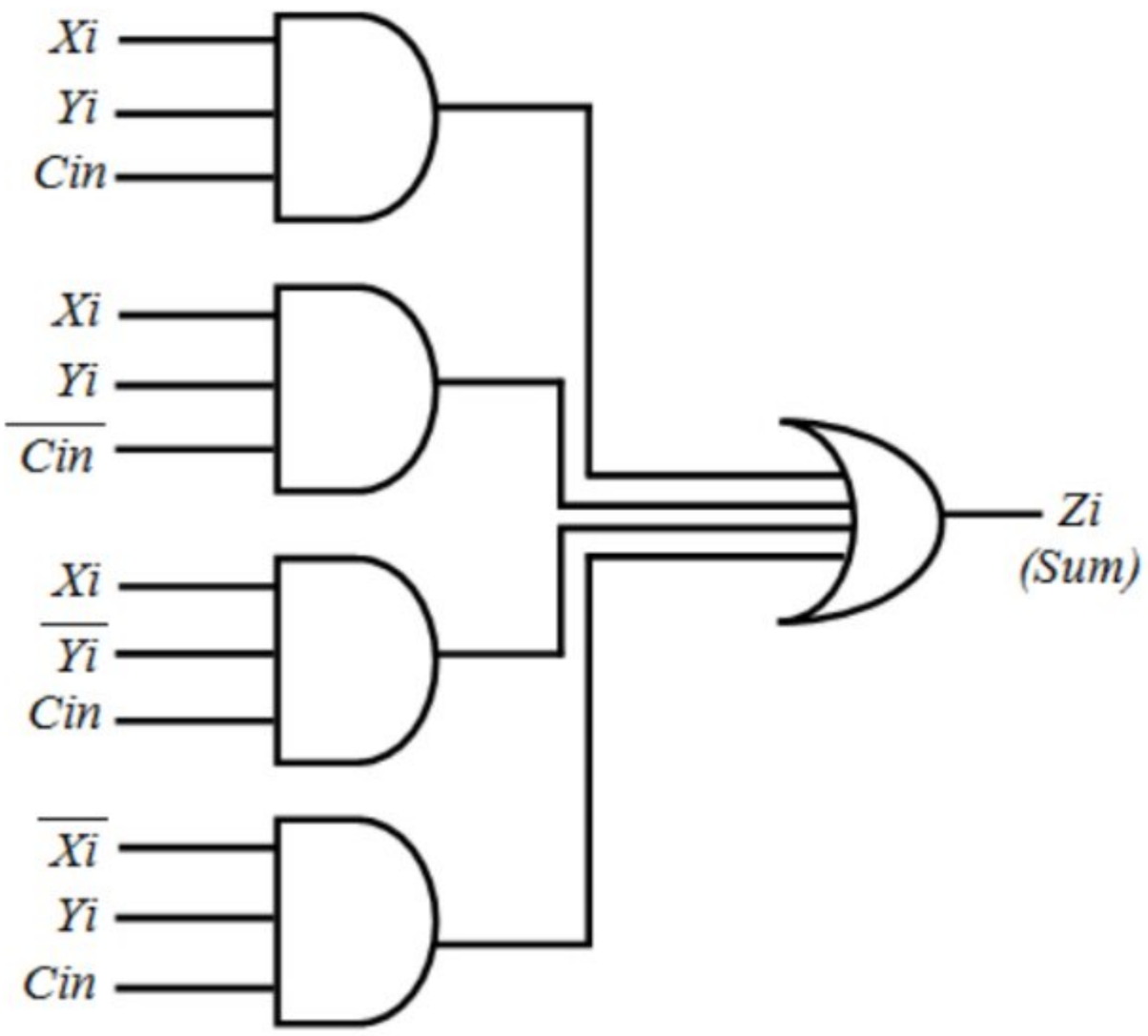


Fig: Circuit for Sum

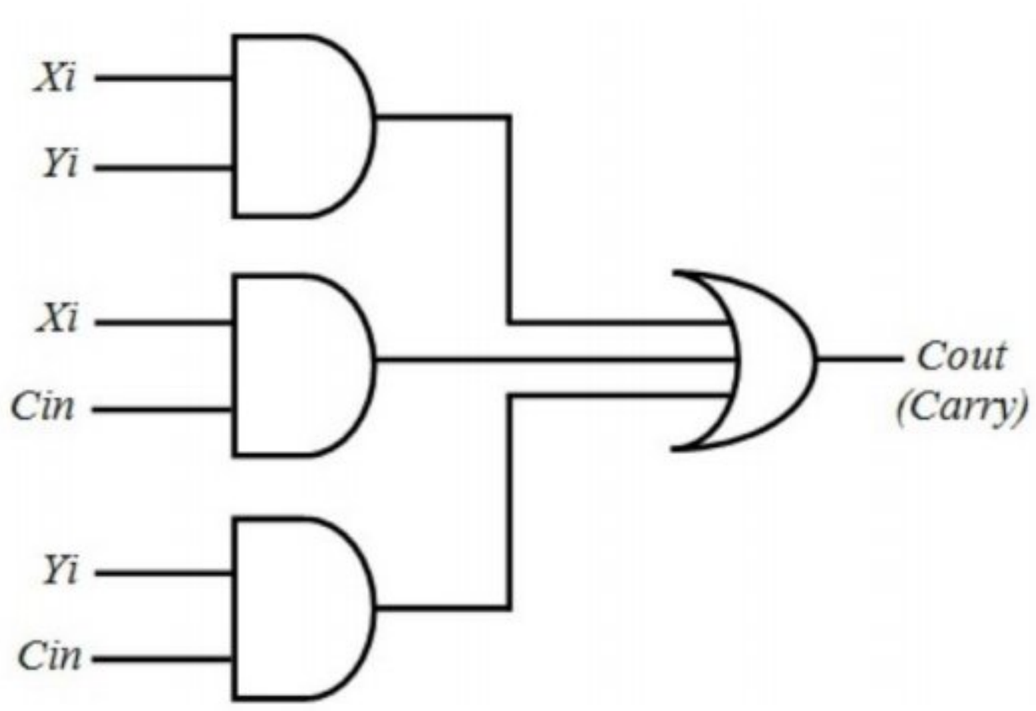


Fig: Circuit for carry

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

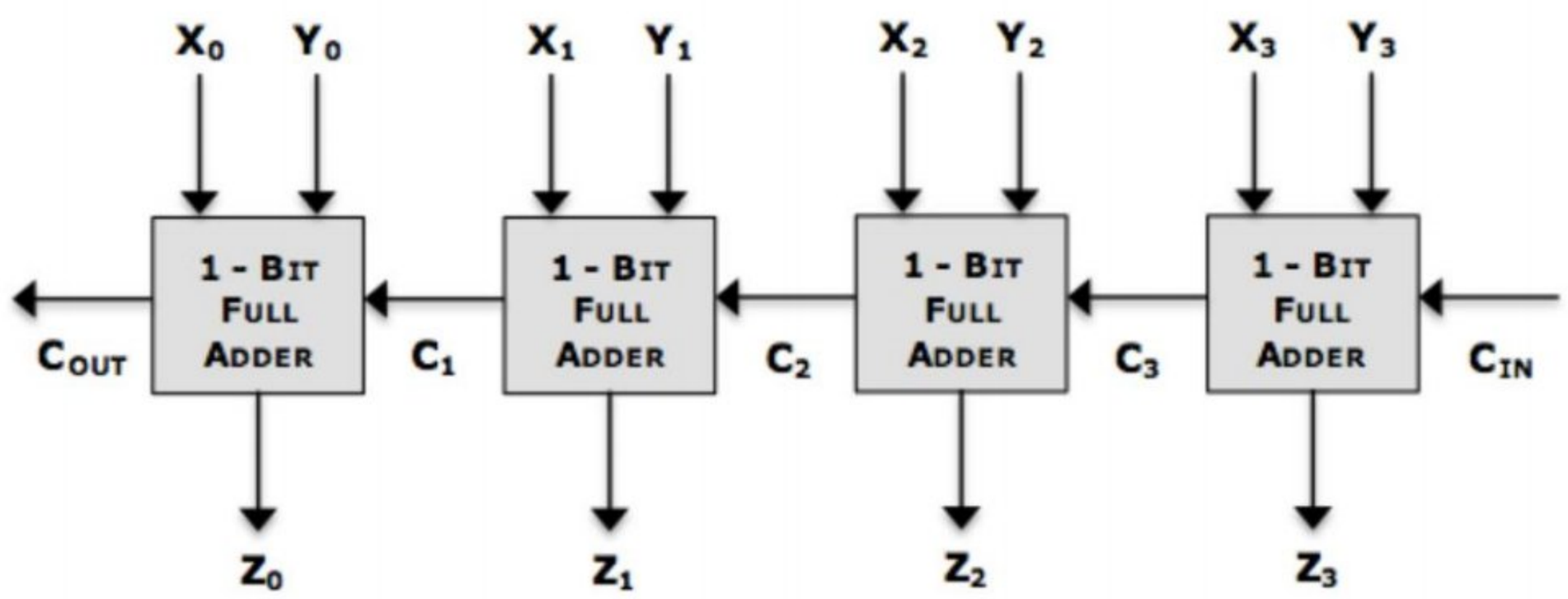


Fig:4-bit Ripple Carry Adder

<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

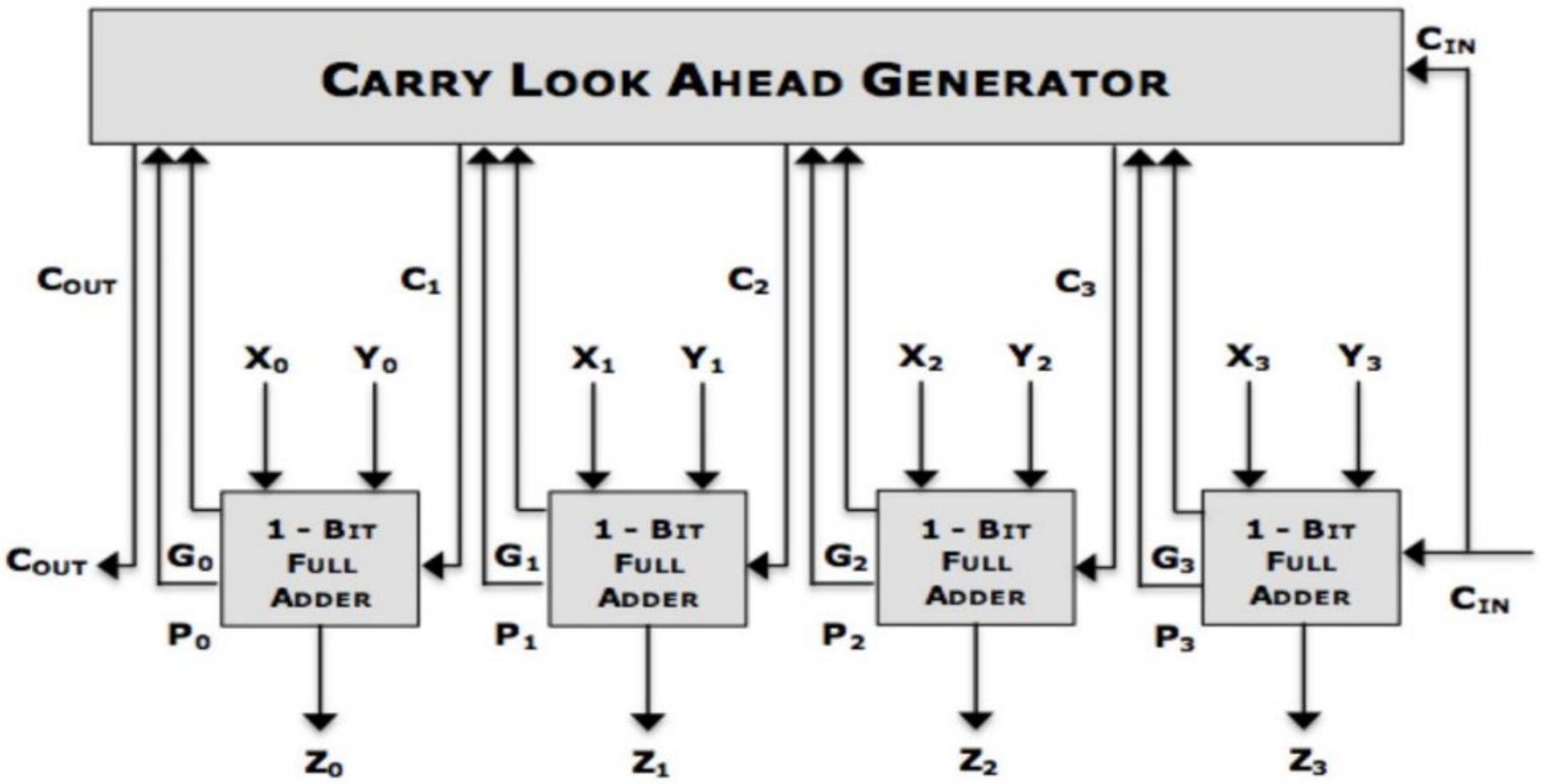


Fig: Circuit for Carry Look ahead Adder

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) <u>Shift and Add:</u> 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.

<u>Algorithm:</u>

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.

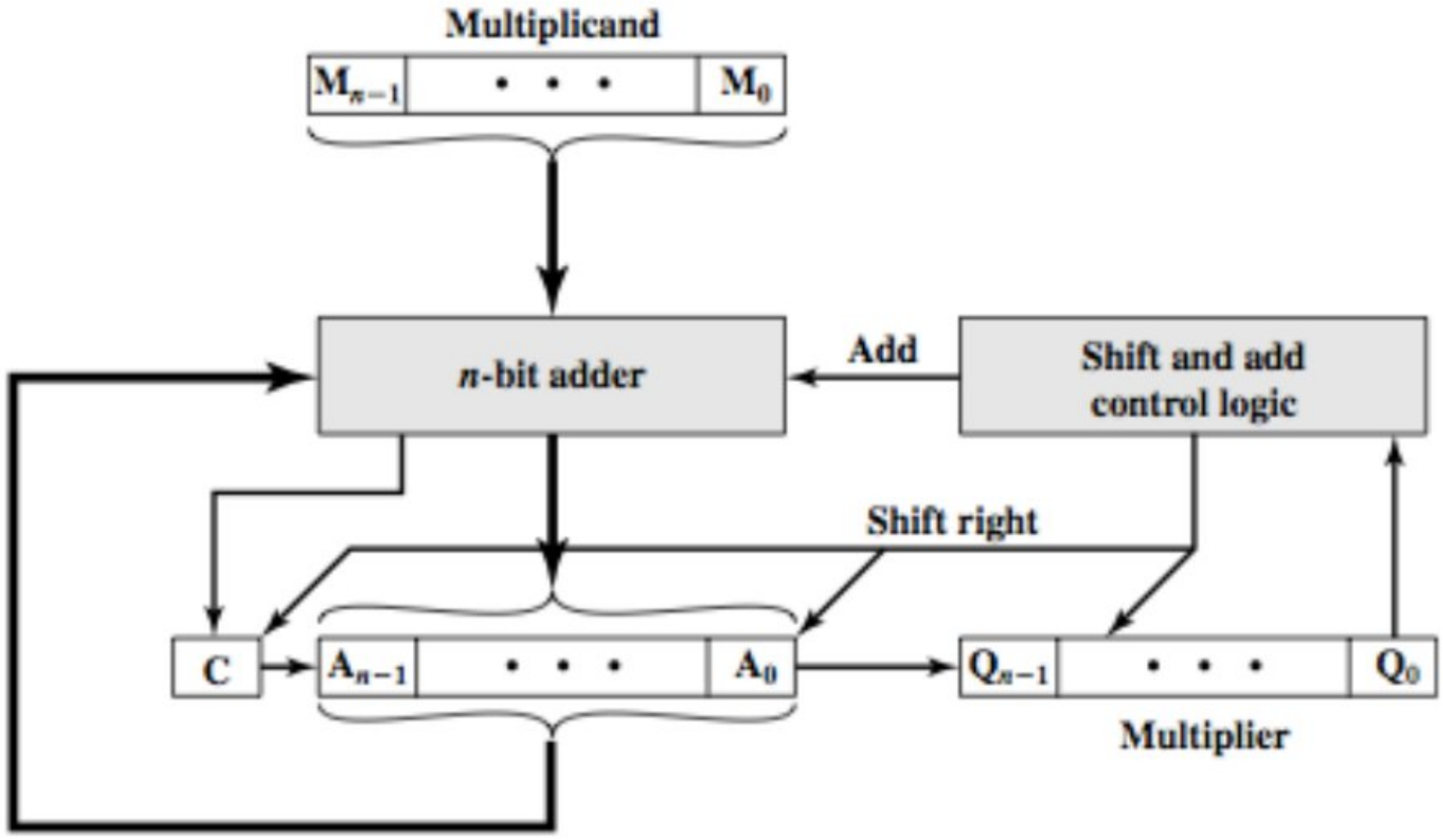


Fig: Shift and Add Multiplication

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**.

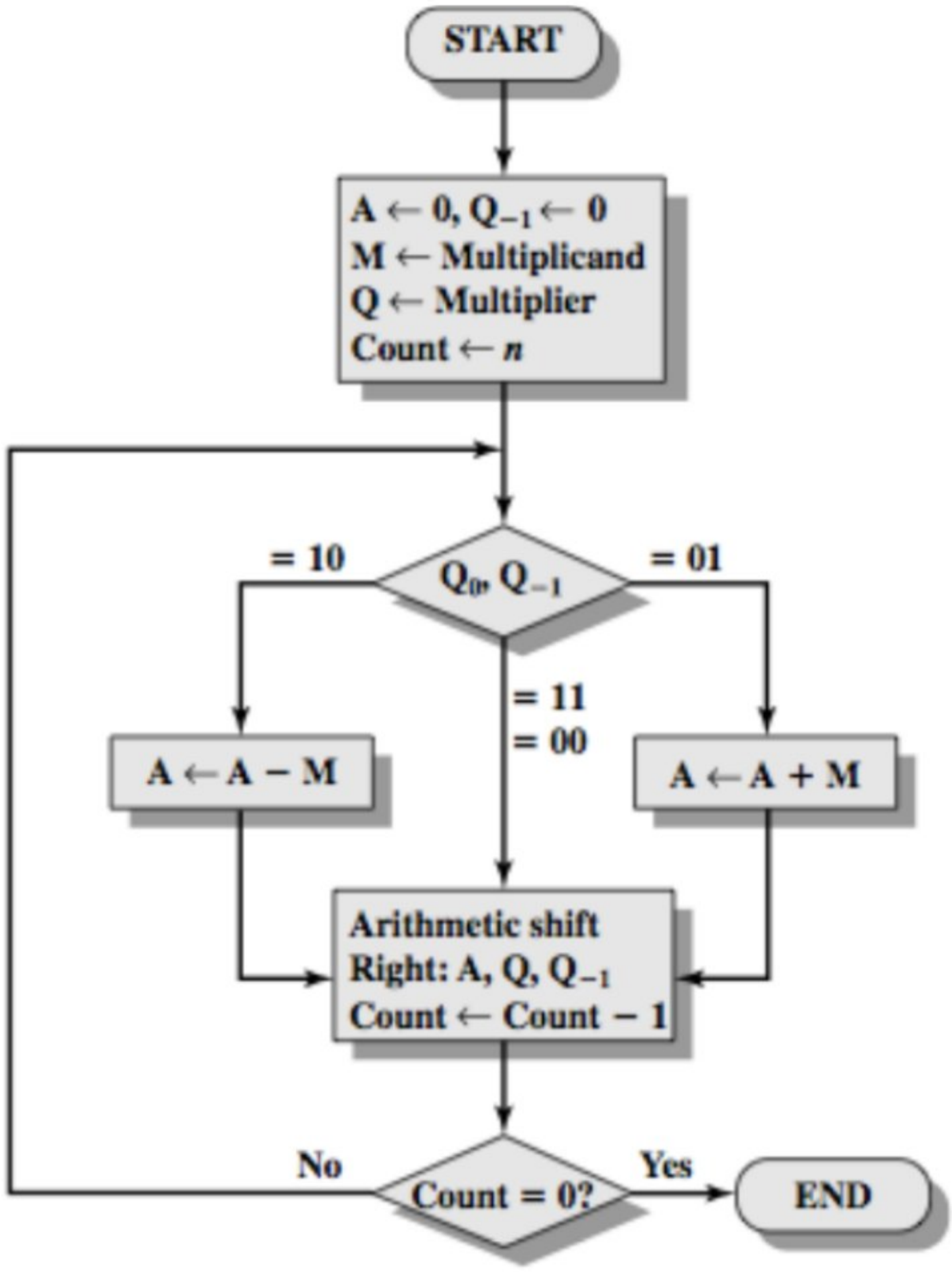
4) If the transition is from **"0 to 0"** then **simply Right-Shift**.

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.

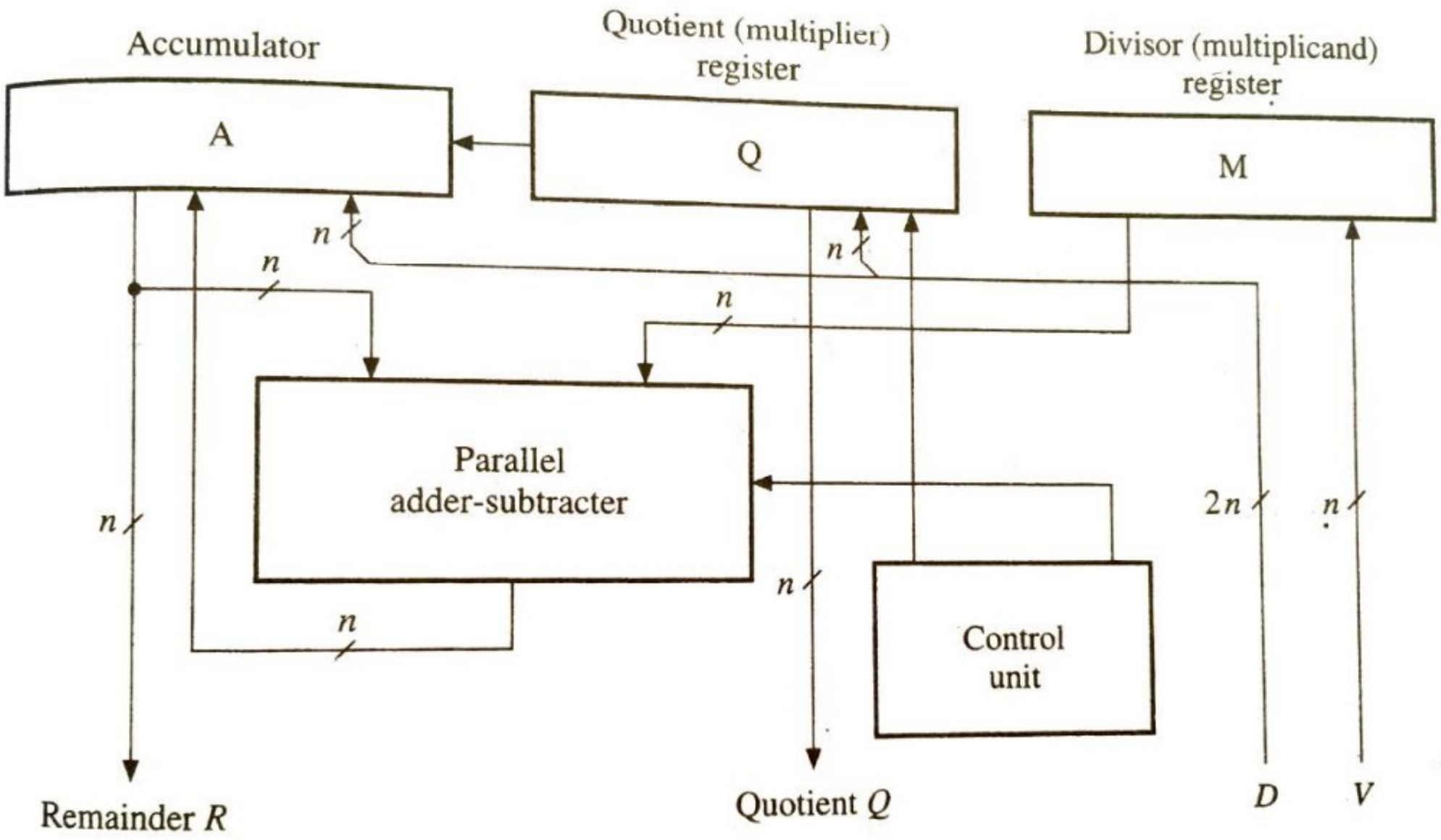
1) At each step, left shift the dividend by 1 position.

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)	