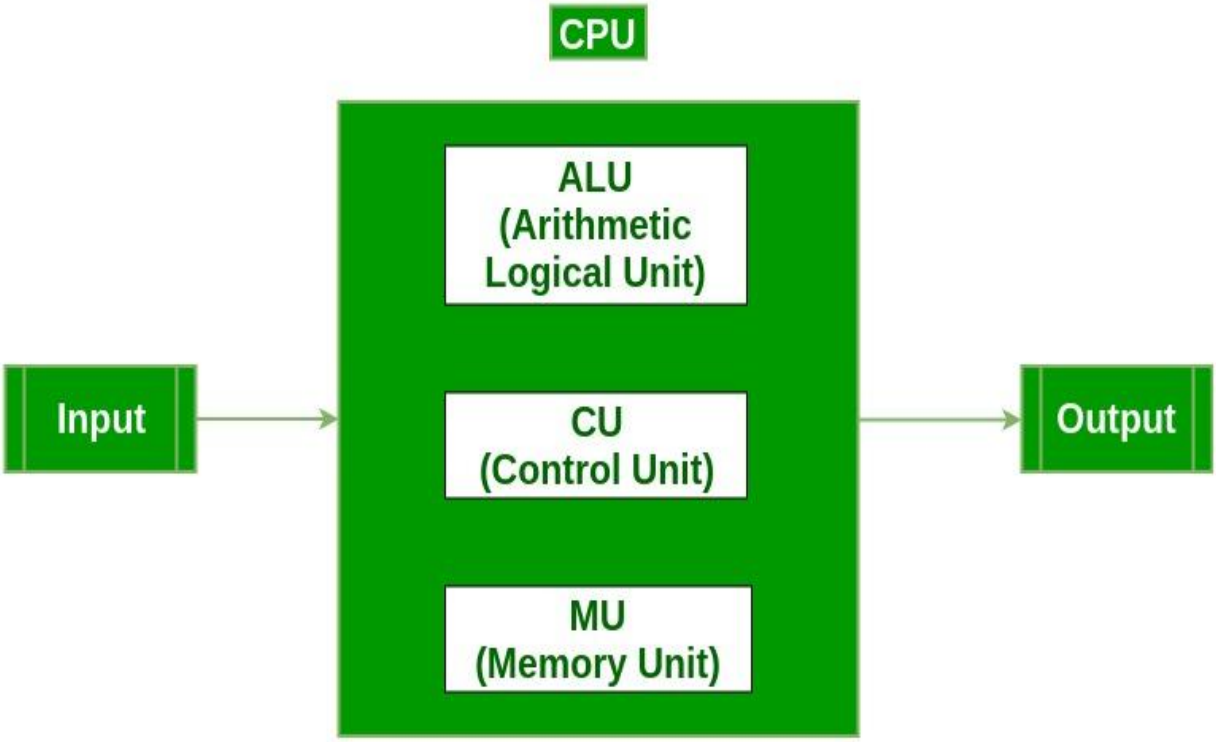
# UNIT -1

## Functional Components of a Computer

**Computer:** A computer is a combination of **hardware and software** resources which integrate together and provides various functionalities to the user. Hardware are the physical components of a computer like the processor, memory devices, monitor, keyboard etc. while software is the set of programs or instructions that are required by the hardware resources to function properly. There are a few basic components that aids the working-cycle of a computer i.e. the Input- Process- Output Cycle and these are called as the functional components of a computer. It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.

**Digital Computer:** A digital computer can be defined as a programmable machine which reads the binary data passed as instructions, processes this binary data, and displays a calculated digital output. Therefore, Digital computers are those that work on the digital data.

**Details of Functional Components of a Digital Computer**



- **Input Unit :**The input unit consists of input devices that are attached to the computer. These devices take input and convert it into binary language that the computer understands. Some of the common input devices are keyboard, mouse, joystick, scanner etc.
- **Central Processing Unit (CPU) :** Once the information is entered into the computer by the input device, the processor processes it. The CPU is called the brain of the computer because it is the control center of the computer. It first fetches instructions from memory and then interprets them so as to know what is to be done. If required, data is fetched from memory or input device. Thereafter CPU executes or performs the required computation and then either stores the output or displays on the output device. The CPU has three main components which are responsible for different functions – Arithmetic Logic Unit (ALU), Control Unit (CU) and Memory registers
- **Arithmetic and Logic Unit (ALU) :** The ALU, as its name suggests performs mathematical calculations and takes logical decisions. Arithmetic calculations include addition, subtraction, multiplication and division. Logical decisions involve comparison of two data items to see which one is larger or smaller or equal.
- **Control Unit :** The Control unit coordinates and controls the data flow in and out of CPU and also controls all the operations of ALU, memory registers and also input/output units. It is also responsible for carrying out all the instructions stored in the program. It decodes the fetched instruction, interprets it and sends control signals to input/output devices until the required operation is done properly by ALU and memory.
- **Memory Registers :** A register is a temporary unit of memory in the CPU. These are used to store the data which is directly used by the processor. Registers can be of different sizes(16 bit, 32 bit, 64 bit and so on) and each register inside the CPU has a specific function like storing data, storing an instruction, storing address of a location in memory etc. The user registers can be used by an assembly language programmer for storing operands, intermediate results etc. Accumulator (ACC) is the main register in the ALU and contains one of the operands of an operation to be performed in the ALU.
- **Memory :** Memory attached to the CPU is used for storage of data and instructions and is called internal memory The internal memory is divided into many storage locations, each of which can store data or instructions.

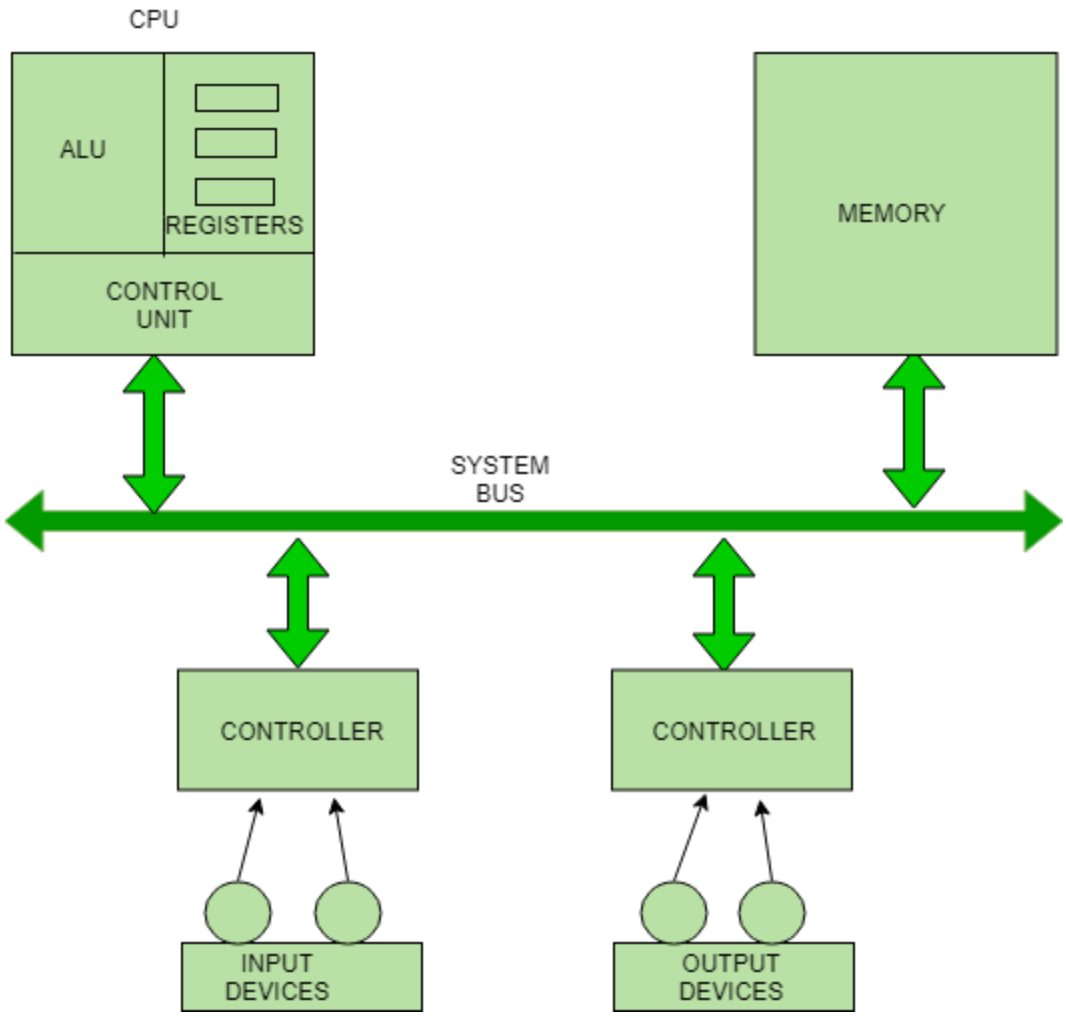
Each memory location is of the same size and has an address. With the help of the address, the computer can read any memory location easily without having to search the entire memory. when a program is executed, it's data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory. The time of access of data is independent of its location in memory, therefore this memory is also called Random Access memory (RAM). Read this for different types of RAMs.

- **Output Unit :** The output unit consists of output devices that are attached with the computer. It converts the binary data coming from CPU to human understandable form. The common output devices are monitor, printer, plotter etc.

## Interconnection between Functional Components

A computer consists of input unit that takes input, a CPU that processes the input and an output unit that produces output. All these devices communicate with each other through a common bus. A bus is a transmission path, made of a set of conducting wires over which data or information in the form of electric signals, is passed from one component to another in a computer. The bus can be of three types – Address bus, Data bus and Control Bus.

Following figure shows the connection of various functional components:

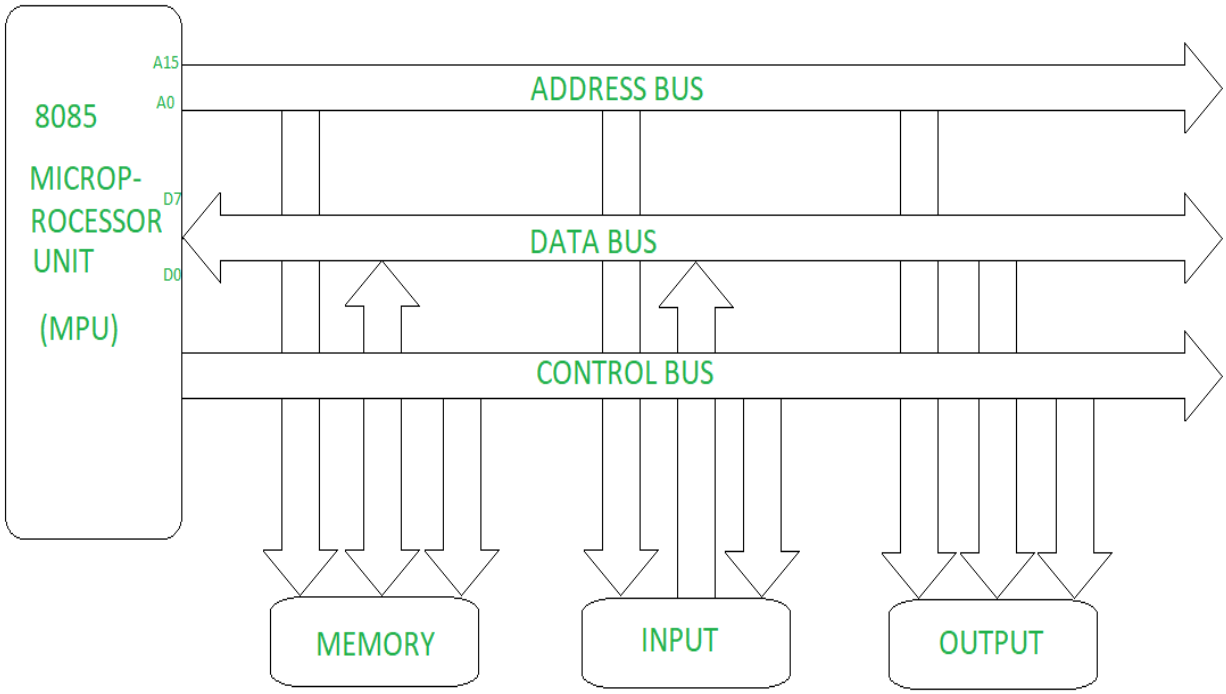



The address bus carries the address location of the data or instruction. The data bus carries data from one component to another and the control bus carries the control signals. The system bus is the common communication path that carries signals to/from CPU, main memory and input/output devices. The input/output devices communicate with the system bus through the controller circuit which helps in managing various input/output devices attached to the computer.

## Bus organization of 8085 microprocessor

Bus is a group of conducting wires which carries information, all the peripherals are connected to microprocessor through Bus.

Diagram to represent bus organization system of 8085 Microprocessor.




Bus organization system of 8085 Microprocessor

There are three types of buses.

1. **Address Bus-**
   It is a group of conducting wires which carries address only.Address bus is unidirectional because data flow in one direction, from microprocessor to memory or from microprocessor to Input/output devices (That is, Out of Microprocessor).
   Length of Address Bus of 8085 microprocessor is 16 Bit (That is, Four Hexadecimal Digits), ranging from 0000 H to FFFF H, (H denotes Hexadecimal). The microprocessor 8085 can transfer maximum 16 bit address which means it can address 65, 536 different memory location.

   The Length of the address bus determines the amount of memory a system can address.Such as a system with a 32-bit address bus can address 2^32

memory locations.If each memory location holds one byte, the addressable memory space is 4 GB.However, the actual amount of memory that can be accessed is usually much less than this theoretical limit due to chipset and motherboard limitations.

2. **Data Bus-**

   It is a group of conducting wires which carries Data only.Data bus is bidirectional because data flow in both directions, from microprocessor to memory or Input/Output devices and from memory or Input/Output devices to microprocessor.

   Length of Data Bus of 8085 microprocessor is 8 Bit (That is, two Hexadecimal Digits), ranging from 00 H to FF H. (H denotes Hexadecimal).

   When it is write operation, the processor will put the data (to be written) on the data bus, when it is read operation, the memory controller will get the data from specific memory block and put it into the data bus.

   The width of the data bus is directly related to the largest number that the bus can carry, such as an 8 bit bus can represent 2 to the power of 8 unique values, this equates to the number 0 to 255.A 16 bit bus can carry 0 to 65535.

3. **Control Bus –**

   It is a group of conducting wires, which is used to generate timing and control signals to control all the associated peripherals, microprocessor uses control bus to process data, that is what to do with selected memory location. Some control signals are:

   - Memory read
   - Memory write
   - I/O read
   - I/O Write
   - Opcode fetch

   If one line of control bus may be the read/write line.If the wire is low (no electricity flowing) then the memory is read, if the wire is high (electricity is flowing) then the memory is written.

# BUS Arbitration in Computer Organization

**Bus Arbitration** refers to the process by which the current bus master accesses and then leaves the control of the bus and passes it to the another bus requesting processor unit. The controller that has access to a bus at an instance is known as **Bus master**.

A conflict may arise if the number of DMA controllers or other controllers or processors try to access the common bus at the same time, but access can be given to only one of those. Only one processor or controller can be Bus master at the same point of time. To resolve these conflicts, Bus Arbitration procedure is implemented

to coordinate the activities of all devices requesting memory transfers. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus. The **Bus Arbiter** decides who would become current bus master.

There are two approaches to bus arbitration:

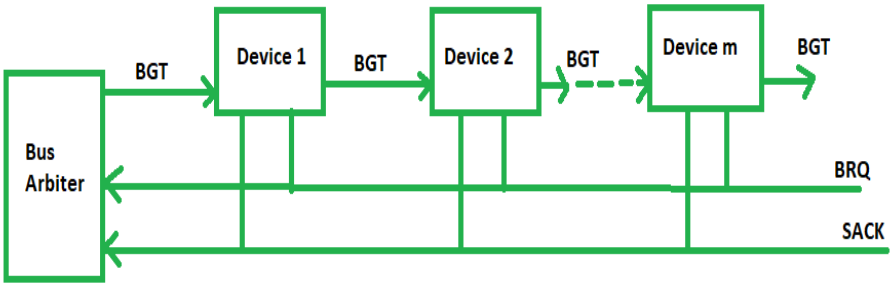
1. **Centralized bus arbitration –** A single bus arbiter performs the required arbitration.
2. **Distributed bus arbitration –** All devices participate in the selection of the next bus master.

**Methods of BUS Arbitration –**

There are three bus arbitration methods:

**(i) Daisy Chaining method –**

It is a centralized bus arbitration method. During any bus cycle, the bus master may be any device – the processor or any DMA controller unit, connected to the bus.




Daisy chained bus arbitration

**Advantages –**

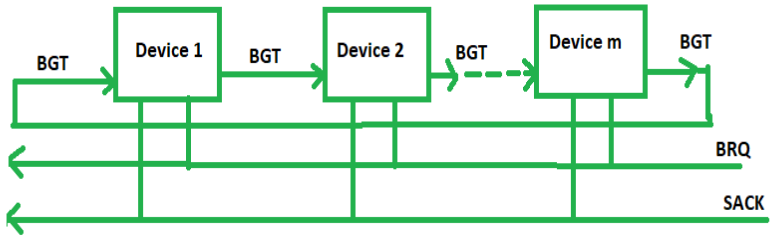
- Simplicity and Scalability.
- The user can add more devices anywhere along the chain, up to a certain maximum value.

**Disadvantages –**

- The value of priority assigned to a device is depends on the position of master bus.
- Propagation delay is arises in this method.
- If one device fails then entire system will stop working.

**(ii) Polling or Rotating Priority method –**

In this method, the devices are assigned unique priorities and complete to access the bus, but the priorities are dynamically changed to give every device an opportunity to access the bus.




Rotating priority bus arbitration

**Advantages –**

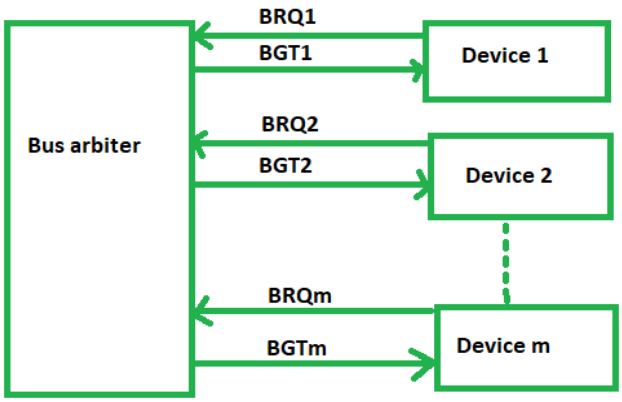
- This method does not favor any particular device and processor.
- The method is also quite simple.
- If one device fails then entire system will not stop working.

**Disadvantages –**

- Adding bus masters is different as increases the number of address lines of the circuit.

**(iii) Fixed priority or Independent Request method –**

In this method, the bus control passes from one device to another only through the centralized bus arbiter.




**Fixed priority bus arbitration method**

**Advantages –**

- This method generates fast response.

**Disadvantages –**

- Hardware cost is high as large no. of control lines are required.

# Bus and Memory Transfers

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

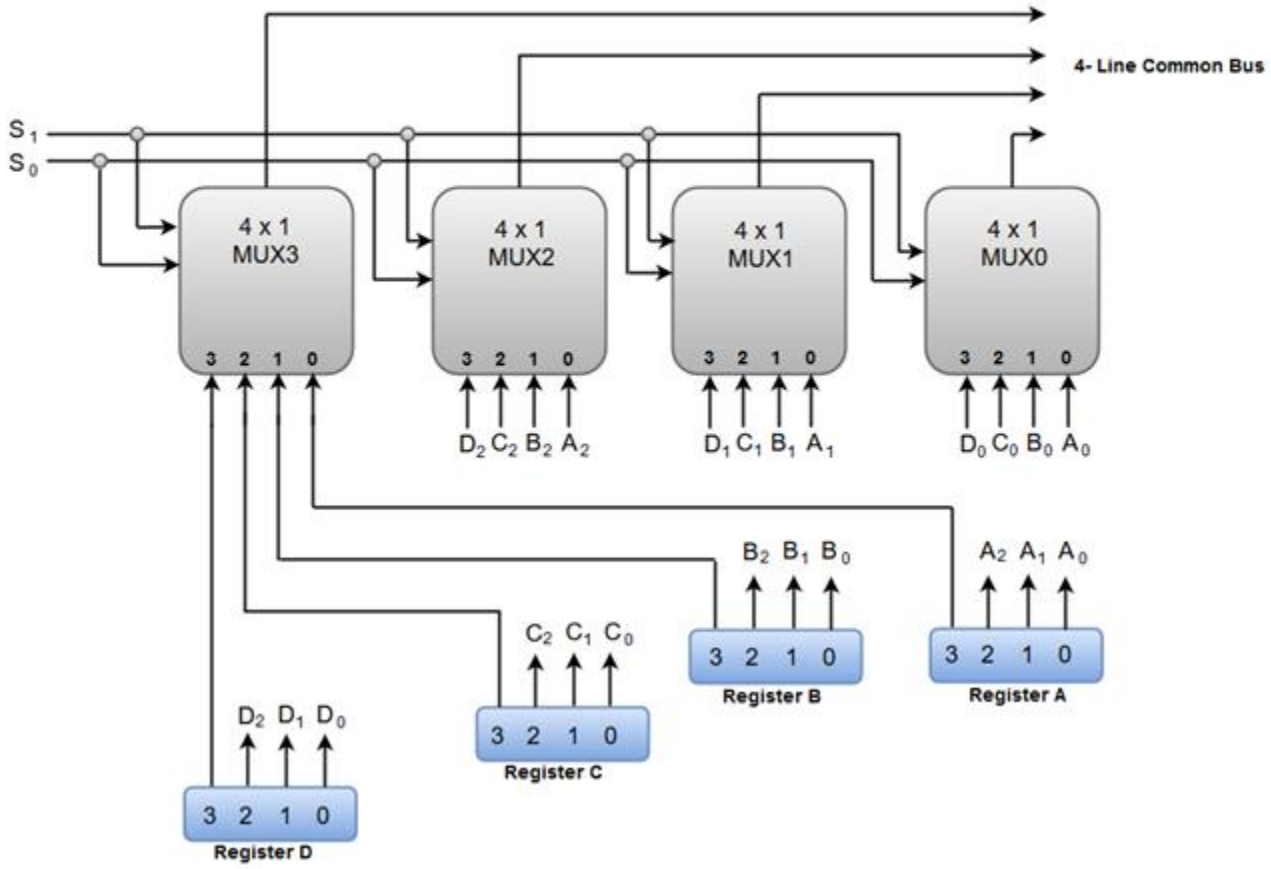
A bus structure, on the other hand, is more efficient for transferring information between registers in a multi-register configuration system.

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

The following block diagram shows a Bus system for four registers. It is constructed with the help of four 4 * 1 Multiplexers each having four data inputs (0 through 3) and two selection inputs (S1 and S2).

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

**Bus System for 4 Registers:**




The two selection lines S1 and S2 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus.

When both of the select lines are at low logic, i.e. S1S0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that forms the bus. This, in turn, causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.

Similarly, when S1S0 = 01, register B is selected, and the bus lines will receive the content provided by register B.

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

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

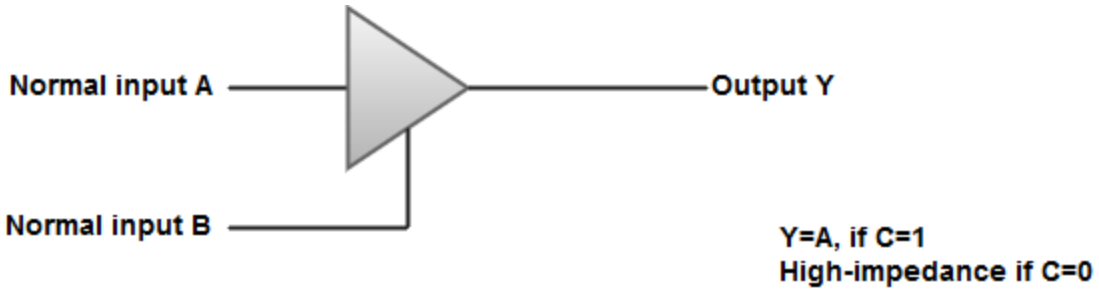
*Note: The number of multiplexers needed to construct the bus is equal to the number of bits in each register. The size of each multiplexer must be 'k * 1' since it multiplexes 'k' data lines. For instance, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in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

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

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

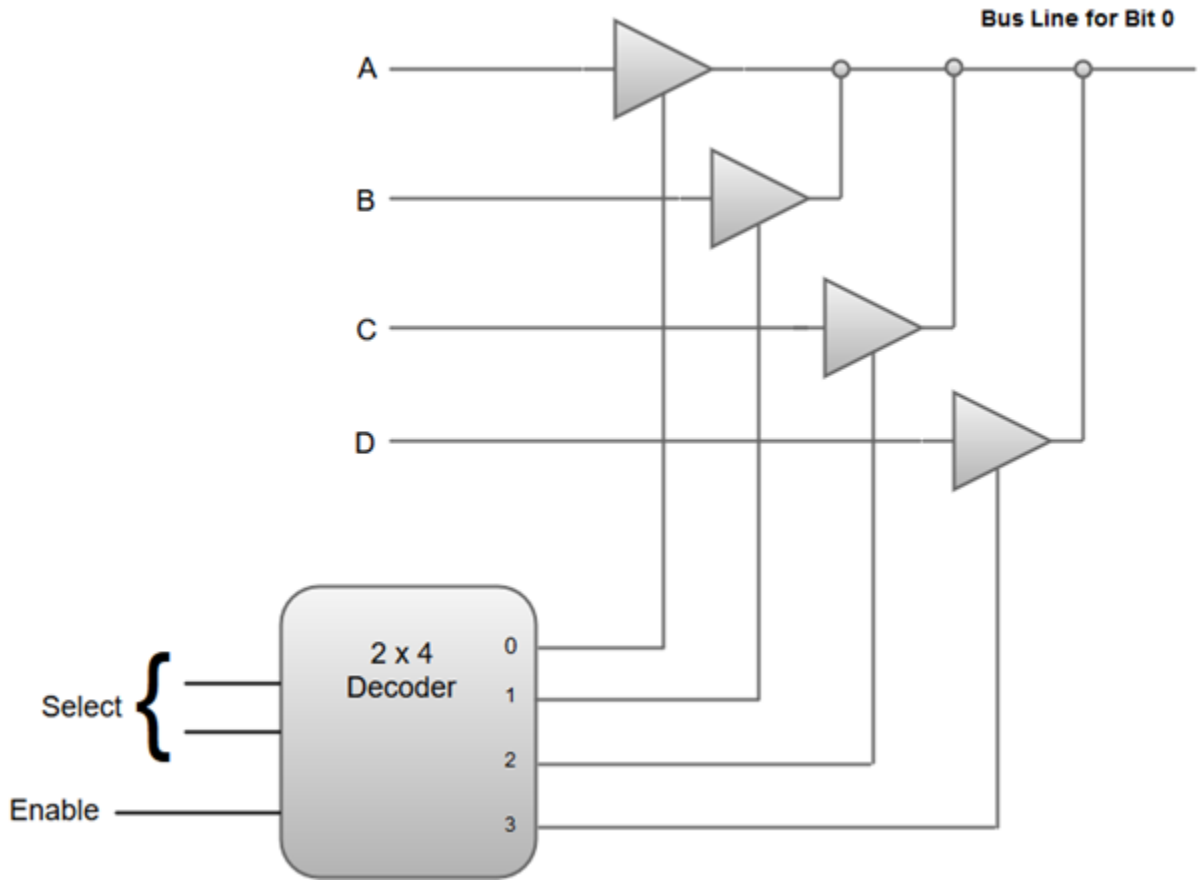
The most commonly used three state gates in case of the bus system is a **buffer gate**.

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




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

**Bus line with three state buffer:**




- The outputs generated by the four buffers are connected to form a single bus line.
- Only one buffer can be in active state at a given point of time.

- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- A 2 * 4 decoder ensures that no more than one control input is active at any given point of time.

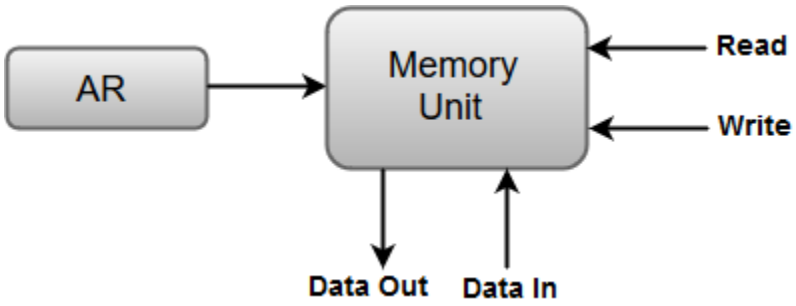
## Memory Transfer

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

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

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

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



# Processor organization (CPU Organization)-

There are following two types of processor (CPU) organization used in general :

1. Stack based organization
2. Register based organization

## Stack based CPU Organization

The computers which use Stack-based CPU Organization are based on a data structure called **stack**. The stack is a list of data words. It uses **Last In First Out (LIFO)** access method which is the most popular access method in most of the CPU. A register is used to store the address of the topmost element of the stack which is known as **Stack pointer (SP)**. In this organisation, ALU operations are performed on stack data. It means both the operands are always required on the stack. After manipulation, the result is placed in the stack.
The main two operations that are performed on the operators of the stack are **Push** and **Pop**. These two operations are performed from one end only.

**1 .Push –**
This operation results in inserting one operand at the top of the stack and it decrease the stack pointer register. The format of the PUSH instruction is:

```
PUSH
```

It inserts the data word at specified address to the top of the stack. It can be implemented as:

```
//decrement SP by 1
SP <-- SP - 1


//store the content of specified memory address
//into SP; i.e, at top of stack
SP <-- (memory address)
```

1. **Pop –**
   This operation results in deleting one operand from the top of the stack and it increase the stack pointer register. The format of the POP instruction is:

```
POP
```

   It deletes the data word at the top of the stack to the specified address. It can be implemented as:

```
//transfer the content of  SP (i.e, at top most data)
//into specified memory location
(memory address) <-- SP


//increment SP by 1
SP <-- SP + 1
```

   Operation type instruction does not need the address field in this CPU organization. This is because the operation is performed on the two operands that are on the top of the stack. For example:

```
SUB
```

   This instruction contains the opcode only with no address field. It pops the two top data from the stack, subtracting the data, and pushing the result into the stack at the top.

   **PDP-11, Intel's 8085 and HP 3000** are some of the examples of the stack organized computers.

   **The advantages of Stack based CPU organization –**

   - Efficient computation of complex arithmetic expressions.
   - Execution of instructions is fast because operand data are stored in consecutive memory locations.
   - Length of instruction is short as they do not have address field.

   **The disadvantages of Stack based CPU organization –**

   - The size of the program increases.

   **Note:***Stack based CPU organisation uses zero address instruction.*

# General Register based CPU Organization

When we are using multiple general purpose registers, instead of single accumulator register, in the CPU Organization then this type of organization is known as General register based CPU Organization. In this type of organization, computer uses two or three address fields in their instruction format. Each address field may specify a general register or a memory word.If many CPU registers are available for heavily used variables and intermediate results, we can avoid memory references much of the time, thus vastly increasing program execution speed, and reducing program size.

For example:

```
MULT R1, R2, R3
```

This is an instruction of an arithmetic multiplication written in assembly language. It uses three address fields R1, R2 and R3. The meaning of this instruction is:

```
R1 <-- R2 * R3
```

This instruction also can be written using only two address fields as:

```
MULT R1, R2
```

In this instruction, the destination register is the same as one of the source registers. This means the operation

```
R1 <-- R1 * R2
```

The use of large number of registers results in short program with limited instructions.

Some examples of General register based CPU Organization are **IBM 360 and PDP- 11**.

**The advantages of General register based CPU organization –**

- Efficiency of CPU increases as there are large number of registers are used in this organization.
- Less memory space is used to store the program since the instructions are written in compact way.

**The disadvantages of General register based CPU organization –**

- Care should be taken to avoid unnecessary usage of registers. Thus, compilers need to be more intelligent in this aspect.
- Since large number of registers are used, thus extra cost is required in this organization.

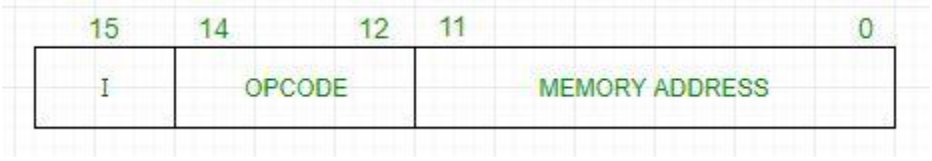
**General register CPU organization of two type:**

1. **Register-memory reference architecture (CPU with less register)**– In this organization Source 1 is always required in register, source 2 can be present either in register or in memory. Here two address instruction format is the compatible instruction format.
2. **Register-register reference architecture(CPU with more register)–** In this organization ALU operations are performed only on a register data. So operands are required in the register. After manipulation result is also placed in register. Here three address instruction format is the compatible instruction format.

# Basic Computer Instructions

The basic computer has 16-bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

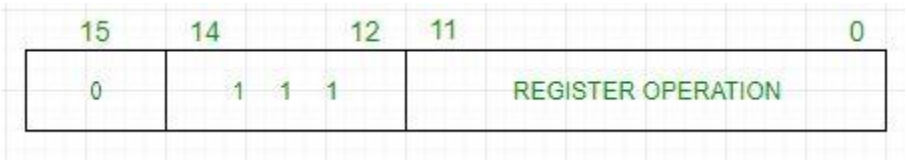
1. **Memory Reference –** These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12-bit address, 3-bit opcode (other than 111) and 1-bit addressing mode for direct and indirect addressing.




**Example –**
IR register contains = 0001XXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

```
Hence, DR ← M[AR]
AC ← AC + DR, SC ← 0
```

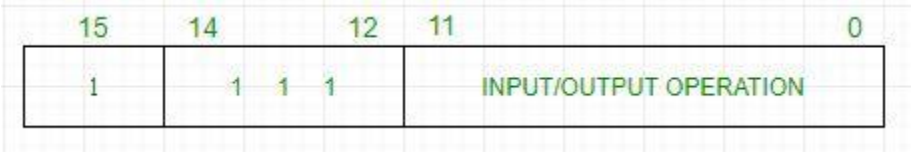
2. **Register Reference –** These instructions perform operations on registers rather than memory addresses. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.




**Example –**
IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.

```
Hence, AC ← ~AC
```

3. **Input/Output –** These instructions are for communication between computer and outside environment. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.




**Example –**
IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputing character. Hence, INPUT character from peripheral device.

The set of instructions incorporated in16 bit IR register are:

1. Arithmetic, logical and shift instructions (and, add, complement, circulate left, right, etc)
2. To move information to and from memory (store the accumulator, load the accumulator)
3. Program control instructions with status conditions (branch, skip)
4. Input output instructions (input character, output character)

| SYMBOL | HEXADECIMAL CODE | | DESCRIPTION |
|---|---|---|---|
| AND | 0xxx | 8xxx | And memory word to AC |
| ADD | 1xxx | 9xxx | Add memory word to AC |
| LDA | 2xxx | Axxx | Load memory word to AC |
| STA | 3xxx | Bxxx | Store AC content in memory |
| BUN | 4xxx | Cxxx | Branch Unconditionally |
| BSA | 5xxx | Dxxx | Branch and Save Return Address |
| ISZ | 6xxx | Exxx | Increment and skip if 0 |
| CLA | 7800 | | Clear AC |
| CLE | 7400 | | Clear E(overflow bit) |
| CMA | 7200 | | Complement AC |
| CME | 7100 | | Complement E |
| CIR | 7080 | | Circulate right AC and E |
| CIL | 7040 | | Circulate left AC and E |
| INC | 7020 | | Increment AC |
| SPA | 7010 | | Skip next instruction if AC > 0 |

| SYMBOL | HEXADECIMAL CODE | DESCRIPTION |
|---|---|---|
| SNA | 7008 | Skip next instruction if AC < 0 |
| SZA | 7004 | Skip next instruction if AC = 0 |
| SZE | 7002 | Skip next instruction if E = 0 |
| HLT | 7001 | Halt computer |
| INP | F800 | Input character to AC |
| OUT | F400 | Output character from AC |
| SKI | F200 | Skip on input flag |
| SKO | F100 | Skip on output flag |
| ION | F080 | Interrupt On |
| IOF | F040 | Interrupt Off |

# Addressing Modes

**Addressing Modes–** The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

**Addressing modes for 8086 instructions are divided into two categories:**
1) Addressing modes for data

2) Addressing modes for branch

The 8086 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. The key to good assembly language programming is the proper use of memory addressing modes.

An assembly language program instruction consists of two parts


The memory address of an operand consists of two components:
**IMPORTANT TERMS**

- **Starting address** of memory segment.
- **Effective address or Offset**: An offset is determined by adding any combination of three address elements: **displacement, base and index.**
  - **Displacement:** It is an 8 bit or 16 bit immediate value given in the instruction.
  - **Base**: Contents of base register, BX or BP.
  - **Index**: Content of index register SI or DI.

According to different ways of specifying an operand by 8086 microprocessor, different addressing modes are used by 8086.

**Addressing modes** used by 8086 microprocessor are discussed below:

- **Implied mode:**: In implied addressing the operand is specified in the instruction itself. In this mode the data is 8 bits or 16 bits long and data is the part of instruction.Zero address instruction are designed with implied addressing mode.

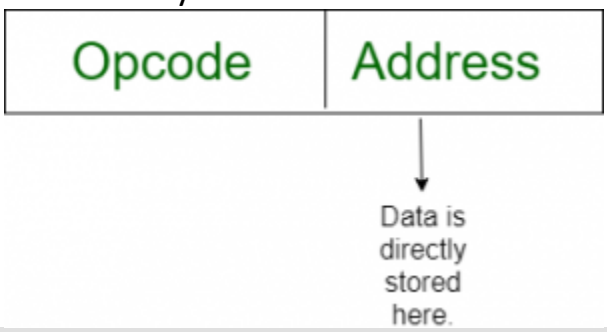
  **Instruction**

  ```
  Example:  CLC (used to reset Carry flag to 0)
  ```

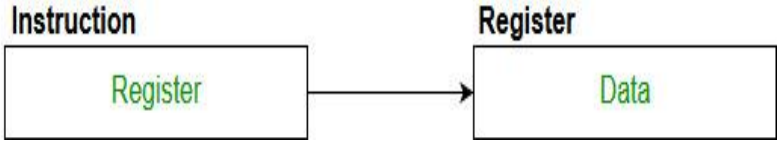
- **Immediate addressing mode :**In this mode data is present in address field of instruction .Designed like one address instruction format.

- **Note:**Limitation in the immediate mode is that the range of constants are restricted by size of address field.




```
Example:  MOV AL, 35H (move the data 35H into AL register)
```

- **Register mode:** In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction.
  *Here one register reference is required to access the data.*




```
Example: MOV AX,CX (move the contents of CX register to AX register)
```

- **Register Indirect mode**: In this addressing the operand's offset is placed in any one of the registers BX,BP,SI,DI as specified in the instruction. The effective address of the data is in the base register or an index register that is specified by the instruction.
  *Here two register reference is required to access the data.*


The 8086 CPUs let you access memory indirectly through a register using the register indirect addressing modes.

- `MOV AX, [BX](move the contents of memory location s`

```
addressed by the register BX to the register AX)
```

- **Auto Indexed (increment mode)**: Effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location.**(R1)+**.
  *Here one register reference, one memory reference and one ALU operation is*

*required to access the data.*

Example:

```
Add R1, (R2)+  // OR
R1 = R1 +M[R2]
R2 = R2 + d
```

*Useful for stepping through arrays in a loop. R2 – start of array d* – size of an element

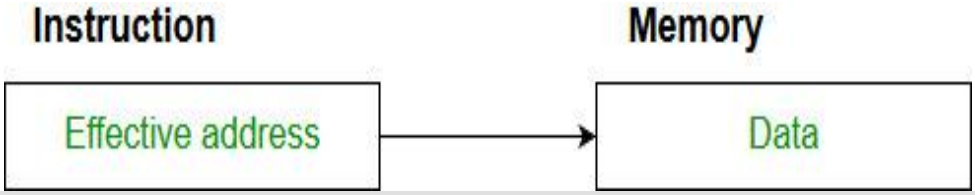
- **Auto indexed ( decrement mode)**: Effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location. **–(R1)**
  *Here one register reference,one memory reference and one ALU operation is required to access the data.*

**Example:**

```
Add R1,-(R2)   //OR
R2 = R2-d
R1 = R1 + M[R2]
```

*Auto decrement mode is same as auto increment mode. Both can also be used to implement a stack as push and pop . Auto increment and Auto decrement modes are useful for implementing "Last-In-First-Out" data structures.*

- **Direct addressing/ Absolute addressing Mode (symbol [ ]):** The operand's offset is given in the instruction as an 8 bit or 16 bit displacement element. In this addressing mode the 16 bit effective address of the data is the part of the instruction.
  *Here only one memory reference operation is required to access the data.*




```
Example:ADD AL,[0301]   //add the contents of offset address 0301 to AL
```

- **Indirect addressing Mode (symbol @ or () )**:In this mode address field of instruction contains the address of effective address.Here two references are required.
  1st reference to get effective address.
  2nd reference to access the data.
  Based on the availability of Effective address, Indirect mode is of two kind:

1. Register Indirect:In this mode effective address is in the register, and corresponding register name will be maintained in the address field of an instruction.
   *Here one register reference,one memory reference is required to access the data.*
2. Memory Indirect:In this mode effective address is in the memory, and corresponding memory address will be maintained in the address field of an instruction.
   *Here two memory reference is required to access the data.*

- **Indexed addressing mode**: The operand's offset is the sum of the content of an index register SI or DI and an 8 bit or 16 bit displacement.

```
Example:MOV AX, [SI +05]
```

- **Based Indexed Addressing:** The operand's offset is sum of the content of a base register BX or BP and an index register SI or DI.

```
Example: ADD AX, [BX+SI]
```

**Based on Transfer of control, addressing modes are:**

- **PC relative addressing mode:** PC relative addressing mode is used to implement intra segment transfer of control, In this mode effective address is obtained by adding displacement to PC.

```
EA= PC + Address field value
```

```
PC= PC + Relative value.
```

- **Base register addressing mode:**Base register addressing mode is used to implement inter segment transfer of control.In this mode effective address is obtained by adding base register value to address field value.

```
EA= Base register + Address field value.
```

```
PC= Base register + Relative value.
```

**Note:**

1. PC relative nad based register both addressing modes are suitable for program relocation at runtime.
2. Based register addressing mode is best suitable to write position independent codes.

**Advantages of Addressing Modes**

1.To give programmers to facilities such as Pointers, counters for loop controls, indexing of data and program relocation.

2.To reduce the number bits in the addressing field of the Instruction.

**Sample Question**

Match each of the high level language statements given on the left hand side with the most natural addressing mode from those listed on the right hand side.

```
1. A[1] = B[J];          a. Indirect addressing

2. while [*A++];         b. Indexed  addressing

3. int temp = *x;        c. Autoincrement
```

**(A**) (1, c), (2, b), (3, a)
**(B)** (1, a), (2, c), (3, b)
**(C)** (1, b), (2, c), (3, a)
**(D)** (1, a), (2, b), (3, c)

**Answer: (C)**

**Explanation:**

```
List 1                              List 2

1) A[1] = B[J];       b) Index addressing

Here indexing is used


2) while [*A++];      c) auto increment

The memory locations are automatically incremented


3) int temp = *x;     a) Indirect addressing

Here temp is assigned the value of int type stored

at the address contained in X
```

Hence (C) is correct solution.

# Binary Adder-Subtractor

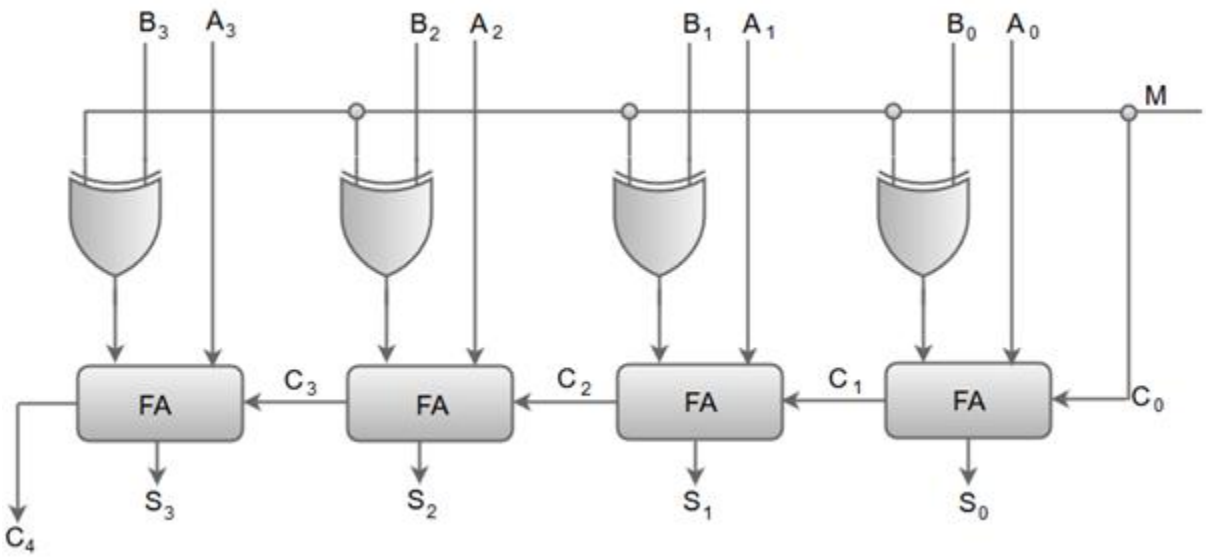
The Subtraction micro-operation can be done easily by taking the 2's compliment of addend bits and adding it to the augend bits.

> *Note: The 2's compliment can be obtained by taking the 1's compliment and adding one to the least significant pair of bits. The 1's compliment can be implemented with inverters, and one can be added to the sum through the input carry.*

The Arithmetic micro-operations like addition and subtraction can be combined into one common circuit by including an exclusive-OR gate with each full adder.

The block diagram for a 4-bit adder-subtractor circuit can be represented as:

**4 bit adder-subtractor:**




- When the mode input (M) is at a low logic, i.e. '0', the circuit act as an adder and when the mode input is at a high logic, i.e. '1', the circuit act as a subtractor.
- The exclusive-OR gate connected in series receives input M and one of the inputs B.

- When M is at a low logic, we have B⊕ 0 = B.
  The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B.
- When M is at a high logic, we have B⊕ 1 = B' and C0 = 1.
  The B inputs are complemented, and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.

# Multiplication Algorithm in Signed Magnitude Representation

Multiplication of two fixed point binary number in *signed magnitude representation* is done with process of *successive shift* and *add operation*.

```
   10111 (Multiplicand)
x  10011 (Multiplier)
  ------
   10111
   10111
  00000
 00000
10111
---------
011011010    (Product)
---------
```

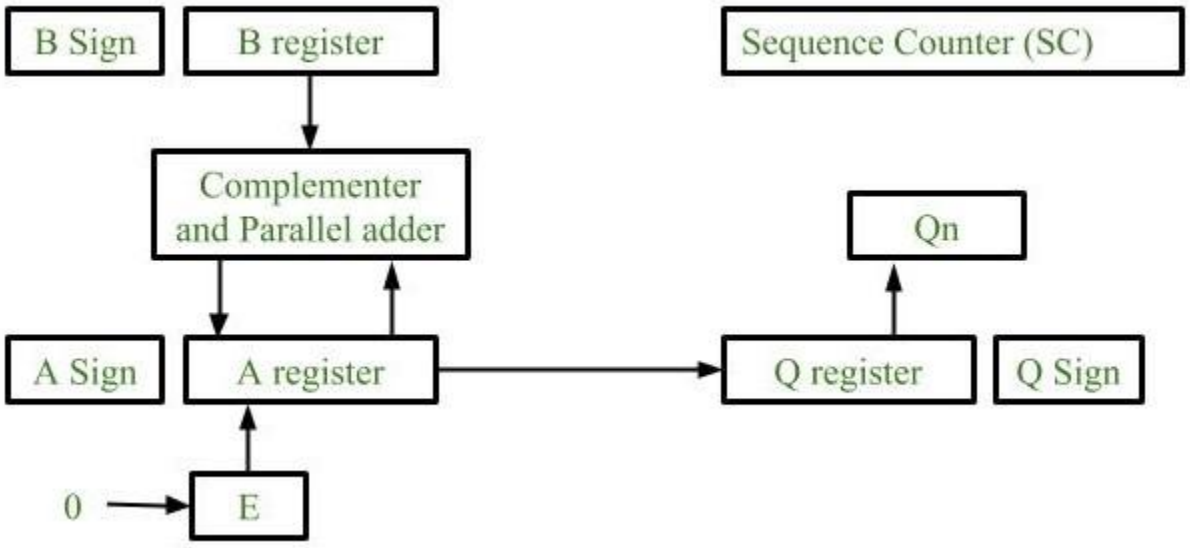
In the multiplication process we are considering successive bits of the multiplier, least significant bit first.
If the multiplier bit is 1, the multiplicand is copied down else 0's are copied down.

The numbers copied down in successive lines are shifted one position to the left from the previous number.
Finally numbers are added and their sum form the product.

The sign of the product is determined from the sign of the multiplicand and multiplier. If they are alike, sign of the product is positive else negative.

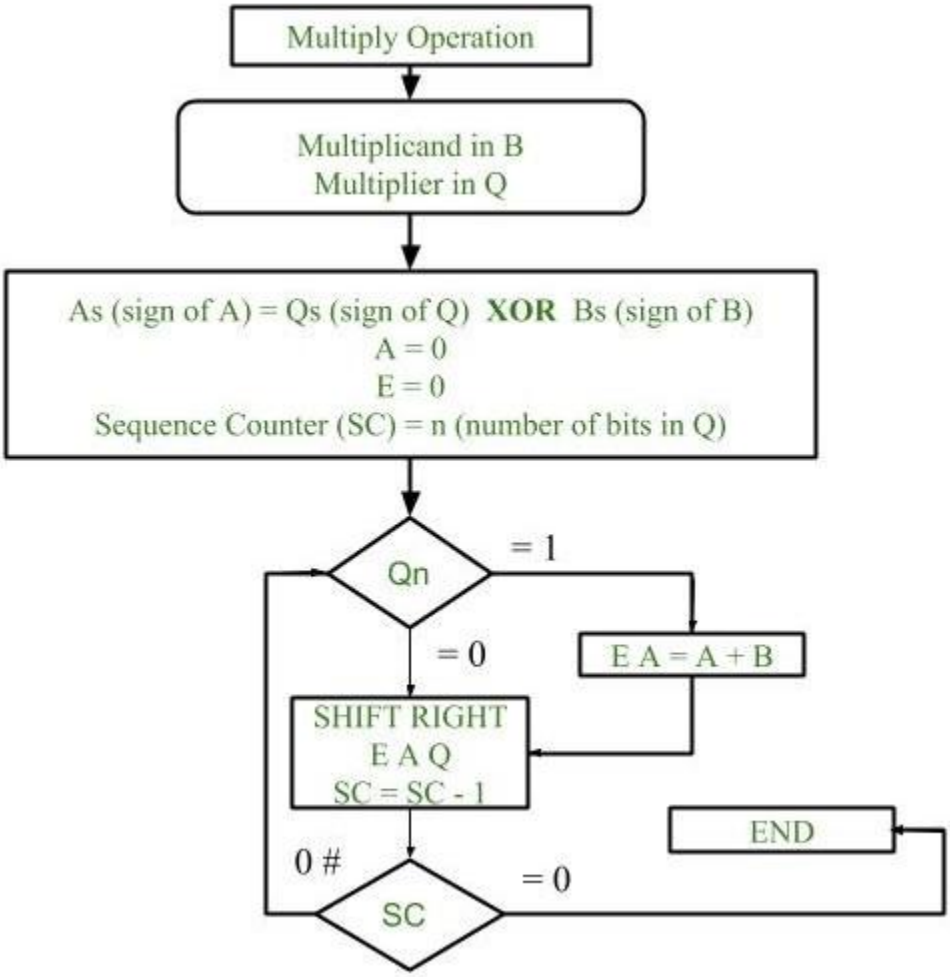
**Hardware Implementation :**

Following components are required for the *Hardware Implementation* of multiplication algorithm :




1. **Registers:**
   Two Registers B and Q are used to store multiplicand and multiplier respectively.
   Register A is used to store partial product during multiplication.
   Sequence Counter register (SC) is used to store number of bits in the multiplier.
2. **Flip Flop:**
   To store sign bit of registers we require three flip flops (A sign, B sign and Q sign).
   Flip flop E is used to store carry bit generated during partial product addition.
3. **Complement and Parallel adder:**
   This hardware unit is used in calculating partial product i.e, perform addition required.

**Flowchart of Multiplication:**




1. Initially multiplicand is stored in B register and multiplier is stored in Q register.
2. Sign of registers B (Bs) and Q (Qs) are compared using **XOR** functionality (i.e., if both the signs are alike, output of XOR operation is 0 unless 1) and output stored in As (sign of A register).
   **Note:** Initially 0 is assigned to register A and E flip flop. Sequence counter is initialized with value n, n is the number of bits in the Multiplier.
3. Now least significant bit of multiplier is checked. If it is 1 add the content of register A with Multiplicand (register B) and result is assigned in A register with carry bit in flip flop E. Content of E A Q is shifted to right by one position, i.e., content of E is shifted to most significant bit (MSB) of A and least significant bit of A is shifted to most significant bit of Q.
4. If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion.

5. Content of Sequence counter is decremented by 1.
6. Check the content of Sequence counter (SC), if it is 0, end the process and the final product is present in register A and Q, else repeat the process.

**Example:**

```
Multiplicand = 10111

Multiplier = 10011
```

| Multiplicand B = 10111 | E | A | Q | SC |
|---|---|---|---|---|
| Multiplier in Q | 0 | 00000 | 10011 | 101 |
| Qn = 1; add B | | 10111 | | |
| First partial product | 0 | 10111 | | |
| Shift right EAQ | 0 | 01011 | 11001 | 100 |
| Qn = 1; add B | | 10111 | | |
| Second partial product | 1 | 00010 | | |
| Shift right EAQ | 0 | 10001 | 01100 | 011 |
| Qn = 0; shift right EAQ | 0 | 01000 | 10110 | 010 |
| Qn = 0; shift right EAQ | 0 | 00100 | 01011 | 001 |
| Qn = 1; add B | | 10111 | | |
| Fifth partial product | 0 | 11011 | | |
| Shift right EAQ | 0 | 01101 | 10101 | 000 |

Final product in AQ
0110110101

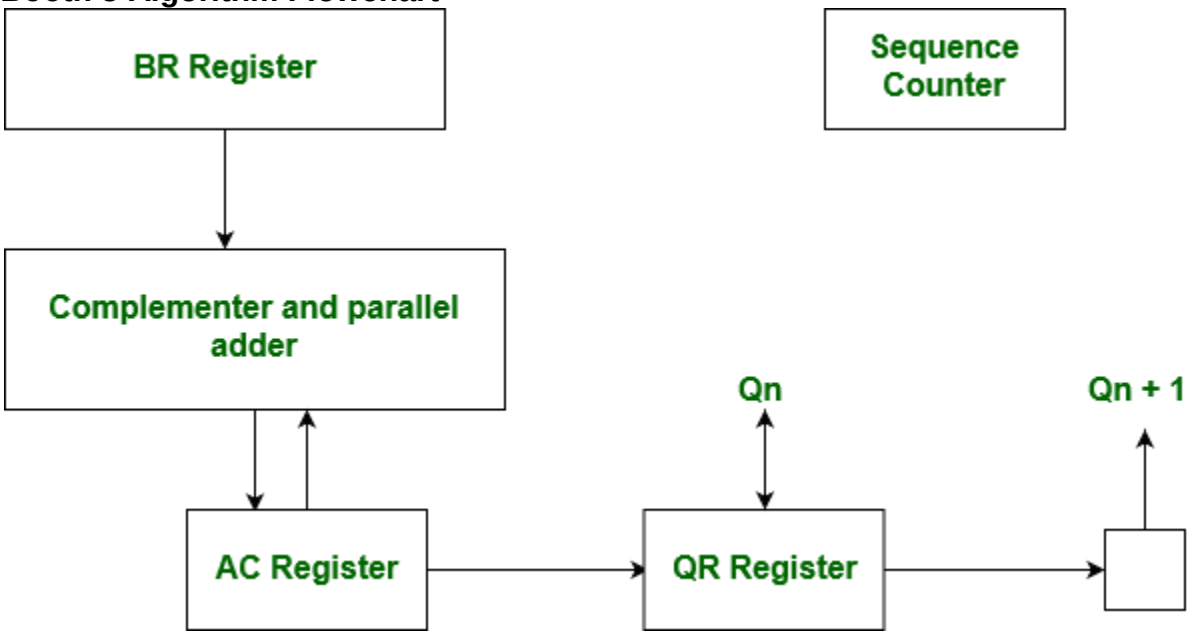
# Booth's Algorithm of Multiplication-

Booth algorithm gives a procedure for **multiplying binary integers** in signed 2's complement representation **in efficient way**, i.e., less number of additions/subtractions required. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^(k+1 ) to 2^m.
As in all multiplication schemes, booth algorithm requires examination **of the multiplier bits** and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to following rules:

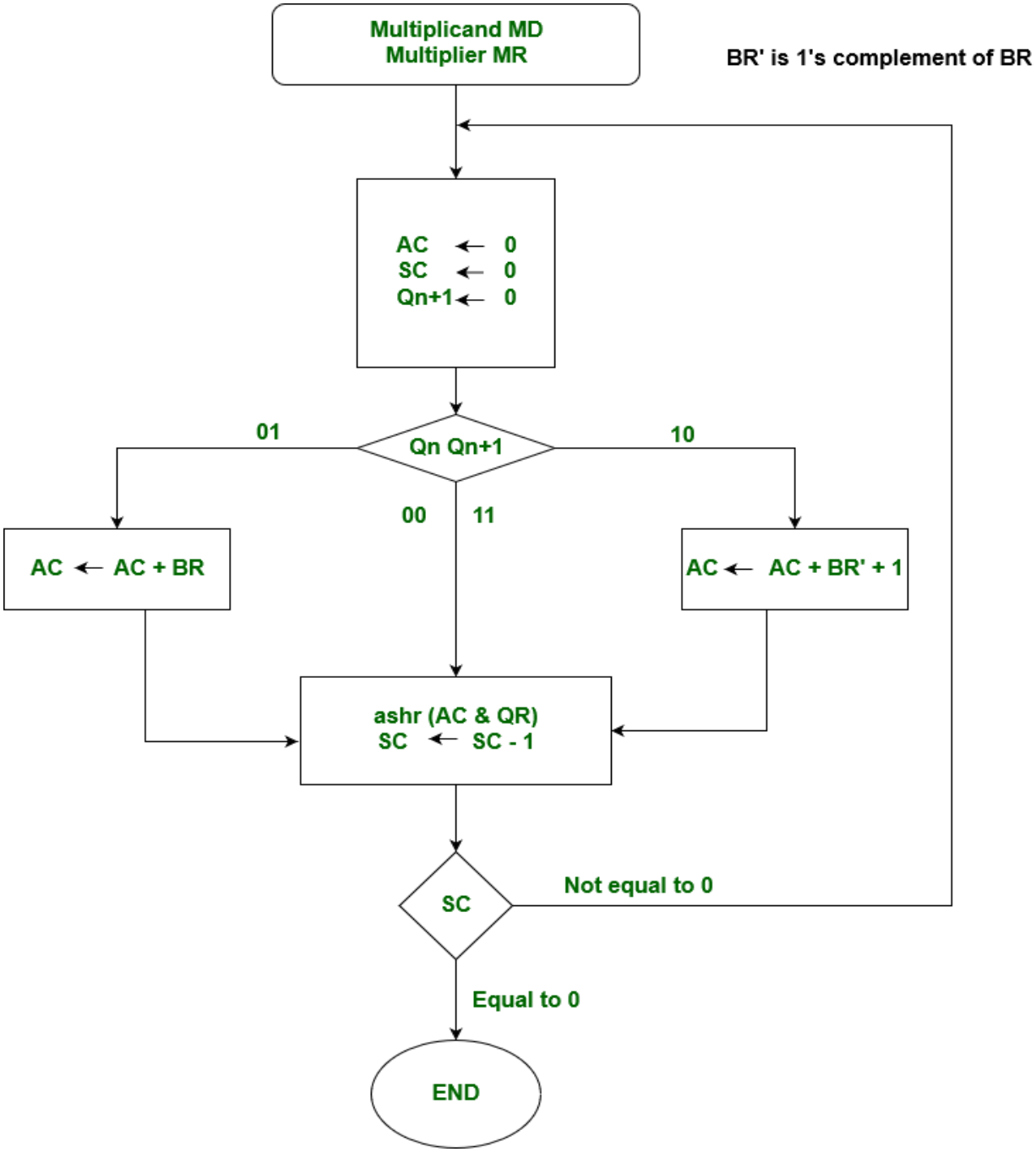
1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous '1') in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

**Hardware Implementation of Booths Algorithm –** The hardware implementation of the booth algorithm requires the register configuration shown in the figure below.

**Booth's Algorithm Flowchart –**



We name the register as A, B and Q, AC, BR and QR respectively. Qn designates the least significant bit of multiplier in the register QR. An extra flip-flop Qn+1is appended to QR to facilitate a double inspection of the multiplier.The flowchart for the booth algorithm is shown below.




AC and the appended bit Qn+1 are initially cleared to 0 and the sequence SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1are inspected. If the two bits are equal to 10, it means

that the first 1 in a string has been encountered. This requires subtraction of the multiplicand from the partial product in AC. If the 2 bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.

When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the 2 numbers that are added always have a opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including Qn+1). This is an arithmetic shift right (ashr) operation which AC and QR ti the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

**Example –** A numerical example of booth's algorithm is shown below for n = 4. It shows the step by step multiplication of -5 and -7.

```
MD = -5 = 1011, MD = 1011, MD'+1 = 0101

MR = -7 = 1001

The explanation of first step is as follows: Qn+1

AC = 0000, MR = 1001, Qn+1 = 0,  SC = 4

Qn Qn+1 = 10

So, we do AC + (MD)'+1, which gives AC = 0101

On right shifting AC and MR, we get

AC = 0010, MR = 1100 and Qn+1 = 1
```

Product is calculated as follows:

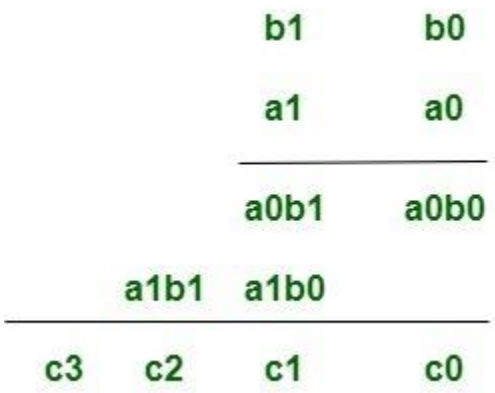
```
Product = AC MR

Product = 0010 0011 =  35
```

# Array Multiplier -

An **array multiplier** is a digital combinational circuit used for multiplying two binary numbers by employing an array of full adders and half adders. This array is used for the nearly simultaneous addition of the various product terms involved. To form the various product terms, an array of AND gates is used before the Adder array.
Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once. This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

For implementation of array multiplier with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in figure. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is

```
c3c2c1c0
```




Assuming A = a1a0 and B= b1b0, the various bits of the final product term P can be written as:-
1. P(0)= a0b0

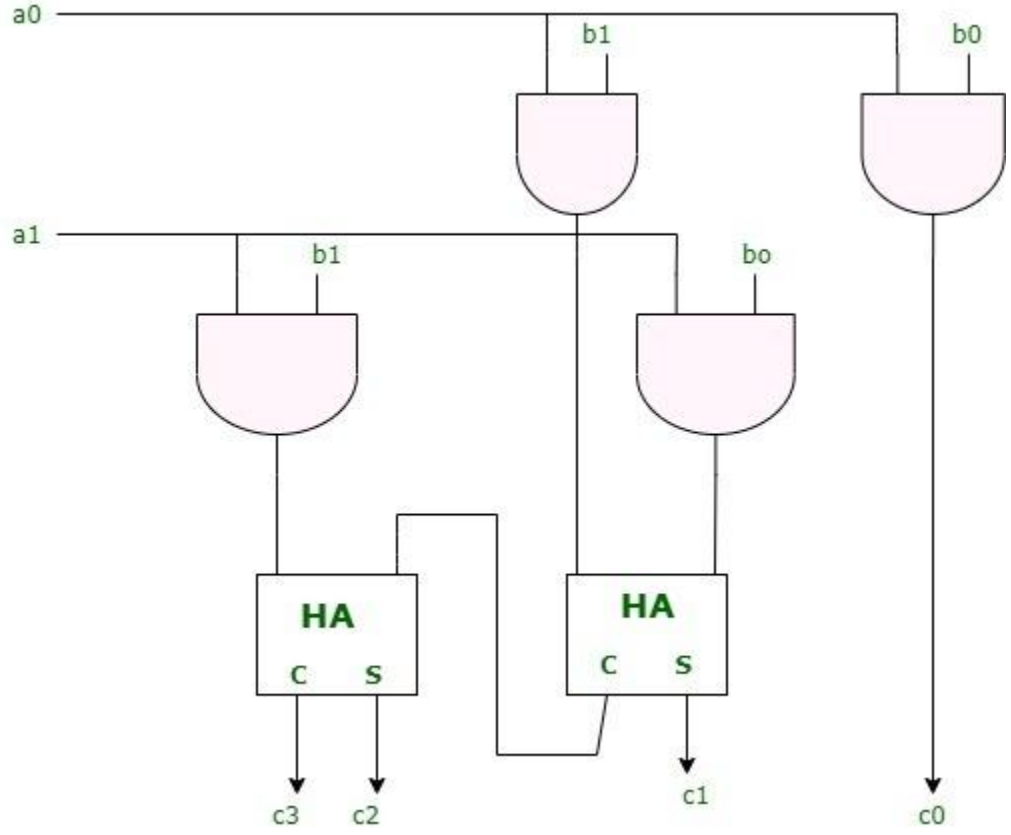
2. P(1)=a1b0 + b1a0

3. P(2) = a1b1 + c1 where c1 is the carry generated during the addition for the P(1) term.

4. P(3) = c2 where c2 is the carry generated during the addition for the P(2) term.

For the above multiplication, an array of four AND gates is required to form the various product terms like a0b0 etc. and then an adder array is required to calculate the sums involving the various product terms and carry combinations mentioned in the above equations in order to get the final Product bits.

1. The first partial product is formed by multiplying a0 by b1, b0. The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces 0. This is identical to an AND operation and can be implemented with an AND gate.
2. The first partial product is formed by means of two AND gates.
3. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left.
4. The above two partial products are added with two half-adder(HA) circuits. Usually there are more bits in the partial products and it will be necessary to use full-adders to produce the sum.
5. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

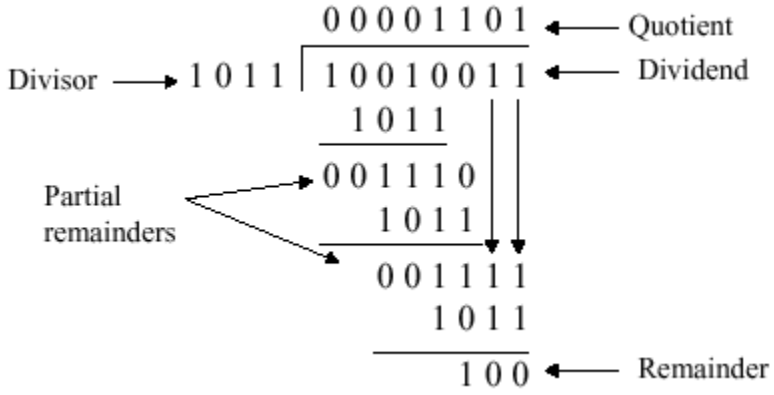



A combinational circuit binary multiplier with more bits can be constructed in similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand

in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand we need **j*k** AND gates and **(j-1)** k-bit adders to produce a product of **j+k** bits.

### Design of Arithmetic Division Hardware

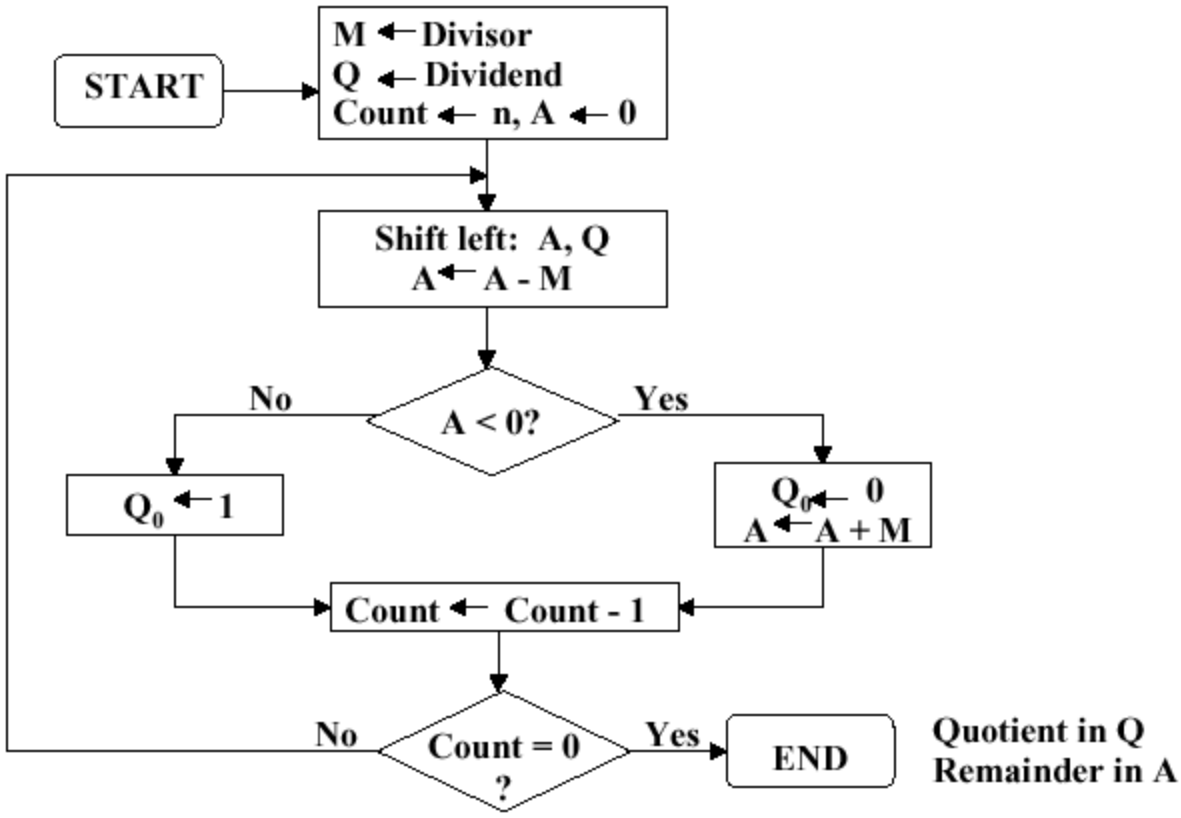
Division is a similar operation to multiplication, especially when implemented using a procedure similar to the algorithm shown in Figure 3.18a. For example, consider the pencil–and–paper method for dividing the byte 10010011 by the nybble 1011:




The governing equation is as follows:

$$\text{Dividend} = \text{Quotient} \cdot \text{Divisor} + \text{Remainder}\ .$$

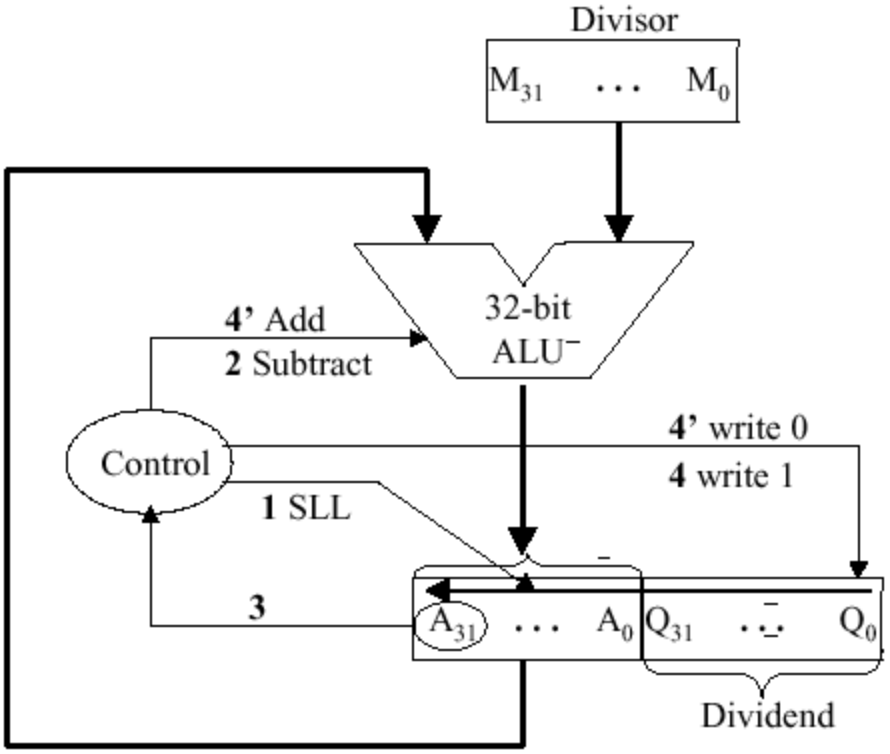
**Unsigned Division** – The *unsigned* division algorithm that is similar to Booth's algorithm is shown in Figure 3.19a, with an example shown in Figure 3.19b. The ALU schematic diagram in given in Figure 3.19c. The analysis of the algorithm and circuit is very similar to the preceding discussion of Booth's algorithm.




(a)

| A | Q | M = 0011 | |
|---|---|---|---|
| 0000 | 0111 | Initial values | |
| 0000 | 1110 | Shift | 1 |
| 1101 | | A = A - M | |
| 0000 | 1110 | A = A + M | |
| 0001 | 1100 | Shift | 2 |
| 1110 | | A = A - M | |
| 0001 | 1100 | A = A + M | |
| 0011 | 1000 | Shift | 3 |
| 0000 | | A = A - M | |
| 0000 | 1001 | $Q_0 = 1$ | |
| 0001 | 0010 | Shift | 4 |
| 1110 | | A = A - M | |
| 0001 | 0010 | A = A + M | |

(b)




(c)

**Figure 3.19.** Division of 32-bit Boolean number representations: (a) algorithm, (b) example using division of the unsigned integer 7 by the unsigned integer 3, and (c) schematic diagram of ALU circuitry - adapted from [Maf01].

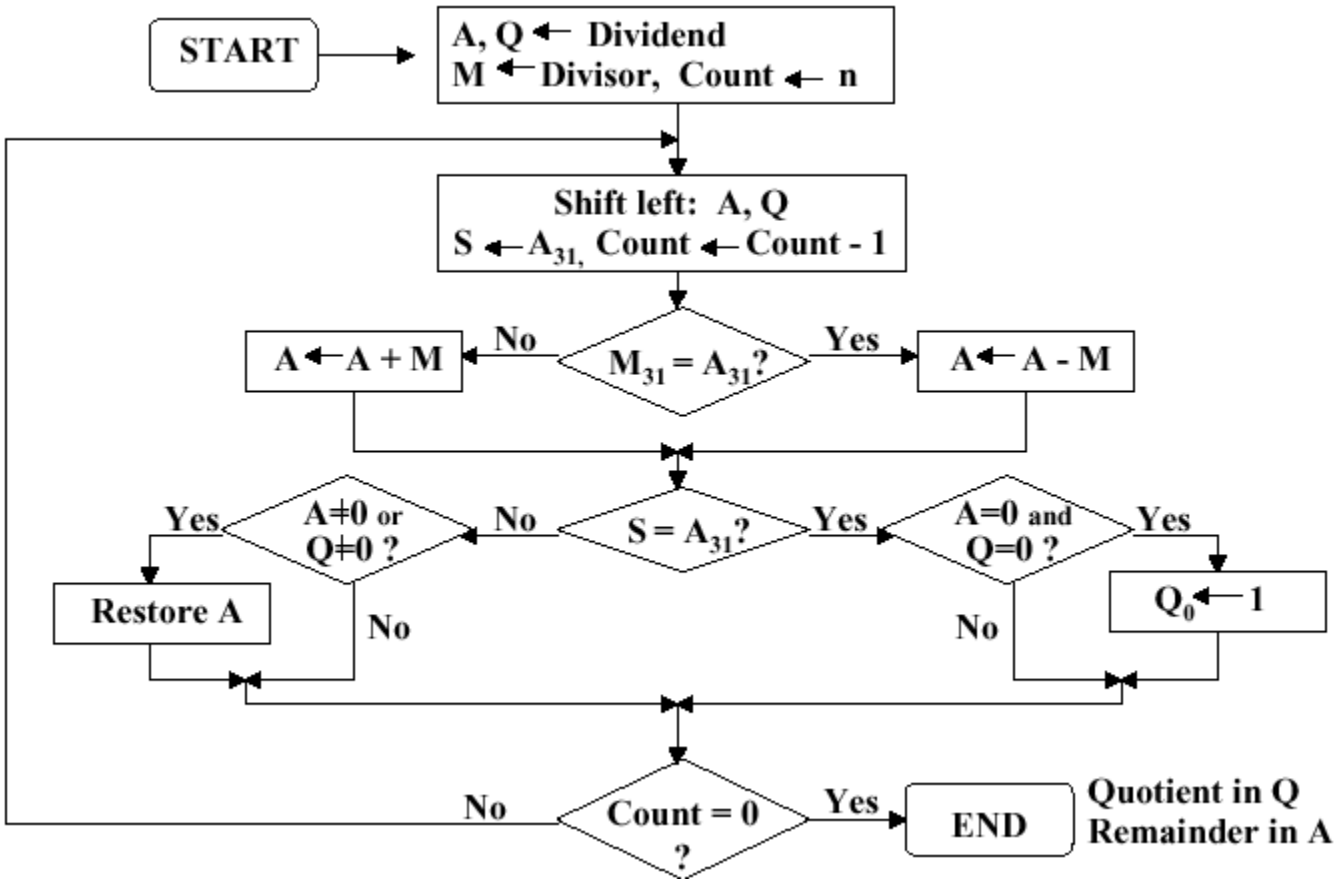
**Signed Division**- With signed division, we negate the quotient if the signs of the divisor and dividend disagree. The remainder and the divident must have the same signs. The governing equation is as follows:

$$\text{Remainder} = \text{Divident} - (\text{Quotient} \cdot \text{Divisor}),$$

and the following four cases apply:

$$\begin{aligned}
(+7) / (+3): &\quad Q = 2; \quad R = 1 \\
(-7) / (+3): &\quad Q = -2; \quad R = -1 \\
(+7) / (-3): &\quad Q = -2; \quad R = 1 \\
(-7) / (-3): &\quad Q = 2; \quad R = -1
\end{aligned}$$

We present the preceding division algorithm, revised for signed numbers, as shown in Figure 3.20a. Four examples, corresponding to each of the four preceding sign permutations, are given in Figure 3.20b and 3.20c.




(a)

| A | Q | M = 0011 | |
|---|---|---|---|
| 0000 | 0111 | Initial values | |
| 0000 | 1110 | Shift | 1 |
| 1101 | | Subtract | |
| 0000 | 1110 | Restore | |
| 0001 | 1100 | Shift | 2 |
| 1110 | | Subtract | |
| 0001 | 1100 | Restore | |
| 0011 | 1000 | Shift | 3 |
| 0000 | | Subtract | |
| 0000 | 1001 | $Q_0 = 1$ | |
| 0001 | 0010 | Shift | 4 |
| 1110 | | Subtract | |
| 0001 | 0010 | Restore | |

(7) / (3)

| A | Q | M = 1101 | |
|---|---|---|---|
| 0000 | 0111 | Initial values | |
| 0000 | 1110 | Shift | 1 |
| 1101 | | Add | |
| 0000 | 1110 | Restore | |
| 0001 | 1100 | Shift | 2 |
| 1110 | | Add | |
| 0001 | 1100 | Restore | |
| 0011 | 1000 | Shift | 3 |
| 0000 | | Add | |
| 0000 | 1001 | $Q_0 = 1$ | |
| 0001 | 0010 | Shift | 4 |
| 1110 | | Add | |
| 0001 | **0010** | Restore | |

(7) / (-3)

(b)

| A | Q | M = 0011 | |
|---|---|---|---|
| 1111 | 1001 | Initial values | |
| 1111 | 0010 | Shift | 1 |
| 0010 | | Add | |
| 1111 | 0010 | Restore | |
| 1110 | 0100 | Shift | 2 |
| 0001 | | Add | |
| 1110 | 0100 | Restore | |
| 1100 | 1000 | Shift | 3 |
| 1111 | | Add | |
| 1111 | 1001 | $Q_0 = 1$ | |
| 1111 | 0010 | Shift | 4 |
| 0010 | | Add | |
| 1111 | **0010** | Restore | |

(-7) / (3)

| A | Q | M = 1101 | |
|---|---|---|---|
| 1111 | 1001 | Initial values | |
| 1111 | 0010 | Shift | 1 |
| 0010 | | Subtract | |
| 1111 | 0010 | Restore | |
| 1110 | 0100 | Shift | 2 |
| 0001 | | Subtract | |
| 1110 | 0100 | Restore | |
| 1100 | 1000 | Shift | 3 |
| 1111 | | Subtract | |
| 1111 | 1001 | $Q_0 = 1$ | |
| 1111 | 0010 | Shift | 4 |
| 0010 | | Subtract | |
| 1111 | 0010 | Restore | |

(-7) / (-3)

(c)

**Figure 3.20.** Division of 32-bit Boolean number representations: (a) algorithm, and (b,c) examples using division of +7 or −7 by the integer +3 or −3; adapted from [Maf01].

## Floating point arithemetic

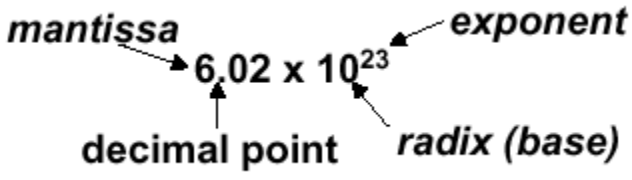
Floating point (FP) representations of decimal numbers are essential to scientific computation using *scientific notation*. The standard for floating point representation is the IEEE 754 Standard. In a computer, there is a tradeoff between range and precision – given a fixed number of binary digits (bits), precision can vary inversely with range. In this section, we overview decimal to FP conversion, MIPS FP instructions, and how registers are used for FP computations.

We have seen that an n-bit register can represent unsigned integers in the range $0$ to $2^n-1$, as well as signed integers in the range $-2^{n-1}$ to $-2^{n-1}-1$. However, there are very large numbers (e.g., $3.15576 \cdot 10^{23}$), very small numbers (e.g., $10^{-25}$), rational numbers with repeated digits (e.g., $2/3 = 0.666666\ldots$), irrationals such as $2^{1/2}$, and transcendental numbers such as e = $2.718\ldots$, all of which need to be represented in computers for scientific computation to be supported.

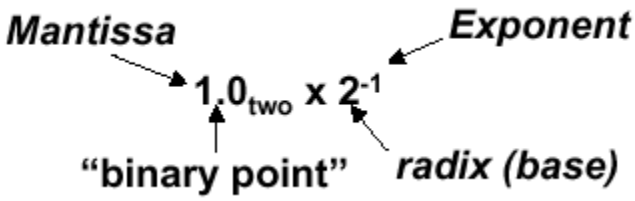
We call the manipulation of these types of numbers *floating point arithmetic* because the decimal point is not fixed (as for integers). In C, such variables are declared as the `float` datatype.

### Scientific Notation and FP Representation

Scientific notation has the following configuration:



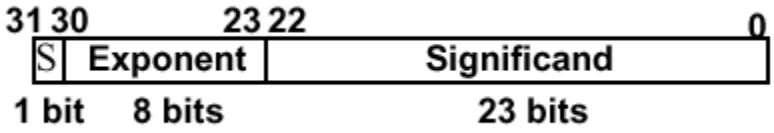
and can be in *normalized form* (mantissa has exactly one digit to the left of the decimal point, e.g., $2.3425 \cdot 10^{-19}$) or *non-normalized form*. Binary scientiic notation has the folowing configuration, which corresponds to the decimal forms:




Assume that we have the following *normal format* for scientific notation in Boolean numbers:

$$+1.xxxxxxx_2 \cdot w^{yyyyy}{}_2 ,$$

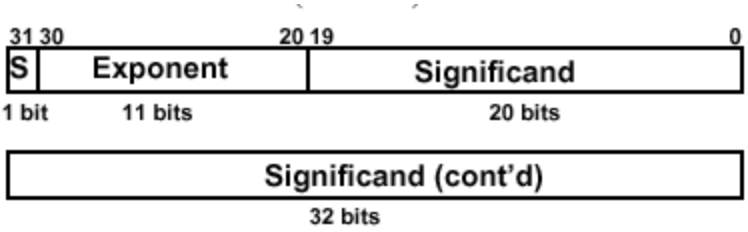
where "xxxxxxx" denotes the *significand* and "yyyyy" denotes the *exponent* and we assume that the number has sign S. This implies the following $32$-bit representation for FP numbers:




which can represent decimal numbers ranging from $-2.0 \cdot 10^{-38}$ to $2.0 \cdot 10^{38}$.

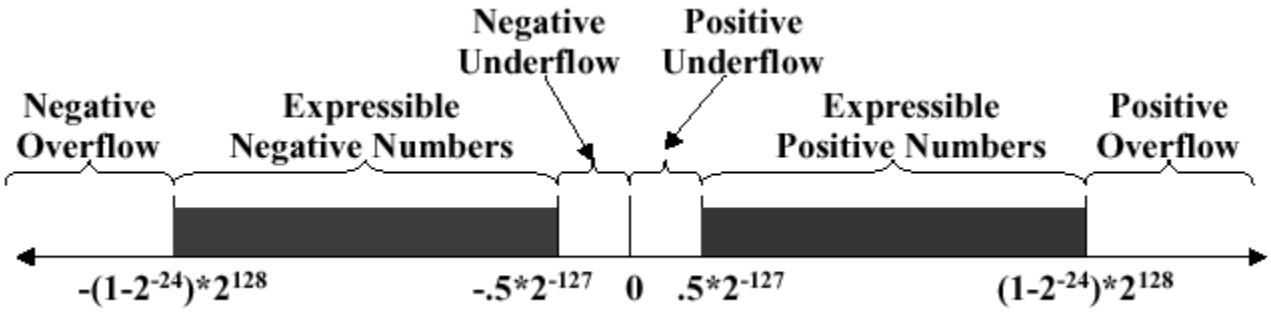
**Overflow and Underflow**

In FP, overflow and underflow are slightly different than in integer numbers. FP overflow (underflow) refers to the positive (negative) exponent being too large for the number of bits alloted to it. This problem can be somewhat ameliorated by the use of *double precision*, whose format is shown as follows:




Here, two $32$-bit words are combined to support an $11$-bit signed exponent and a $52$-bit significand. This representation is declared in C using the `double` datatype, and can support numbers with exponents ranging from $-308_{10}$ to $308_{10}$. The primary advantage is greater precision in the mantissa.

The following chart illustrates specific types of overflow and underflow encountered in standard FP representation:




**IEEE 754 Standard**

Both single- and double-precision FP representations are supported by the IEEE $754$ Standard, which is used in the vast majority of computers since its publication in $1980$. IEEE $754$ facilitates the porting of FP programs, and ensures minimum standards of quality for FP computer arithmetic. The result is a signed representation - the sign bit is $1$ if the FP number represented by IEEE$754$ is negative. Otherwise, the sign is zero. A leading value of $1$ in the significand is implicit for normalized numbers. Thus, the significand, which always has a value between zero and one, occupies $23 + 1$ bits in single-precision FP and $52 + 1$ bits in double precision. Zero

is represented by a zero significand and a zero exponent – there is no leading value of one in the significand. The IEEE 754 representation is thus computed as:

$$\text{FPnumber} = (-1)^{S} \cdot (1 + \text{Significand}) \cdot 2^{\text{Exponent}} .$$

As a parenthetical note, the significand can be translated into decimal values via the following expansion:

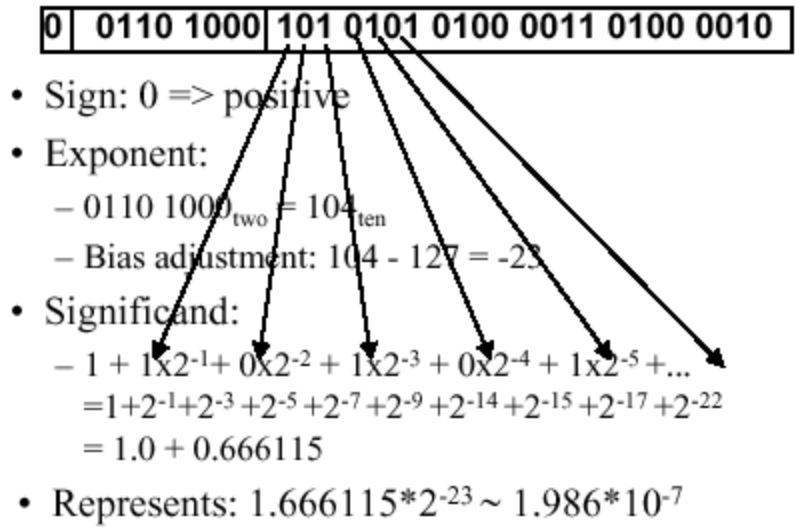
$$1.1001 = (1x2^{0}) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$$

With IEEE 754, it is possible to manipulate FP numbers without having special-purpose FP hardware. For example, consider the sorting of FP numbers. IEEE 754 facilitates breaking FP numbers up into three parts (sign, significant, exponent). The numbers to be sorted are ordered first according to sign (negative < positive), second according to exponent (larger exponent => larger number), and third according to significand (when one has at least two numbers with the same exponents).

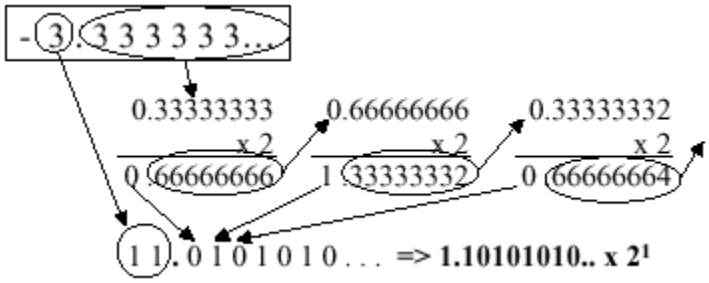
Another issue of interest in IEEE 754 is *biased notation* for exponents. Observe that twos complement notation does not work for exponents: the largest negative (positive) exponent is $00000001_2$ ($11111111_2$). Thus, we must add a *bias term* to the exponent to center the range of exponents on the bias number, which is then equated to zero. The bias term is 127 (1023) for the IEEE 754 single-precision (double-precision) representation. This implies that

$$\text{FPnumber} = (-1)^{S} \cdot (1 + \text{Significand}) \cdot 2^{(\text{Exponent} - \text{Bias})} .$$

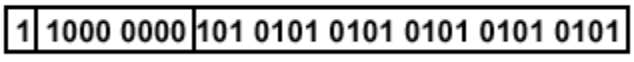
As a result, we have the following example of binary to decimal floating point conversion:




Decimal-to-binary FP conversion is somewhat more difficult. Three cases pertain: (1) the decimal number can be expressed as a fraction n/d where d is a power of two; (2) the decimal number has repeated digits (e.g., 0.33333); or (3) the decimal number does not fit either Case 1 or Case 2. In Case 1, one selects the exponent as $-\log_2(d)$, and converts n to binary notation. Case 3 is more difficult, and will not be discussed here. Case 2 is exemplified in the following diagram:




Here, the significand is 101 0101 0101 0101 0101 0101, the sign is negative (representation = 1), and the exponent is computed as $1 + 127 = 128_{10} = 1000\ 0000_2$. This yields the following representation in IEEE 754 standard notation:



The following table summarizes special values that can be represented using the IEEE 754 standard.

**Table 3.1**. Special values in the IEEE 754 standard.

| Special Value | Exponent | Significand |
|---|---|---|
| +/- 0 | 0000 0000 | 0 |
| Denormalized number | 0000 0000 | Nonzero |
| NaN | 1111 1111 | Nonzero |
| +/- infinity | 1111 1111 | 0 |

Of particular interest in the preceding table is the *NaN* (not a number) representation. For example, when taking the square root of a negative number, or when dividing by zero, we encounter operations that are undefined in the arithmetic operations over real numbers. These resuls are called NaNs and are represented with an exponent of 255 and a zero significand. NaNs can help with debugging, but they contaminate calculations (e.g., NaN + $x$ = NaN). The recommended approach to NaNs, especially for software designers or engineers early in their respective careers, is not to use NaNs.

# Computer Arithmetic

## Negative Number Representation

- **Sign Magnitude**

Sign magnitude is a very simple representation of negative numbers. In sign magnitude the first bit is dedicated to represent the sign and hence it is called sign bit.

Sign bit '1' represents negative sign.
Sign bit '0' represents positive sign.

In sign magnitude representation of a n – bit number, the first bit will represent sign and rest n-1 bits represent magnitude of number.

For example,

- +25 = 011001

Where 11001 = 25
And 0 for '+'

- -25 = 111001

Where 11001 = 25
And 1 for '-'.
**Range of number represented by sign magnitude method** = $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$ (for n bit number)

But there is one problem in sign magnitude and that is we have two representations of 0

+0 = 000000
– 0 = 100000

- **2's complement method**

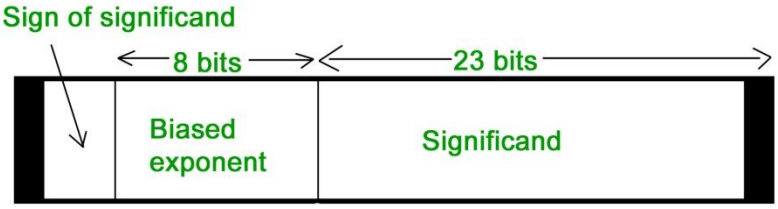
To represent a negative number in this form, first we need to take the 1's complement of the number represented in simple positive binary form and then add 1 to it.
For example:
(-8)10 = (1000)2
1's complement of 1000 = 0111
Adding 1 to it, 0111 + 1 = 1000

So, (-8)10 = (1000)2
Please don't get confused with (8)10 =1000 and (-8)10=1000 as with 4 bits, we can't represent a positive number more than 7. So, 1000 is representing -8 only.

**Range of number represented by 2's complement =** $(-2^{n-1}$ to $2^{n-1} – 1)$

## Floating point representation of numbers

- **32-bit representation floating point numbers IEEE standard**




**Normalization**

- Floating point numbers are usually normalized
- Exponent is adjusted so that leading bit (MSB) of mantissa is 1
- Since it is always 1 there is no need to store it
- Scientific notation where numbers are normalized to give a single digit before the decimal point like in decimal system e.g. 3.123 x 103

For example, we represent 3.625 in 32 bit format.

Changing 3 in binary=11
Changing .625 in binary

| | |
|---|---|
| .625 X 2 | 1 |
| .25 X 2 | 0 |
| .5 X 2 | 1 |

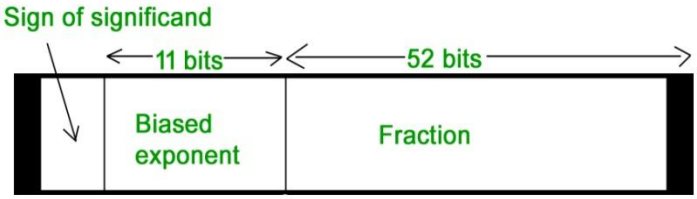
Writing in binary exponent form
3.625=11.101 X 20
On normalizing
11.101 X 20=1.1101 X 21
On biasing exponent = 127 + 1 = 128
(128)10=(10000000) 2
**For getting significand**

Digits after decimal = 1101
Expanding to 23 bit = 11010000000000000000000
**Setting sign bit**
As it is a positive number, sign bit = 0

Finally we arrange according to representation

```
Sign bit       exponent       significand


0              10000000    11010000000000000000000
```

- **64-bit representation floating point numbers IEEE standard**




Again we follow the same procedure upto normalization. After that, we add 1023 to bias the exponent.

For example, we represent -3.625 in 64 bit format.

Changing 3 in binary = 11

Changing .625 in binary

```
.625 X 2    1

.25 X 2     0

.5 X 2      1
```

Writing in binary exponent form

3.625 = 11.101 X 20

On normalizing

11.101 X 20 = 1.1101 X 21

On biasing exponent 1023 + 1 = 1024

(1024)10 = (10000000000)2

So 11 bit exponent = 10000000000

52 bit significand = 1101000000000 ……. making total 52 bits

Setting sign bit = 1 (number is negative)

So, final representation
1 10000000000 110100000000 ……. making total 52 bits by adding further 0's

**Converting floating point into decimal**

Let's convert a FP number into decimal
1 01111100 11000000000000000000000
The decimal value of an IEEE number is given by the formula:
$\mathbf{(1 -2s) * (1 + f) * 2^{(e - bias)}}$
where

- s, f and e fields are taken as decimal here.
- (1 -2s) is 1 or -1, depending upon sign bit 0 and 1
- add an implicit 1 to the significand (fraction field f), as in formula

Again, the bias is either 127 or 1023, for single or double precision respectively.

First convert each individual field to decimal.

- The sign bit s is 1
- The e field contains 01111100 = (124)10
- The mantissa is 0.11000 … = (0.75)10

Putting these values in formula
$(1 - 2) * (1 + 0.75) * 2124 - 127 = ( - 1.75 * 2^{-3} ) = - 0.21875$

## FLOATING POINT ADDITION AND SUBTRACTION

- **FLOATING POINT ADDITION**

To understand floating point addition, first we see addition of real numbers in decimal as same logic is applied in both cases.

**For example,** we have to add **1.1 * 103** and **50.**
We cannot add these numbers directly. First, we need to align the exponent and then, we can add significand.

After aligning exponent, we get **50 = 0.05 * 103**
Now adding significand, **0.05 + 1.1 = 1.15**
So, finally we get **(1.1 * 103 + 50) = 1.15 * 103**
Here, notice that we shifted **50** and made it **0.05** to add these numbers.

**Now let us take example of floating point number addition**
We follow these steps to add two numbers:

1. Align the significand
2. Add the significands
3. Normalize the result

**Let the two numbers be**
x = 9.75
y = 0.5625

Converting them into 32-bit floating point representation,
**9.75**'s representation in 32-bit format = **0 10000010 00111000000000000000000**
**0.5625**'s representation in 32-bit format = **0 01111110 00100000000000000000000**

Now we get the difference of exponents to know how much shifting is required.
$(\mathbf{10000010 - 01111110})_2 = (\mathbf{4})_{10}$

Now, we shift the mantissa of lesser number right side by 4 units.
Mantissa of **0.5625 = 1.00100000000000000000000**
(note that 1 before decimal point is understood in 32-bit representation)
Shifting right by **4** units, we get **0.00010010000000000000000**
Mantissa of **9.75** = **1. 00111000000000000000000**

Adding mantissa of both
**0. 00010010000000000000000**
**+ 1. 00111000000000000000000**

———————————————————-

**1. 01001010000000000000000**

In final answer, we take exponent of bigger number
So, final answer consist of :
Sign bit = **0**
Exponent of bigger number = **10000010**
Mantissa = **01001010000000000000000**

32 bit representation of answer = **x + y** = **0 10000010 01001010000000000000000**

- **FLOATING POINT SUBTRACTION**

Subtraction is similar to addition with some differences like we subtract mantissa unlike addition and in sign bit we put the sign of greater number.

**Let the two numbers be**
x = 9.75
y = – 0.5625

Converting them into 32-bit floating point representation
**9.75**'s representation in 32-bit format = **0 10000010 00111000000000000000000**
**– 0.5625**'s representation in 32-bit format = **1 01111110 00100000000000000000000**

Now, we find the difference of exponents to know how much shifting is required.

(**10000010 – 01111110**)$_2$ = (**4**)$_{10}$
Now, we shift the mantissa of lesser number right side by 4 units.
Mantissa of **– 0.5625 = 1.00100000000000000000000**
(note that 1 before decimal point is understood in 32-bit representation)
Shifting right by **4** units, **0.00010010000000000000000**
Mantissa of **9.75**= **1. 00111000000000000000000**

Subtracting mantissa of both
**0. 00010010000000000000000**

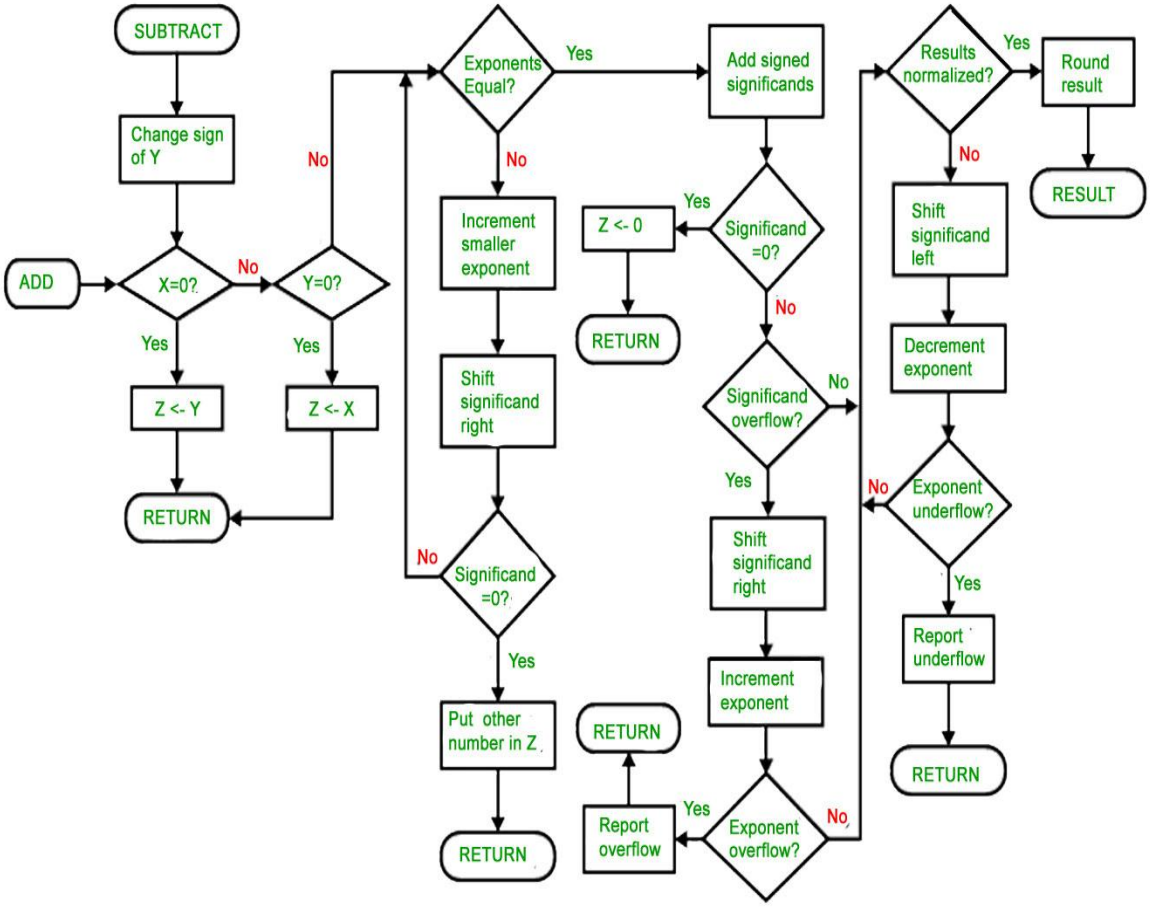
**– 1. 00111000000000000000000**

————————————————————

**1. 00100110000000000000000**

Sign bit of bigger number = **0**

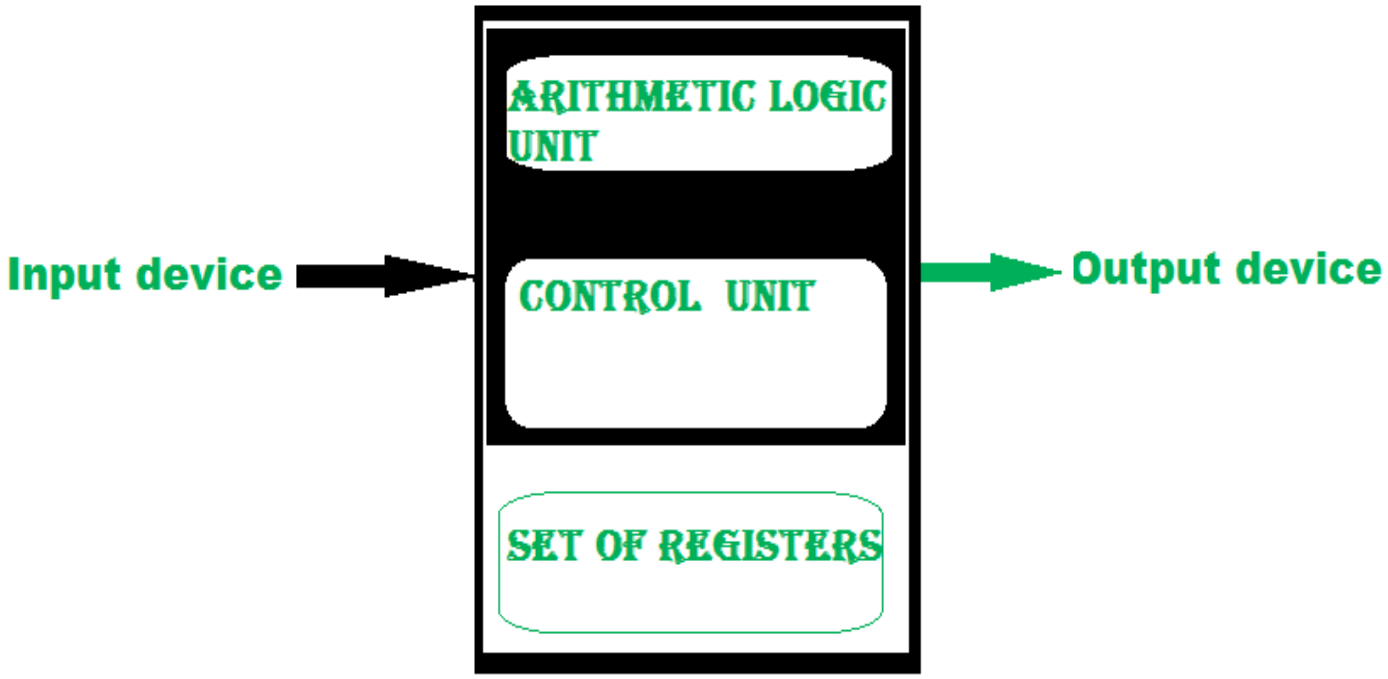
So, finally the answer = **x – y = 0 10000010 00100110000000000000000**




Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

# Introduction of ALU and Data Path

Representing and storing numbers were the basic of operation of the computers of earlier times. The real go came when computation, manipulating numbers like adding, multiplying came into picture. These operations are handled by computer's **arithmetic logic unit (ALU)**. The ALU is the mathematical brain of a computer. The first ALU was INTEL 74181 implemented as a 7400 series is a TTL integrated circuit which was released in 1970.
The **ALU** is a digital circuit that provides arithmetic and logic operation. It is the fundamental building block of central processing unit of a computer. A modern CPU has very powerful ALU and it is complex in design. In addition to ALU modern CPU contains control unit and set of registers. Most of the operations are performed by one or more ALU's, which load data from input register. Registers are a small amount of storage available to CPU. These registers can be accessed very fast. The control unit tells ALU what operation to perform on the available data. After calculation/manipulation the ALU stores the output in an output register.




Central Processing Unit

The CPU can be divided into two section: data section and control section. The DATA section is also known as data path.

**BUS:**
In early computers “BUS” were parallel electrical wires with multiple hardware connections. Therefore a bus is communication system that transfers data between component inside a computer, or between computers. It includes hardware components like wires, optical fibers, etc and software, including communication protocols. The Registers, ALU and the interconnecting BUS are collectively referred as data path.
Types of bus are:

1. **Address bus:** The buses which are used to carry address.
2. **Data bus:** The buses which are used to carry data.
3. **Control bus:** If the bus is carrying control signals .
4. **Power bus:** If it is carrying clock pulse, power signals it is known as power bus, and so on.

The bus can be dedicated, i.e., it can be used for a single purpose or it can be multiplexed, i.e., it can be used for multiple purpose. When we would have different kinds of buses, different types of bus organisation will take place.

- **Program Counter –**
  A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory. As each instruction gets fetched, the program counter increases its stored value by 1. It is a digital counter needed for faster execution of tasks as well as for tracking the current execution point.
- **Instruction Register –**
  In computing, an instruction register (IR) is the part of a CPU’s control unit that holds the instruction currently being executed or decoded. An instruction register is the part of a CPU’s control unit that holds the instruction currently being executed or decoded. Instruction register specifically holds the instruction and provides it to instruction decoder circuit.
- **Memory Address Register –**
  The Memory Address Register (MAR) is the CPU register that either stores the memory address from which data will be fetched from the CPU, or the address to which data will be sent and stored. It is a temporary storage component in the CPU(central processing unit) which temporarily stores the address (location) of the data sent by the memory unit until the instruction for the particular data is executed.
- **Memory Data Register –**
  The memory data register (MDR) is the register in a computer’s processor, or central processing unit, CPU, that stores the data being transferred to and from the immediate access storage. Mmemory data register (MDR) is also known as memory buffer register (MBR).
- **General Purpose Register –**
  General purpose registers are used to store temporary data within the

microprocessor. It is a multipurpose register. They can be used either by programmer or by a user.

**One Bus organization –**

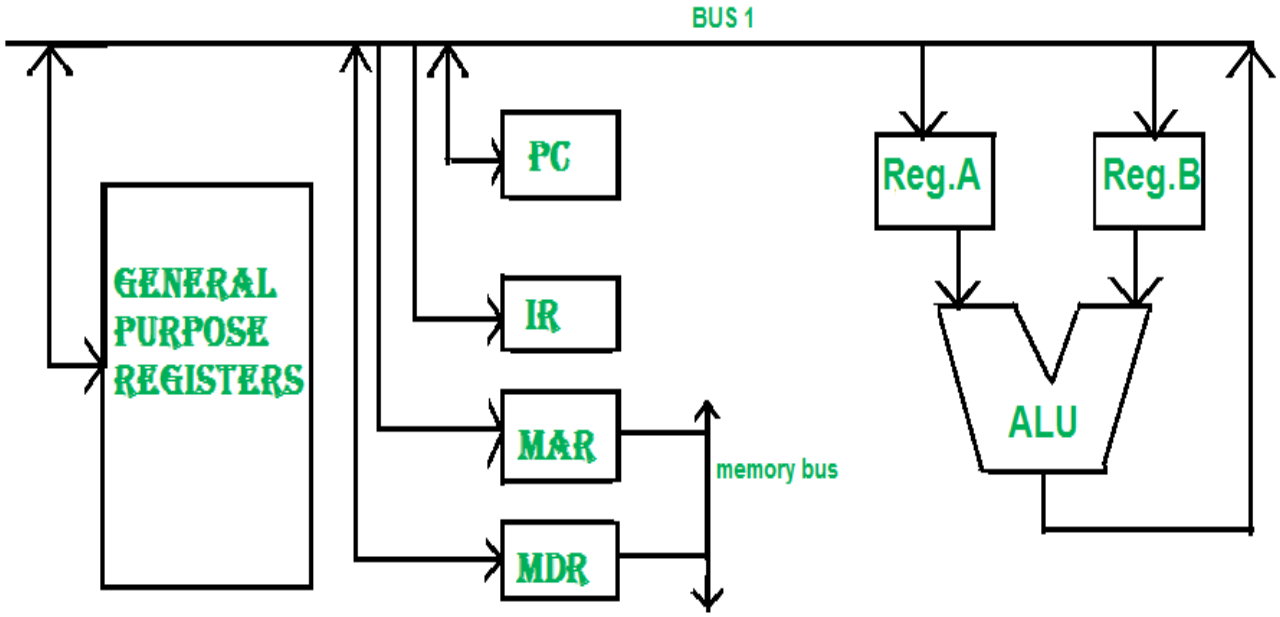



fig. One Bus Organisation

In one bus organisation, a single bus is used for multiple purpose. A set of general purpose register, program counter, instruction register, memory address register(MAR), memory data register(MDR) are connected with the single bus. Memory read/write can be done with MAR and MDR. The program counter points to the memory location from where the next instruction is to be fetched. Instruction register is that very register will hold the copy of the current instruction. In case of one bus organisation, at a time only one operand can be read from the bus.

As a result of that, if the requirement is to read two operand for the operation then read operation need to be carried twice. So that's why it is making the process little longer. One of the advantage of one bus organisation is that, it is one of the simplest and also this is very cheap to implement. At the same time a disadvantage lies that it has only one bus and this "one bus" is accessed by all general purpose registers, program counter, instruction register, MAR, MDR making each and every operation sequential. No one recommend this architecture now-a-days.

**Two Bus organizatrion –**

Two overcome the disadvantage of one bus organisation an another architecture was developed known as two bus organisation. In two bus organisation there are two buses. The general purpose register can read/write from both the buses. In this case, two

operands can be fetched at the same time because of the two buses. One of bus fetch operand for ALU and another bus fetch for register. The situation arrises when both buses are busy fetching operands, output can be stored in temporary register and when the buses are free, particular output can be dumped on the buses.
There are two versions of two bus organisation, i.e., in-bus and out-bus.From in-bus the general purpose register can read data and to the out bus the general purpose registers can write data.Here buses gets dedicated.

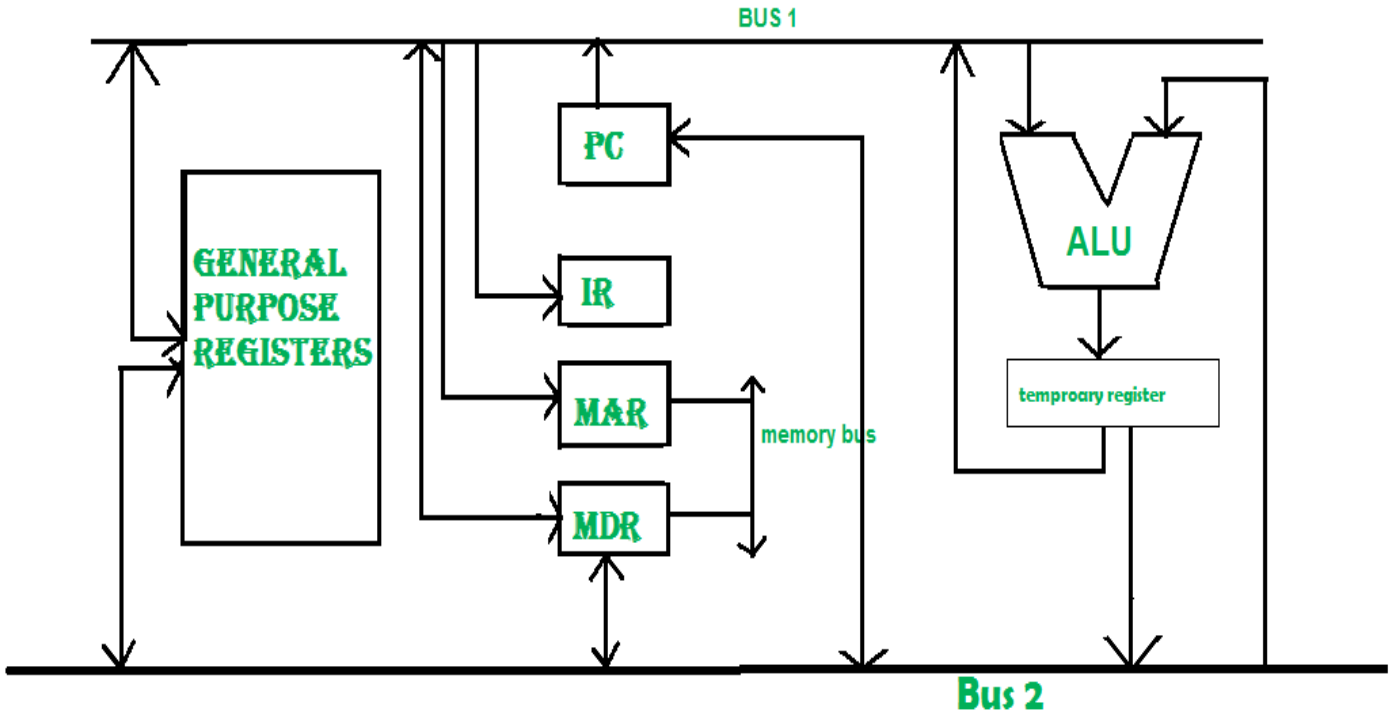



fig.TWO BUS ORGANISATION

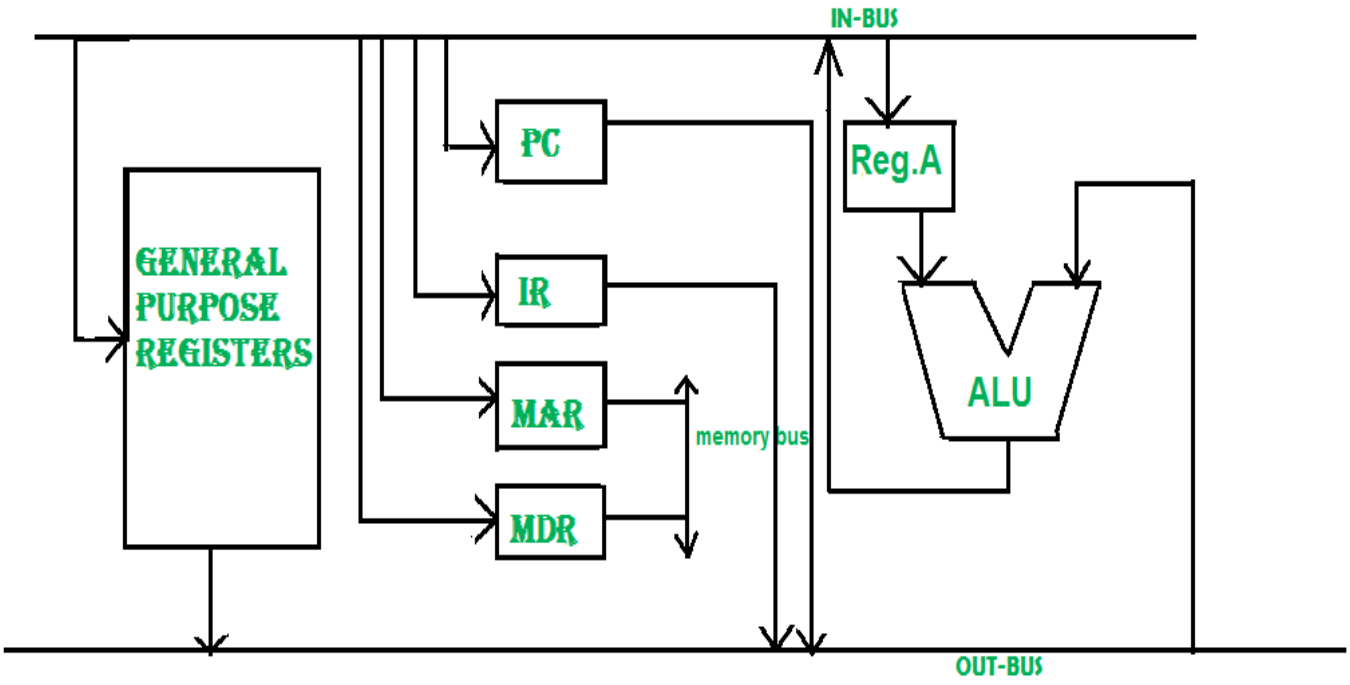



fig.TWO BUS ORGANISATION

**Three Bus organizatrion –**

In three bus organisation we have three bus, OUT bus1, OUT bus2 and a IN bus. From the out buses we can get the operand which can come from general purpose register and evaluated in ALU and the output is dropped on In Bus so it can be sent to respective registers.This implementation is a bit complex but faster in nature because in parallel two operands can flow into ALU and out of ALU. It was developed to overcome the “busy waiting” problem of two bus organisation. In this structure after execution, the output can be dropped on the bus without waiting because of presence of an extra bus. The structure is given below in the figure.

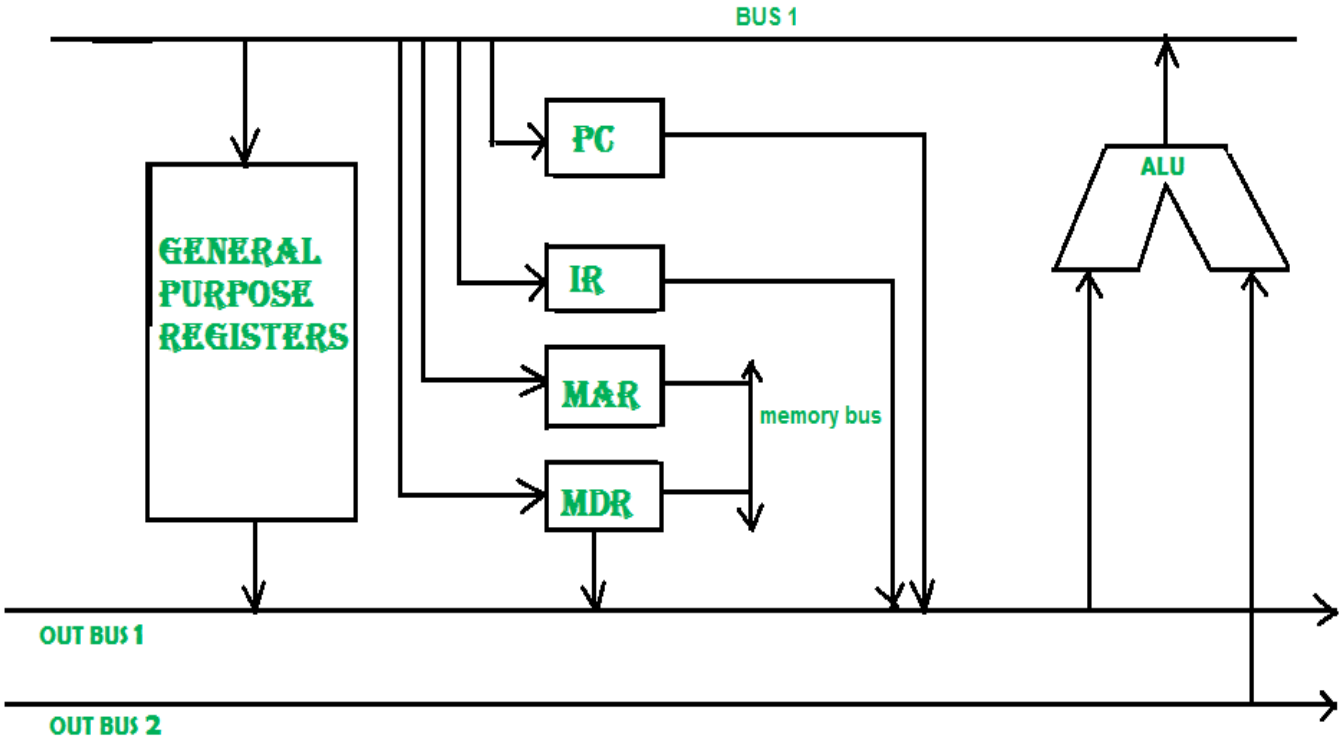



fig.THREE BUS ORGANISATION

The main **advantages** of multiple bus organisations over single bus are as given below.

1. Increase in size of the registers.
2. Reduction in the number of cycles for execution.
3. Increases the speed of execution or we can say faster execution.