Basic Structure of Computers
&
Machine Instructions and Programs

TOPIC: ***Basic Structure of Computers:*** Basic Operational Concepts, Bus Structures, Performance –Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement. ***Machine Instructions and Program:*** Memory Location and Addresses Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and Queues, Subroutines, Additional Instructions, Encoding of Machine Instructions

1. BASIC OPERATIONAL CONCEPT:

The program to be executed is stored in memory. Instructions are accessed from memory to the processor one by one and executed.

STEPS FOR INSTRUCTION EXECUTION

Consider the following instruction

Ex: 1 Add LOCA, R_0

This instruction is in the form of the following instruction format

Opcode Source, Source/ Destination

Where Add is the *operation code,* LOCA is the Memory operand and R_0 is Register operand
This instruction adds the contents of memory location LOCA with the contents of Register R_0 and the result is stored in R_0 Register.
The symbolic representation of this instruction is

$R_0 \leftarrow [LOCA] + [R_0]$

The contents of memory location LOCA and Register R_0 before and after the execution of this instruction is as follows

Before instruction execution	After instruction execution
LOCA = 23H	**LOCA = 23H**
R_0 = 22H	**R_0 = 45H**

The steps for instruction execution are as follows

1. Fetch the instruction from memory into the IR (instruction register in CPU).
2. Decode the instruction 1111000000 10011010
3. Access the first Operand
4. Access the second Operand
5. Perform the operation according to the Opcode (operation code).
6. Store the result into the Destination Memory location or Destination Register.

Ex:2 **Add R_1, R_2, R_3** (3 address instruction format)

This instruction is in the form of the following instruction format

Opcode, Source-1, Source-2, Destination

Where R1 is Source Operand-1, R2 is the Source Operand-2 and R3 is the Destination. This instruction adds the contents of Register R1 with the contents of R2 and the result is placed in R3 Register.

The symbolic representation of this instruction is

R3 ← [R1] + [R2]

The contents of Registers R1,R2,R3 before and after the execution of this instruction is as follows.

Before instruction execution	After instruction execution
R1 = 24H	**R1 = 24H**
R2 = 34H	**R2 = 34H**
R3 = 38H	**R3 = 58H**

The steps for instruction execution is as follows

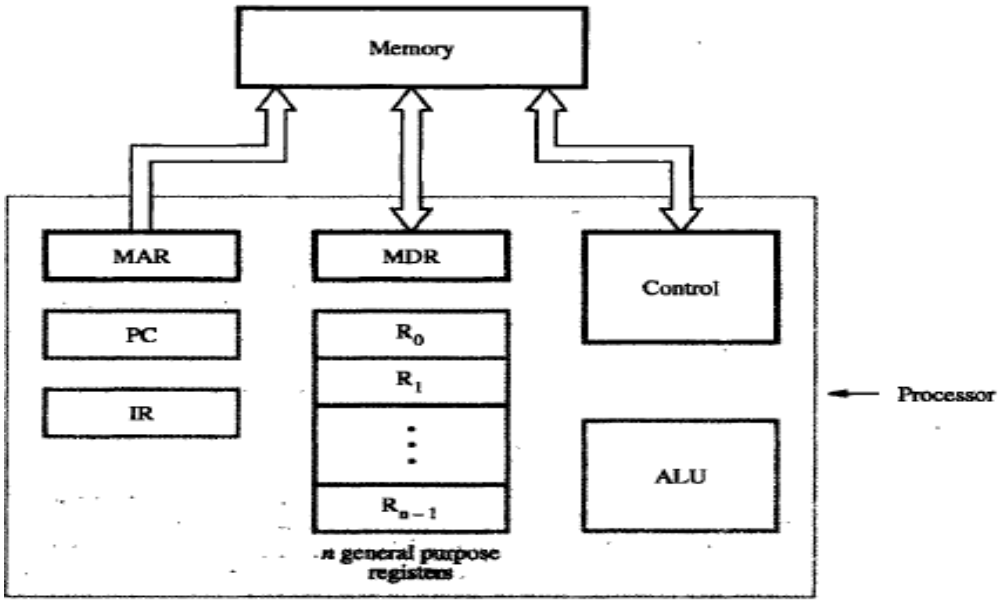
1. Fetch the instruction from memory into the IR.
2. Decode the instruction
3. Access the First Operand R1
4. Access the Second Operand R2
5. Perform the operation according to the Operation Code.
6. Store the result into the Destination Register R3.

CONNECTION BETWEEN MEMORY AND PROCESSOR

The connection between Memory and Processor is as shown in the figure.

The Processor consists of different types of registers.

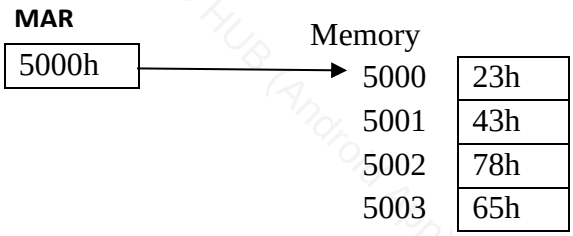
1. MAR (Memory Address Register)
2. MDR (Memory Data Register)
3. Control Unit
4. PC (Program Counter)
5. General Purpose Registers
6. IR (Instruction Register)
7. ALU (Arithmetic and Logic Unit)



The functions of these registers are as follows

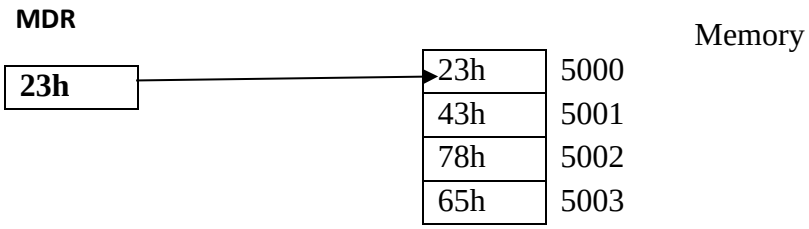
1. MAR

- It establishes communication between Memory and Processor
- It stores the address of the Memory Location as shown in the figure.



2. MDR

- It also establishes communication between Memory and the Processor.
- It stores the **contents** of the memory location (data or operand), written into or read from memory as shown in the figure.



3. CONTROL UNIT

- It controls the data transfer operations between memory and the processor.
- It controls the data transfer operations between I/O and processor.
- It generates control signals for Memory and I/O devices.

4. PC (PROGRAM COUNTER)

- It is a special purpose register used to hold the address of the next instruction to be executed.
- The contents of PC are incremented by 1 or 2 or 4, during the execution of current instruction.
- The contents of PC are incremented by 1 for 8 bit CPU, 2 for 16 bit CPU and for 4 for 32 bit CPU.

4. GENERAL PURPOSE REGISTER / REGISTER ARRAY

The structure of register file is as shown in the figure

R_0
R_1
R_2
.
R_{n-1}

- It consists of set of registers.
- A register is defined as group of flip flops. Each flip flop is designed to store 1 bit of data.
- It is a storage element.
- It is used to store the data temporarily during the execution of the program(eg: result).
- It can be used as a pointer to Memory.
- The Register size depends on the processing speed of the CPU
- EX: Register size = 8 bits for 8 bit CPU

5. IR (INSTRUCTION REGISTER

It holds the instruction to be executed. It notifies the control unit, which generates timing signals that controls various operations in the execution of that instruction.

6. ALU (ARITHMETIC and LOGIC UNIT)

- It performs arithmetic and logical operations on given data.

Steps for fetch the instruction

PC contents are transferred to MAR
Read signal is sent to memory by control unit.
The instruction from memory location is sent to MDR.
The content of MDR is moved to IR.

[PC] → MAR ⟶ Memory → MDR → IR
CU (read signal)

2. BUS STRUCTURE

Bus is defined as set of parallel wires used for data communication between different parts of computer. Each wire carries 1 bit of data. There are 3 types of buses, namely

1. Address bus
2. Data bus and
3. Control bus1.

1. ***Address bus :***

- It is unidirectional.
- The processor (CPU) sends the address of an I/O device or Memory device by means of this bus.

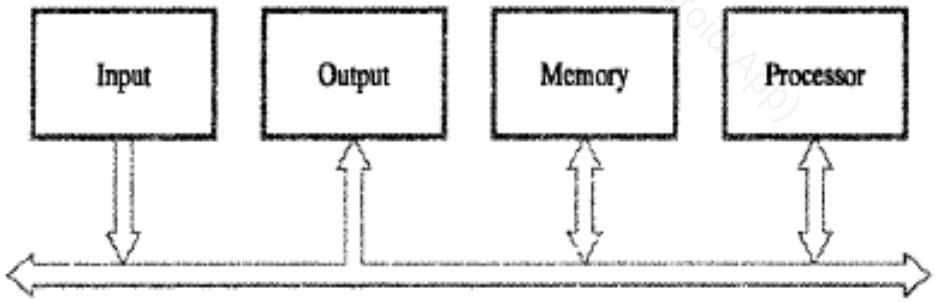
2. ***Data bus***

- It is a bidirectional bus.
- The CPU sends data from Memory to CPU and vice versa as well as from I/O to CPU and vice versa by means of this bus.

3. ***Control bus:***

- This bus carries control signals for Memory and I/O devices. It generates control signals for Memory namely MEMRD and MEMWR and control signals for I/O devices namely IORD and IOWR.

The structure of single bus organization is as shown in the figure.



- The I/O devices, Memory and CPU are connected to this bus is as shown in the figure.
- It establishes communication between two devices, at a time.

Features of Single bus organization are

- Less Expensive
- Flexible to connect I/O devices.
- Poor performance due to single bus.

There is a variation in the devices connected to this bus in terms of speed of operation. Few devices like keyboard, are very slow. Devices like optical disk are faster. Memory and processor are faster, but all these devices uses the same bus. Hence to provide the synchronization

between two devices, a buffer register is attached to each device. It holds the data temporarily during the data transfer between two devices.

3. *PERFORMANCE*

Basic performance Equation

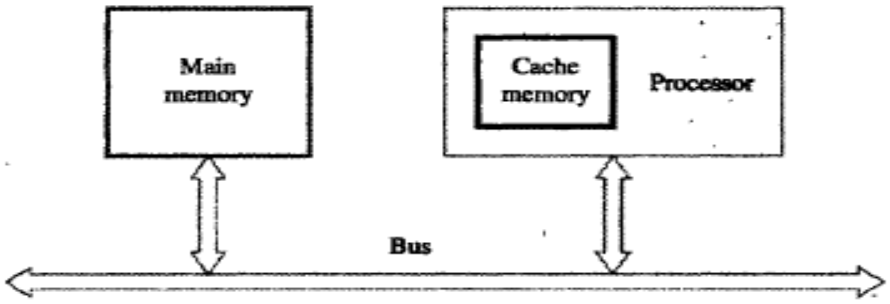
- The performance of a Computer System is based on hardware design of the processor and the instruction set of the processors.
- To obtain high performance of computer system it is necessary to reduce the execution time of the processor.
- Execution time: It is defined as total time required executing one complete program.
- The processing time of a program includes time taken to read inputs, display outputs, system services, execution time etc.
- The performance of the processor is inversely proportional to execution time of the processor.

More performance = Less Execution time.
Less Performance = More Execution time.

The Performance of the Computer System is based on the following factors

1. *Cache Memory*
2. *Processor clock*
3. *Basic Performance Equation*
4. *Instructions*
5. *Compiler*

***CACHE MEMORY*:** It is defined as a *fast access memory* located in between CPU and Memory. It is part of the processor as shown in the fig



The processor needs more time to read the data and instructions from main memory because main memory is away from the processor as shown in the figure. Hence it slowdown the performance of the system.

The processor needs less time to read the data and instructions from Cache Memory because it is part of the processor. Hence it improves the performance of the system.

***PROCESSOR CLOCK*:** The processor circuits are controlled by timing signals called as Clock. It defines constant time intervals and are called as Clock Cycles. To execute one instruction there are 3 basic steps namely

1. Fetch
2. Decode
3. Execute.

The processor uses one clock cycle to perform one operation as shown in the figure

Clock Cycle → T1 T2 T3
Instruction → Fetch Decode Execute

The performance of the processor depends on the length of the clock cycle. To obtain high performance reduce the length of the clock cycle. Let ' P ' be the number of clock cycles generated by the Processor and ' R ' be the Clock rate .

The Clock rate is inversely proportional to the number of clock cycles.

i.e R = 1/P.

Cycles/second is measured in Hertz (Hz). Eg: 500MHz, 1.25GHz.

Two ways to increase the clock rate –

- Improve the IC technology by making the logical circuit work faster, so that the time taken for the basic steps reduces.
- Reduce the clock period, P.

BASIC PERFORMANCE EQUATION

Let ' T ' be *total time* required to execute the program.

Let 'N ' be the *number of instructions* contained in the program.

Let ' S ' be the *average number of steps* required to execute one instruction.

Let ' R' be number of clock cycles per second generated by the processor to execute one program.

Processor Execution Time is given by

T = N * S / R

This equation is called as Basic Performance Equation.

For the programmer the value of T is important. To obtain high performance it is necessary to reduce the values of N & S and increase the value of R

Performance of a computer can also be measured by using **benchmark** programs.

SPEC (System Performance Evaluation Corporation) is an non-profitable organization, that measures performance of computer using SPEC rating. The organization publishes the application programs and also time taken to execute these programs in standard systems.

$$SPEC = \frac{Running\ time\ of\ reference\ Computer}{Running\ time\ of\ computer\ under\ test}$$

DIFFERENCES MULTIPROCESSOR AND MULTICOMPUTER

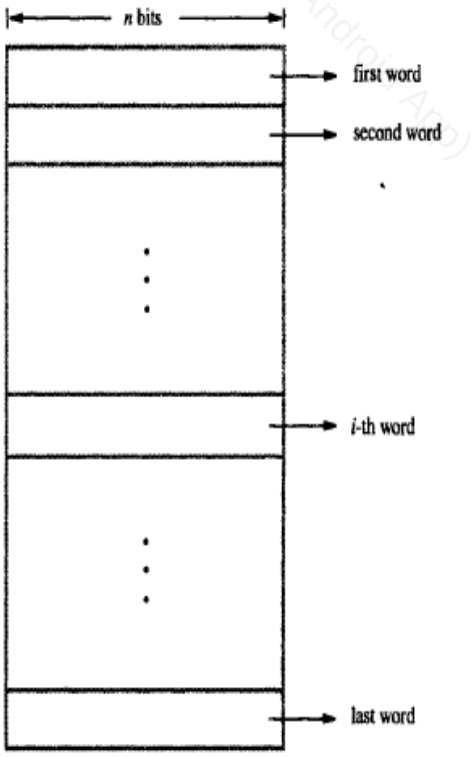
MULTIPROCESSOR	MULTICOMPUTER
1. Interconnection of two or more processors by means of system bus.	Interconnection of two or more computers by means of cables.
2. It uses common memory to hold the data and instructions.	It has its own memory to store data and instructions.
3. Complexity in hardware design.	Not much complexity in hardware design.
4. Difficult to program for multiprocessor system.	Easy to program for multiprocessor system

4. MEMORY LOCATIONS AND ADDRESSES

1. Memory is a storage device. It is used to store character operands, data operands and instructions.
2. It consists of number of semiconductor cells and each cell holds 1 bit of information. A group of 8 bits is called as byte and a group of 16 or 32 or 64 bits is called as word.

World length = 16 for 16 bit CPU and World length = 32 for 32 bit CPU. Word length is defined as number of bits in a word.

- Memory is organized in terms of bytes or words.
- The organization of memory for 32 bit processor is as shown in the fig.



The contents of memory location can be accessed for read and write operation. The memory is accessed either by specifying address of the memory location or by name of the memory location.

- **Address space :** It is defined as number of bytes accessible to CPU and it depends on the number of address lines.

5. BYTE ADDRESSABILITY

Each byte of the memory are addressed, this addressing used in most computers are called byte addressability. Hence Byte Addressability is the process of assignment of address to successive bytes of the memory. The successive bytes have the addresses 1, 2, 3, 4…….2^n-1. The memory is accessed in words.

In a 32 bit machine, each word is 32 bit and the successive addresses are 0,4,8,12,… and so on.

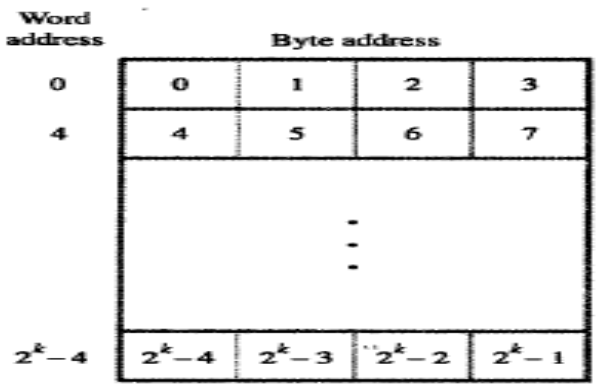
Address	32 – bit word			
0000	0^{th} byte	1^{st} byte	2^{nd} byte	3^{rd} byte
0004	4^{th} byte	5^{th} byte	6^{th} byte	7^{th} byte
0008	8^{th} byte	9^{th} byte	10^{th} byte	11^{th} byte
0012	12^{th} byte	13^{th} byte	14^{th} byte	15^{th} byte
…..	…..	…..	…..	…..
n-3	n-3^{th} byte	n-2^{th} byte	n-1^{th} byte	n^{th} byte

BIG ENDIAN and LITTLE ENDIAN ASSIGNMENT

Two ways in which byte addresses can be assigned in a word.
Or
Two ways in which a word is stored in memory.

1. Big endian
2. Little endian

BIG ENDIAN ASSIGNMENT



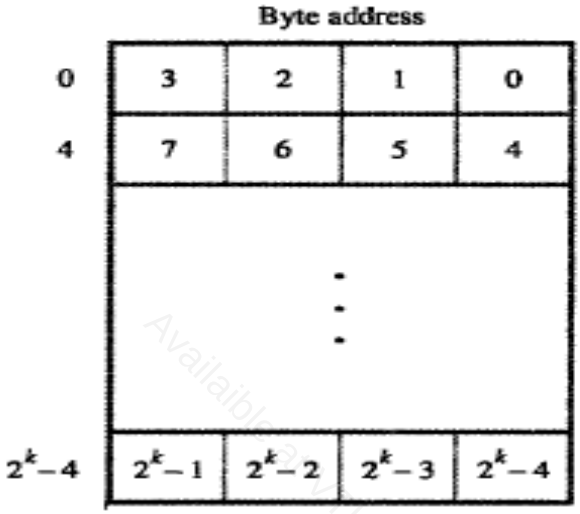
In this technique lower byte of data is assigned to higher address of the memory and higher byte of data is assigned to lower address of the memory.

The structure of memory to represent 32 bit number for big endian assignment is as shown in the above figure.

LITTLE ENDIAN ASSIGNMENT

In this technique lower byte of data is assigned to lower address of the memory and higher byte of data is assigned to higher address of the memory.

The structure of memory to represent 32 bit number for little endian assignment is as shown in the fig.



Eg – store a word "JOHNSENA" in memory starting from word 1000, using Big Endian and Little endian.

Bigendian -

1000	J **1000**	O **1001**	H **1002**	N **1003**
1004	S **1004**	E **1005**	N **1006**	A **1007**

Little endian -

1000	N **1000**	H **1001**	O **1002**	J **1003**
1004	A **1004**	N **1005**	E **1006**	S **1007**

WORD ALLIGNMENT

Word is the group of bytes in memory. Number of bits in a word is the word length.

Eg – 32-bit word length, 64-bit word length etc.

The word locations of memory are aligned, if they begin with the address, which is multiple of number of bytes in a word.

The structure of memory for 16 bit CPU, 32 bit CPU and 64 bit CPU are as shown in the figures 1,2 and 3 respectively

For 16 bit CPU

4000	**34H**
4002	**65H**
4004	**86H**
4006	**93H**
4008	**45H**

(Here, no. of bytes of a word is 2, and the address of word is in multiples of 2)

For 32 bit CPU

4000	**34H**
4004	**65H**
4008	**86H**
4012	**93H**
4016	**45H**

(Here, no. of bytes of a word is 4, and the address of word is in multiples of 4)

For 64 bit CPU

4000	**34H**
4008	**65H**
4016	**86H**
4024	**93H**
4032	**45H**

(Here, no. of bytes of a word is 8, and the address of word is in multiples of 8)

ACCESSING CHARACTERS AND NUMBERS

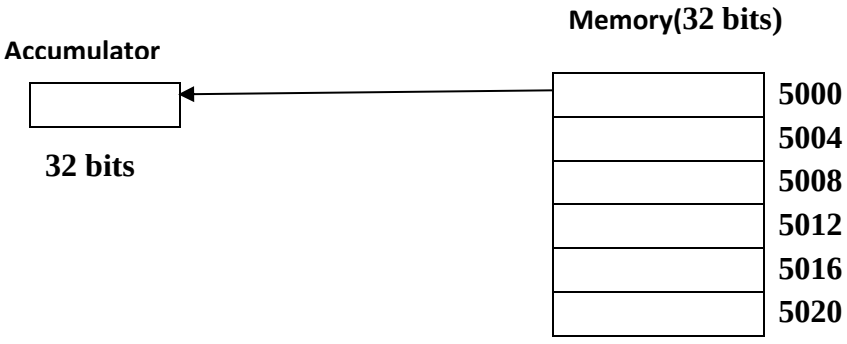
The character occupies 1 byte of memory and hence byte address for memory.
The numbers occupies 2 bytes of memory and hence word address for numbers.

6. MEMORY OPERATION

Both program instructions and operands are in memory.
To execute an instruction, each instruction has to be read from memory and after execution the results must be written to memory.

There are two types of memory operations namely 1. Memory read and 2. Memory write
Memory read operation [Load/ Read / Fetch]
Memory write operation [Store/ write]

1. ***MEMORY READ OPERATION:***
   - ✓ It is the process of transferring of 1 word of data from memory into Accumulator (GPR).
   - ✓ It is also called as Memory fetch operation.
   - ✓ The Memory read operation can be implemented by means of LOAD instruction.
   - ✓ The LOAD instruction transfers 1 word of data (1 word = 32 bits) from Memory into the Accumulator as shown in the fig.



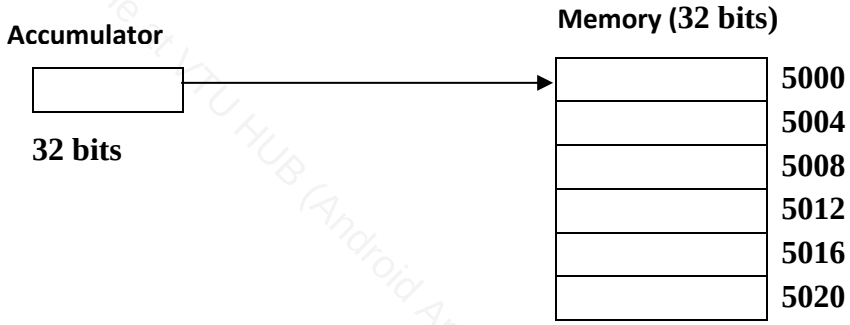
Steps for Memory Read Operation

(1) The processor loads MAR (Memory Address Register) with the address of the memory location.
(2) The Control unit of processor issues memory read control signal to enable the memory component for read operation.
(3) The processor reads the data from memory into the MDR (Memory Data Register) by means of bi-directional data bus.

[MAR] → Memory → MDR

2. MEMORY WRITE OPERATION

- It is the process of transferring the 1 word of data from Accumulator into the Memory.
- The Memory write operation can be implemented by means of STORE instruction. The STORE instruction transfers 1 word of data from Accumulator into the Memory location as shown in the fig.



Steps for Memory Write Operation

- The processor loads MAR with the address of the Memory location.
- The processor loads MDR with the data to be stored in Memory location.
- The Control Unit issues the Memory Write control signal.
- The processor transfers 1 word of data from MDR to Memory location by means of bi-directional data bus.

7. COMPUTER OPERATIONS (OR) INSTRUCTIONS AND INSTRUCTION EXECUTION

The Computer is designed to perform 4 types of operations, namely

- Data transfer operations
- ALU Operations
- Program sequencing and control.
- I/O Operations.

1. Data Transfer Operations

a) Data transfer between two registers.

Format: Opcode Source1 , Destination

The processor uses MOV instruction to perform data transfer operation between two registers The mathematical representation of this instruction is R1 → R2.

Ex : MOV R_1 , R_2 : R1 and R2 are the registers.

Where MOV is the operation code, R1 is the source operand and R2 is the destination operand. This instruction transfers the contents of R1 to R2.

EX: Before the execution of MOV R1,R2, the contents of R1 and R2 are as follows

R1 = 34h and R2 = 65h

After the execution of MOV R1, R2, the contents of R1 and R2 are as follows

R1 = 34H and R2 = 34H

b) Data transfer from memory to register

The processor uses **LOAD** instruction to perform data transfer operation from memory to register. The mathematical representation of this instruction is

ACC ←[LOCA]. Where ACC is the Accumulator.

Format : opcode operand

Ex: LOAD LOCA

For this instruction Memory Location is the source and Accumulator is the destination.

c) Data transfer from Accumulator register to memory

The processor uses **STORE** instruction to perform data transfer operation from Accumulator register to memory location. The mathematical representation of this instruction is

LOCA ←[ACC]. Where, ACC is the Accumulator.

Format: opcode operand

Ex: STORE LOCA

For this instruction accumulator is the source and memory location is the destination.

2. ALU Operations

The instructions are designed to perform arithmetic operations such as Addition, Subtraction, Multiplication and Division as well as logical operations such as AND, OR and NOT operations.

Ex1: ADD R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] + [R_1]$; Adds the content of R_0 with the content of R_1 and result is placed in R_1.

Ex2: SUB R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] - [R_1]$; Subtracts the content of R_0 from the content of R_1 and result is placed in R_1.

EX3: AND R_0, R_1 ; It Logically multiplies the content of R_0 with the content of R_1 and result is stored in R_1. ($R_1 = R_0$ AND R1)

3. **I/O Operations:** The instructions are designed to perform INPUT and OUTPUT operations. The processor uses MOV instruction to perform I/O operations.

The input Device consists of one temporary register called as DATAIN register and output register consists of one temporary register called as DATAOUT register.

a) Input Operation: It is a process of transferring one WORD of data from DATA IN register to processor register.

Ex: MOV DATAIN, R0

The mathematical representation of this instruction is as follows,

$R_0 \leftarrow$ [DATAIN]

b) Output Operation: It is a process of transferring one WORD of data from processor register to DATAOUT register.

Ex: MOV R_0, DATAOUT

The mathematical representation of this instruction is as follows,

$[R_0] \rightarrow$ DATAOUT

REGISTER TRANSFER NOTATION

There are 3 locations to store the operands during the execution of the program namely

1. Register 2. Memory location 3. I/O Port. Location is the storage space used to store the data.

- The instructions are designed to transfer data from one location to another location.

Eg 1 - Consider the first statement to transfer data from one location to another location

- " Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register R_0."
- The mathematical representation of this statement is given by

$R_0 \leftarrow$ [AMOUNT]

Eg 2 -Consider the second statement to add data between two registers

- "Add the contents of R_0 with the contents of R_1 and result is stored in R_2"
- The mathematical representation of this statement is given by

$R_2 \leftarrow [R_0] + [R_1]$.

Such a notation is called as "Register Transfer Notation".

It uses two symbols

1. A pair of square brackets [] to indicate the contents of Memory location and
2. ← to indicate the data transfer operation.

ASSEMBLY LANGUAGE NOTATION

Consider the first statement to transfer data from one location to another location

"Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register R_0."

The assembly language notation of this statement is given by

MOV AMOUNT, R_0

Opcode Source Destination

This instruction transfers 1 word of data from Memory location whose symbolic name is given by AMOUNT into the processor register R_0.

The mathematical representation of this statement is given by

$R_0 \leftarrow$ [AMOUNT]

Consider the second statement to add data between two registers

"Add the contents of R_0 with the contents of R_1 and result is stored in R_2"

The assembly language notation of this statement is given by

ADD R_0, R_1, R_2

Opcode source1, Source2, Destination

This instruction adds the contents of R_0 with the contents of R_1 and result is stored in R_2.

- The mathematical representation of this statement is given by

  $R_2 \leftarrow [R_0] + [R_1]$.

Such a notations are called as "Assembly Language Notations"

BASIC INSTRUCTION TYPES

There are 3 types of basic instructions namely

1. Three address instruction format
2. Two address instruction format
3. One address instruction format

Consider the arithmetic expression Z = A + B, Where A,B,Z are the Memory locations.

Steps for evaluation

1. Access the first memory operand whose symbolic name is given by A.
2. Access the second memory operand whose symbolic name is given by B.
3. Perform the addition operation between two memory operands.
4. Store the result into the 3rd memory location Z.
5. The mathematical representation is Z $\leftarrow$[A] + [B].

a) Three address instruction format : Its format is as follows

opcode	**Source-1**	**Source-2**	**destination**

Destination $\leftarrow$ [source-1] + [source-2]

Ex: ADD A, B, Z

Z $\leftarrow$ [A] + [B]

a) Two address instruction format : Its format is as follows

opcode	Source	Source/destination

Destination ← [source] + [destination]

The sequence of two address m/c instructions to evaluate the arithmetic expression Z ← A + B are as follows

```
MOV   A,   R0
MOV   B,   R1
ADD   R0,  R1
MOV   R1,  Z
```

b) One address instruction format : Its format is as follows

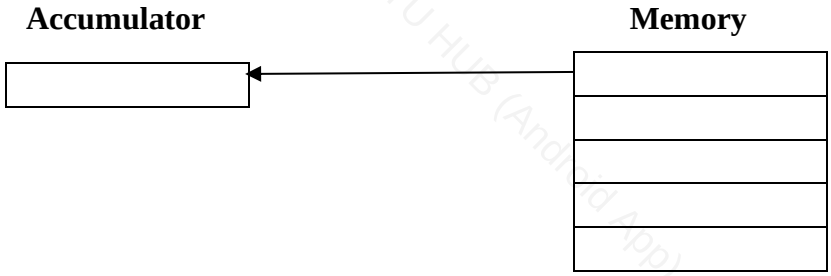
opcode	operand

Ex1: LOAD B

This instruction copies the contents of memory location whose symbolic name is given by 'B' into the Accumulator as shown in the figure.

The mathematical representation of this instruction is as follows

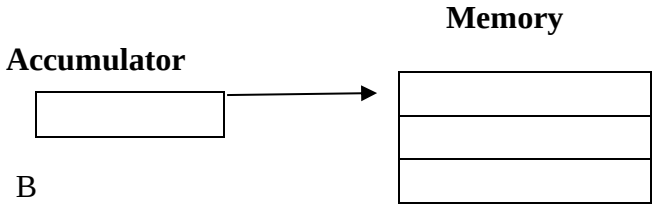
ACC ← [B]



Ex2: STORE B

This instruction copies the contents of Accumulator into memory location whose symbolic name is given by 'B' as shown in the figure. The mathematical representation is as follows

B ← [ACC].



Ex3: ADD B

- This instruction adds the contents of Accumulator with the contents of Memory location 'B' and result is stored in Accumulator.
- The mathematical representation of this instruction is as follows

ACC ←[ACC]+ [B]

STRIGHT LINE SEQUENCING AND INSTRUCTION EXECUTION

Consider the arithmetic expression

C = A+B , Where A,B,C are the memory operands.

The mathematical representation of this instruction is

C = [A] + [B].

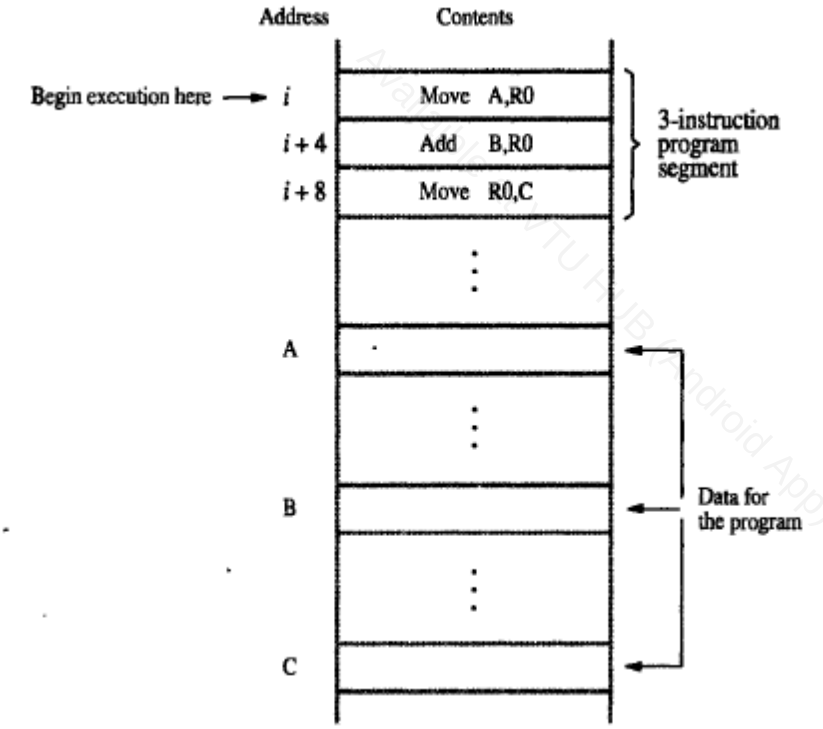
The sequence of instructions using two address instruction format are as follows

MOV A, R_0
ADD B, R_0
MOV R_0, C

Such a program is called as 3 instruction program.

NOTE: The size of each instruction is 32 bits.

- The 3 instruction program is stored in the successive memory locations of the processor is as shown in the fig.



- The system bus consists of uni-directional address bus,bi-directional data bus and control bus

"It is the process of accessing the 1st instruction from memory whose address is stored in program counter into Instruction Register (IR) by means of bi-directional data bus and at the same time after instruction access the contents of PC are incremented by 4 in order to access the next instruction. Such a process is called as "Straight Line Sequencing".

INSTRUCTION EXECUTION

There are 4 steps for instruction execution

1 Fetch the instruction from memory into the Instruction Register (IR) whose address is stored in PC.

IR ← [[PC]]

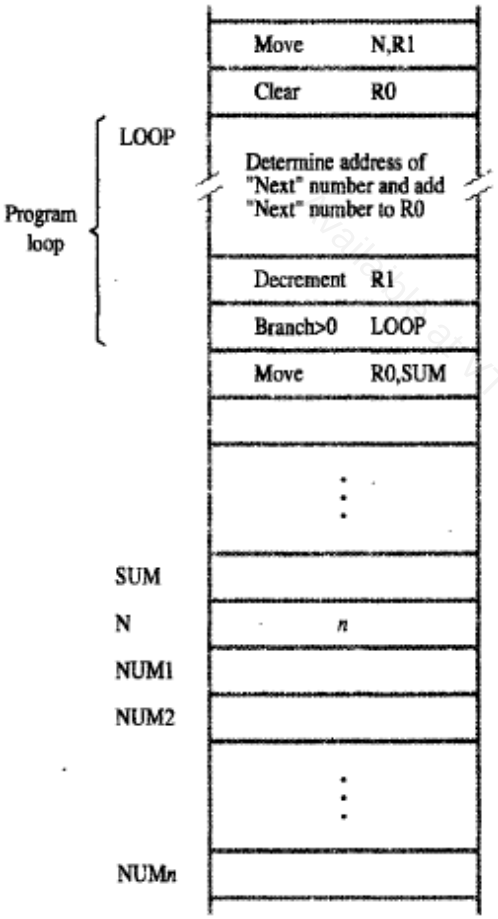
2 Decode the instruction.
3 Perform the operation according to the opcode of an instruction
4 Load the result into the destination.
5 During this process, Increment the contents of PC to point to next instruction (In 32 bit machine increment by 4 address)

PC ← [PC] + 4.

6 The next instruction is fetched, from the address pointed by PC.

BRANCHING

Suppose a list of 'N' numbers have to be added. Instead of adding one after the other, the add statement can be put in a loop. The loop is a straight-line of instructions executed as many times as needed.



The 'N' value is copied to R1 and R1 is decremented by 1 each time in loop. In the loop find the value of next elemet and add it with Ro.

In conditional branch instruction, the loop continues by coming out of sequence only if the condition is true. Here the PC value is set to 'LLOP' if the condition is true.

Branch > 0 LOOP // if >0 go to LOOP

The PC value is set to LOOP, if the previous statement value is >0 ie. after decrementing R1 value is greater than 0.

If R1 value is not greater than 0, the PC value is incremented in a mormal sequential way and the next instruction is executed.

CONDITION CODE BITS

- The processor consists of series of flip-flops to store the status information after ALU operation.
- It keeps track of the results of various operations, for subsequent usage.
- The series of flip-flip-flops used to store the status and control information of the processor is called as "Condition Code Register". It defines 4 flags. The format of condition code register is as follows.

C	V	Z	N

1 N (NEGATIVE) Flag:
It is designed to differentiate between positive and negative result.
It is set 1 if the result is negative, and set to 0 if result is positive.

2 Z (ZERO) Flag:
It is set to 1 when the result of an ALU operation is found to zero, otherwise it is cleared.

3 V (OVER FLOW) Flag:
In case of 2^s Complement number system n-bit number is capable of representing a range of numbers and is given by -2^{n-1} to $+2^{n-1.}$. The Over-Flow flag is set to 1 if the result is found to be out of this range.

4 C (CARRY) Flag :
This flag is set to 1 if there is a carry from addition or borrow from subtraction, otherwise it is cleared.

8. Addressing Modes

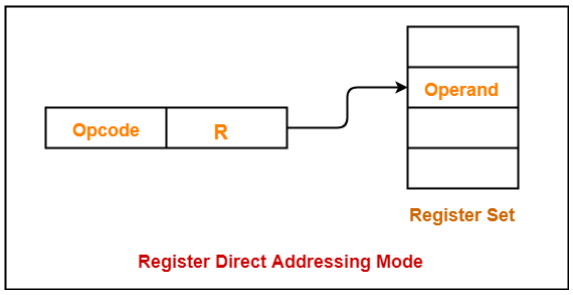
The various formats of representing operand in an instruction or location of an operand is called as "Addressing Mode". The different types of Addressing Modes are

a) Register Addressing
b) Direct Addressing
c) Immediate Addressing
d) Indirect Addressing
e) Index Addressing
f) Relative Addressing
g) Auto Increment Addressing
h) Auto Decrement Addressing

a. REGISTER ADDRESSING:

In this mode operands are stored in the registers of CPU. The name of the register is directly specified in the instruction.

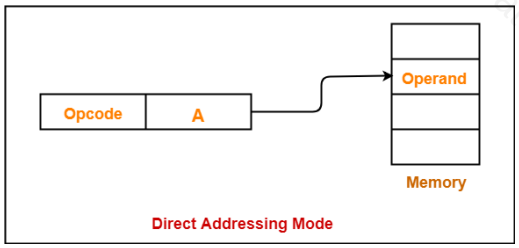
Ex: MOVE R_1,R_2 Where R1 and R2 are the Source and Destination registers respectively. This instruction transfers 32 bits of data from R1 register into R2 register. This instruction does not refer memory for operands. The operands are directly available in the registers.



b. DIRECT ADDRESSING

It is also called as Absolute Addressing Mode. In this addressing mode operands are stored in the memory locations. The name of the memory location is directly specified in the instruction.

Ex: MOVE LOCA, R_1 : Where LOCA is the memory location and R1 is the Register.

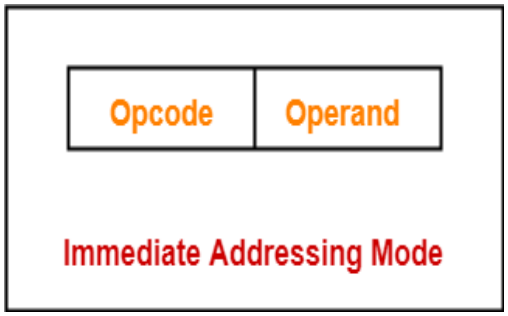


This instruction transfers 32 bits of data from memory location LOCA into the General Purpose Register R1.

c. IMMEDIATE ADDRESSING

In this Addressing Mode operands are directly specified in the instruction. The source field is used to represent the operands. The operands are represented by # (hash) sign.

Ex: MOVE #23, R0



d. INDIRECT ADDRESSING

In this Addressing Mode effective address of an operand is stored in the memory location or General Purpose Register.

[Effective address (EA) – the actual memory address of the operand]

The memory locations or GPRs are used as the memory pointers.

Memory pointer: It stores the address of the memory location.

There are two types Indirect Addressing

i) Indirect through GPRs
ii) Indirect through memory location

i) Indirect Addressing Mode through GPRs

In this Addressing Mode the effective address of an operand is stored in the one of the General

Purpose Register of the CPU.

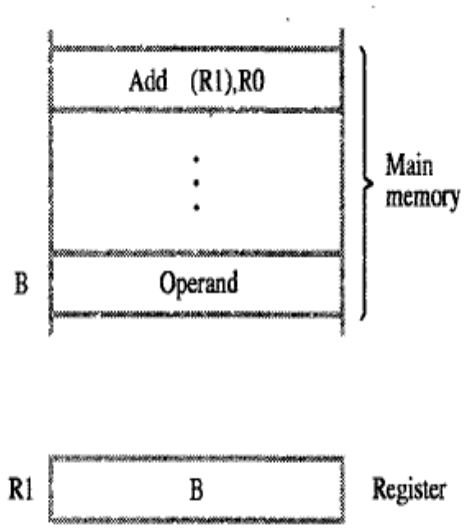
Ex: ADD (R_1), R_0 ; Where R_1 and R_0 are GPRs

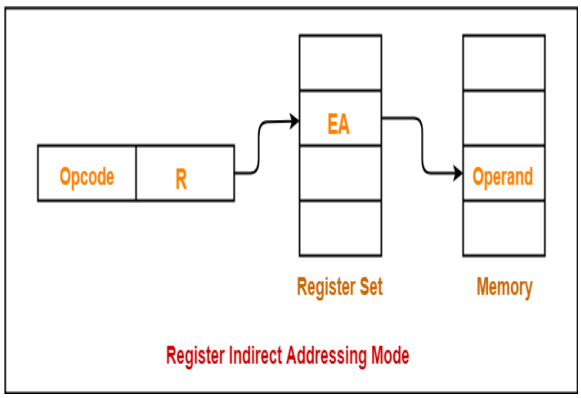
(R1) – R1 stores the address of a location where operand value is present.

This instruction adds the data from the memory location whose address is stored in R_1, with the contents of R_0 Register and the result is stored in R_0 register as shown in the fig.

$R_0 \leftarrow [[R1]] + R_0$

The diagrammatic representation of this addressing mode is as shown in the fig.





ii) Indirect Addressing Mode through Memory Location.

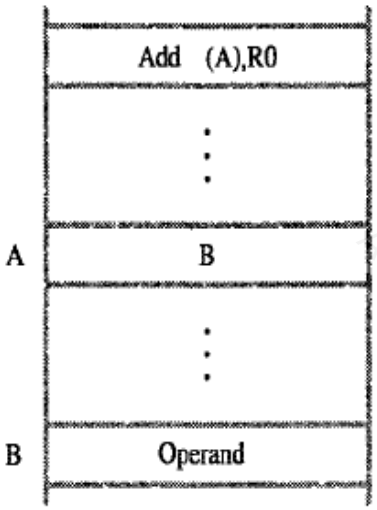
In this Addressing Mode, effective address of an operand is stored in the memory location.

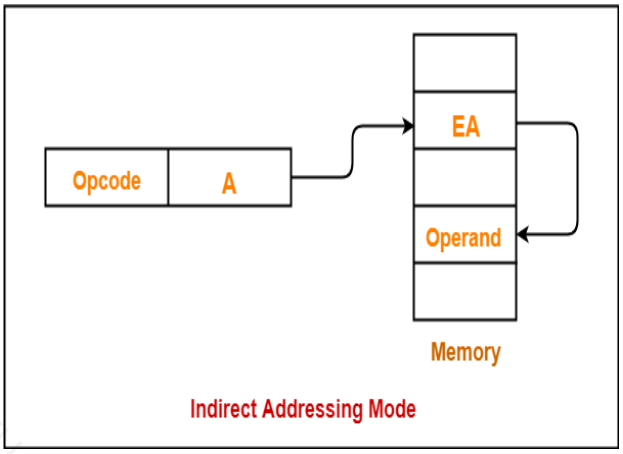
Ex: ADD (A), R_0

This instruction adds the data from the memory location, whose address is stored in 'A' memory location with the contents of R_0 and result is stored in R_0 register.

$R_0 \longleftarrow [[A]] + R_0$

The diagrammatic representation of this addressing mode is as shown in the fig.





e. INDEX ADDRESSING MODE

In this addressing mode, the effective address of an operand is computed by adding constant value with the contents of Index Register. Any one of the General Purpose Register namely R_0 to R_{n-1} can be used as the Index Register. The constant value is directly specified in the instruction.

The symbolic representations of this mode are as follows

1. X (R_i) where X is the Constant value and R_j is the GPR.
   It can be represented as
   Effective Address (EA) of an operand = X + (R_i)
   Eg: Add 5(R2) , R3
   Effective Address(EA) of first operand = 5 + [R2].
2. (R_i , R_J) Where R_i and R_j are the General Purpose Registers used to store addresses of an operand and constant value respectively. It can be represented as
   The EA of an operand is given by EA = (R_i) + (R_j)

3. X (R_i , R_j) Where X is the constant value and R_I and R_J are the General Purpose Registers used to store the addresses of the operands.It can be represented as

The EA of an operand is given by

EA = (R_i) + (R_j) + X

Eg : Add 5(R1)(R2) , R3

EA of first operand is [R1]+[R2]+5

There are two types of Index Addressing Modes

i) **Offset is given as constant.**

ii) **Offset is in Index Register.**

Note : Offset : It is the difference between the starting effective address of the memory location and the effective address of the operand fetched from memory.

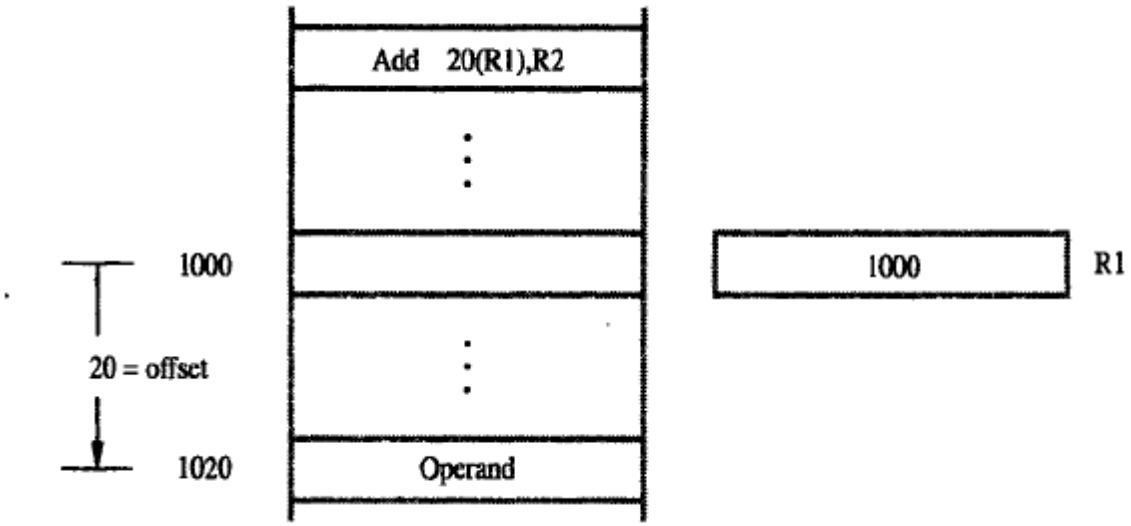
i)Offset is given as constant

Ex: ADD 20(R_1), R_2

The EA of an operand is given by

EA = 20 + [R_1]

This instruction adds the contents of memory location whose EA is the sum of contents of R_1 with 20 and with the contents of R_2 and result is placed in R_2 register. The diagrammatic representation of this mode is as shown in the fig.



ii) Offset is in Index Register

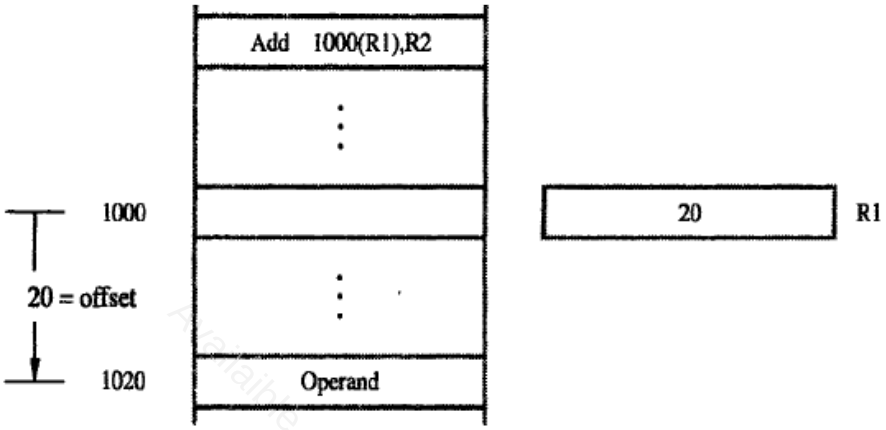
Ex: ADD 1000(R_1) , R_2 R_1 holds the offset address of an operand.

The EA of an operand is given by

EA = 1000 + [R_1]

This instruction adds the data from the memory location whose address is given by [1000 + [R1] with the contents of R_2 and result is placed in R_2 register.

The diagrammatic representation of this mode is as shown in the fig.



f. RELATIVE ADDRESSING MODE:

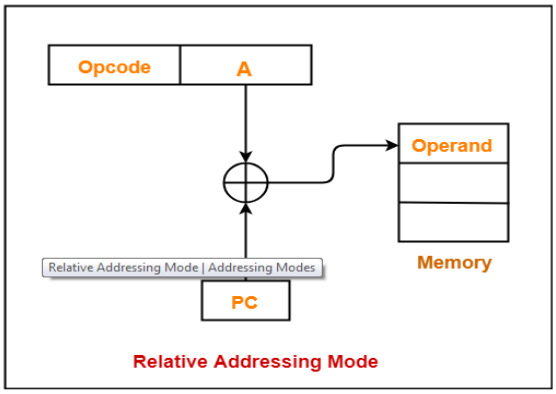
In this Addressing Mode EA of an operand is computed by the Index Addressing Mode. This Addressing Mode uses PC (Program Counter) to store the EA of the next instruction instead of GPR.

The symbolic representation of this mode is X(PC), where X is the offset value and PC is the Program Counter to store the address of the next instruction to be executed.

EA of operand = X + (PC).

This Addressing Mode is useful to calculate the EA of the target memory location.

Effective Address
= Content of Program Counter + Address part of the instruction



g. AUTO INCREMENT ADDRESSING MODE

In this Addressing Mode , EA of an operand is stored in the one of the GPRs of the CPU. This Addressing Mode increment the contents of register, to point to next memory locations after operand access.

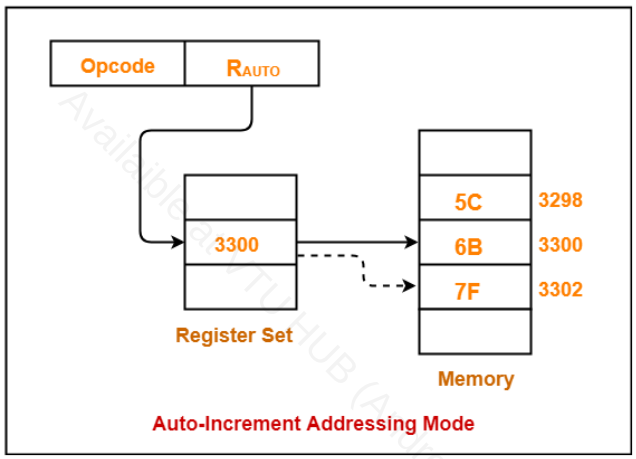
In 32- bit machine, it points to the next memory location, by adding 4 to current location value.

The symbolic representation is

$(R_I)+$ Where R_i is the one of the GPR.

Ex: MOVE $(R_1)+$, R_2

This instruction transfer's data from the memory location whose address is stored in R_1 into R_2 register and then it increments the contents of R_1 to point to next address.



h. AUTO DECREMENT ADDRESSING MODE

In this Addressing Mode , EA of an operand is stored in the one of the GPRs of the CPU. This Addressing Mode decrements the contents of register, to point to previous memory locations after operand access.

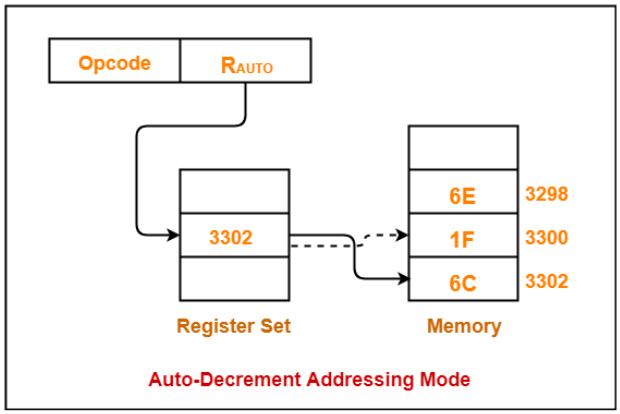
In 32- bit machine, it points to the previous memory location, by subtracting 4 from current location value.

The symbolic representation is

$-(R_I)$ Where R_i is the one of the GPR.

Ex: MOVE $-(R_1)$, R_2

This instruction first decrements the contents of R_1 by 4 memory locations and then transfer's data of that location to destination register.



9. ASSEMBLY LANGUAGE

- The Assembly language uses Symbolic names to represent opcodes ,memory locations and registers.
- The Assembler converts Assembly language programs into machine level language programs.
- The Assembly language is called as "Source program" and m/c language program is called as"Object program".
- The Assembler converts source program into object program.
- A set of symbolic names and set of rules for their use forms a programming language and is called as "Assembly Language".

Ex: MOV, ADD, LOAD → Opcodes.

X, Y, AMOUNT → Memory locations.

Where R0 to R7 are the registers.

- The examples for Assembly Language instructions are as follows

MOV R_0, R_1 ; Register Addressing.

MOV #23, R_0 ; Immediate Addressing.

9.1 DIRECTIVES

There are two types of instructions namely i) Processor Instructions and ii) Assembler Instructions.

- The Processor instructions are converted into m/c instructions by means of Assembler. Hence Assembler generates m/c code for processor instructions.
- The Assembler instructions are not converted into m/c instructions and hence Assembler does not generate the m/c code for Assembler instructions

" **A set of commands given to Assembler while converting source program into object program is called as Assembler Directive".**

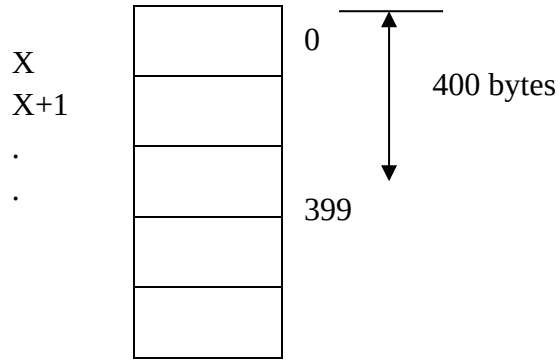
TYPES OF ASSEMBLER DIRECTIVES

1. RESERVE

This directive is used to allocate a block of memory. This block of memory is used only for data.

```
X       RESERVE       400
```

This directive reserves 400 bytes of memory whose symbolic name is X as shown in the fig.



2.EQU directive

This directive is used to assign numerical values to symbolic names during the execution of the assembly language program.

The following code describes the use of EQU directive.

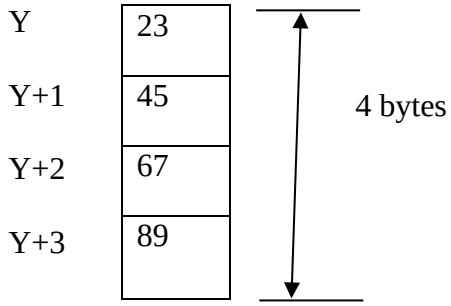
```
 X          EQU      32
MOVE       #23,      R0
 MOVE       X,       R1
 ADD        R0,      R1
```

3. DATA WORD

This directive is used to allocate 4 bytes of memory.

```
Y          DATAWORD          23456789
```

The memory representation of this directive is as shown in the fig.



4.ORIGIN DIRECTIVE

This directive converts source program into object program. The object program is loaded into memory for execution by means of loader. The Origin directive is used to assign the successive addresses for sequence of instructions or operands.

5. END DIRECTIVE

This Directive is used to terminate the Assembly level language program. The Assembler ignores the execution of instructions after the execution of this directive.

END

9.2 NUMBER REPRESENTATION

There are 3 types of numbers namely

a. Decimal Numbers (0 to 9)
b. Binary Numbers (0 or 1)
c. Hexadecimal Numbers (0 to 9 and A to F)

The 32 bit processor is capable of performing ALU operations on three types of operands namely decimal, binary and hexadecimal

The representation of all these operands is as follows

For decimal	MOVE	#23,	R0
For binary	MOVE	#% 01100011,	R2
For hexadecimal	MOVE	#$24,	R4

10. BASIC INPUT AND OUTPUT OPERATIONS

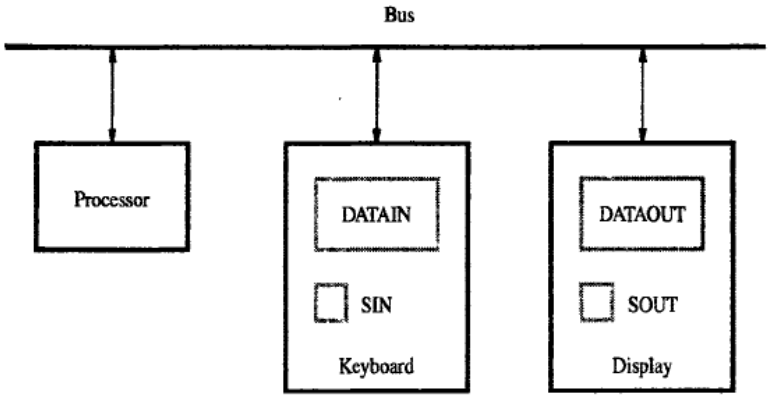
The simple arrangement of connecting i/p and o/p devices into the processor is as shown in the fig

The Processor performs two operations with respect to i/o device namely

i) Input operation and

ii)Output operation

i)**Input operation:** It is the process of reading the data or instructions from the input device. The I/O subsystem consists of block of instructions to perform i/p operation.



ii)**Output operation:** It is the process of writing the data or instructions into the output device. The I/O subsystem consists of block of instructions to perform o/p operation

Consider a problem of transferring 1 byte of data from i/p device to o/p device. The i/p device transfers few characters/sec. The data transfer rate of i/p device is expressed in terms of few characters/sec. Similarly o/p device transfers thousands of characters to o/p device for display. The processor is capable of executing millions of instructions per second.

From the above analysis it is clear that the processing and transfer speed varies in different devices. So the devices must be synchronized.

DATAIN and DATAOUT are the two registers of input and output devices respectively. These are the buffers of input and output devices.

To provide the synchronization between Processor, i/p device and o/p device it is necessary to follow the several steps are as follows

Steps to provide synchronization between Processor and i/p device

1) SIN=0 indicates that, no data is read from keyboard yet. The processor has to wait, as the DATAIN register is empty.

2) When a character is pressed on the keyboard, the ASCII value of a character is stored in DATAIN register and hence SIN flag is set to 1.

3) When SIN = 1, the processor reads the ASCII value of a character from DATAIN register into the Processor register.

4) After reading, the SIN flag is reset to 0.

```
READWAIT    if SIN = 0
            Branch  to READWAIT          //No data to read
            Input data from DATAIN to R1 //Data is read
```

Steps to provide synchronization between Processor and O/p device

1. SOUT = 0 indicates that, there is already some data to output. The buffer DATAOUT is full.
2. When SOUT =1, the output device is ready to take the characters to be displayed.
3. When the o/p device is ready to display the character , the Processor transfers the character code from the processor register into the DATAOUT register.
4. The SOUT flag is cleared, after the character is transferred to o/p device and is free to accept new data.

```
WRITEWAIT    if SOUT = 0
             Branch  to WRITEWAIT          //No data to read
             Output data from R1 to DATAIN  //Data is read
```

The i/p operation can be implemented as follows

Let R_0 be the Processor register and DATAIN be the internal register of the i/p device.

MOVE DATAIN, R0

The o/p operation can be implemented as follows

Let R_0 be the Processor register and DATAOUT be the internal register of the O/p device.

MOVE R_0, DATAOUT

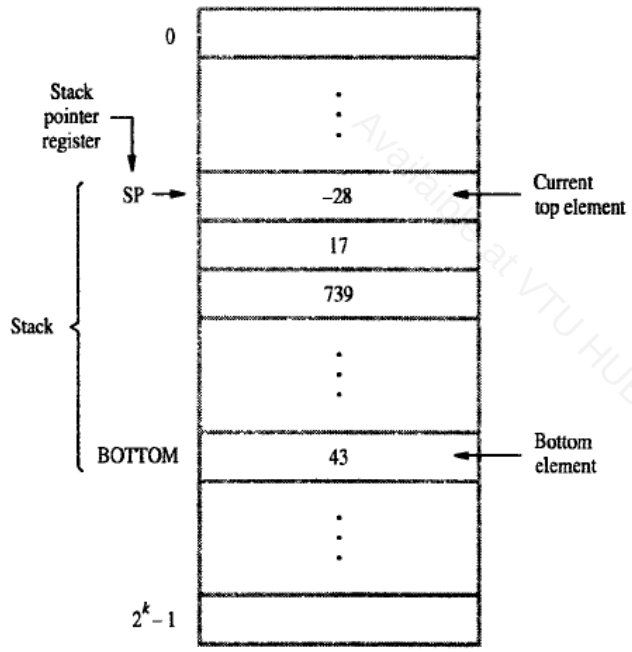
A program that reads a line of characters and displays it.

	Move	#LOC , R0	Initialize R0 to a memory location
READ	TestBit	#3, INSTATUS	if bit #3 of keyboard buffer , is 0. Then no data is entered from keyboard
	Branch=0	READ	
	MoveByte	DATAIN, (R0)	Move the characters from keyboard buffer to memory location pointed by R0.
ECHO	TestBit	#3, OUTSTATUS	if bit #3 of display buffer , is 1. Then no data to display.
	Branch=0	ECHO	
	MoveByte	(R0), DATAOUT	Move the characters from memory Location pointed by R0, to display buffer.
	Compare	#CR, (R0)+	If the read data is carriage return,
	Branch!=0	READ	then stop reading.

[Compare returns 0, if compared data are equal.]
The above program, reads the data from keyboard to memory location pointed by R0 and displays the data from memory location to output device.
If R0 value is carriage return (empty key pressed), stop reading.
If Ro value is some other value, Compare returns a non-zero value and reading is continued.
(R0)+ R0 points to next memory location, so that new value is stored in next locations.

11. STACKS AND QUEUES

STACK - A stack is a Data Structure, in which the accessing is restricted at only one end of the stack. It is similar to a bottle, in which elements can be added and removed from the same end. The end of the stack, from which elements can be added or removed is called the top of the stack and the other end is called the bottom of the stack.



It works on the principle of LIFO (Last In First Out), the last item placed on the stack is the first to be removed. The term 'push' and 'pop' are used to describe the placing a new item on stack and removing the top item from the stack.

Assume that the first element is placed in the location BOTTOM, and when new elements are pushed to the stack, they are placed in successive lower addresses.

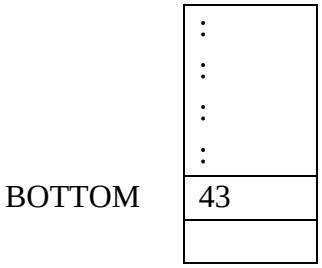
A processor register is used to keep track of the address of the element that is at the top of stack at any time. This register is called Stack Pointer **(SP).**

In the above figure, the SP pointer is currently pointing to the topmost value -28. To add a new element, the SP will decrement its value by 1 address, so as to point at next location and add the new value.

PUSH Operation – Subtract #4, SP
MOV NEWITEM, (SP)

The subtract instruction subtracts the SP value by 4, now SP points to the next lower address. The MOV instruction moves the new element to the address location stored in SP.

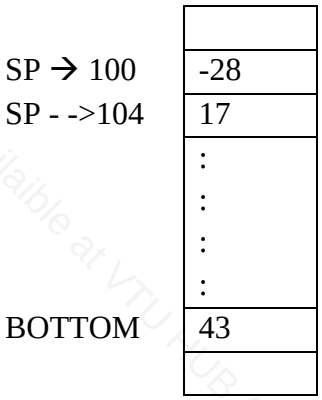
SP - -> 96	
SP → 100	-28



The push operation, using auto-decrement instruction can be written as –
MOV NEWITEM, - (SP)

POP Operation - MOV (SP), ITEM
ADD #4, SP

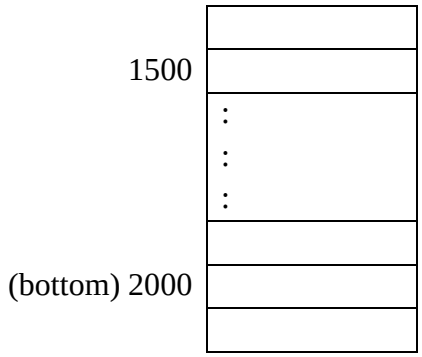
The MOV instruction moves the element at the location pointed by SP to ITEM and the SP pointer is moved to the next higher address, so that it points to the new top element.



The POP operation, using auto-increment instruction can be written as –
MOV (SP)+, ITEM

Condition checking in stack –
When a stack is used in program, the space allocated to stack is fixed. When stack pointer (SP) reaches maximum size, no more values can be pushed.

Similarly, when there are no elements in stack, elements cannot be popped.



Suppose the stack is located from address 2000(Bottom) to address 1500. The stack pointer is initially pointed to 2004. [to push an element we decrement SP & then add the element].

Conditions to be checked before push and pop operations –

1) Check for stack full condition , before adding the element to stack
2) Check for stack empty condition, before poping the element from the stack.

PUSH operation -

```
COMPARE #1500, SP
Branch<0 ERROR
MOV NEWITEM,-(SP)
```

Compare, address value in SP register and value 1500, if SP is lessthan than 1500, then push operation is not possible. If SP value is greater than 1500, push operation can be performed.

POP operation -

```
COMPARE #2000,SP
Branch>0 ERROR
MOV (SP)+, ITEM
```

Compare, address value in SP register and value 2000, if SP is greater than 2000, then pop operation is not possible. If SP value is greater than 2000, pop operation can be performed.

QUEUE – A queue is a Data Structure that works on the principal of FIFO (First In First Out) ie., data that are stored first are retrieved first on FIFO basis.

The elements are added at one end (IN) and retrieved from other end (OUT). In stack, one end is fixed where as in queue both ends are pointed by pointers and both end changes its location. One end is used to add items and other end is used to delete items.

12. SUBROUTINE

Subfunctions in a program necessary to perform a particular subtask is called a subroutine.
Eg: Subroutine to sort a list of numbers,
Subroutine to add the given numbers etc.

In a program, the subroutines can be called from different locations and different functions. When a program branches to a subroutine, then it is calling the subroutine. The instruction that perform this branch operation is called **call instruction**.

Whenever the subroutine is called, the execution starts from the starts from the starting address of subroutine. After its execution, the execution of calling function is resumed from the location where it called the subroutine. Hence the content of PC is stored before moving to the subroutine.

The way in which a computer calls and returns from a subroutine is called **subroutine linkage method**. The return address is stored in **link register**. After the execution of subroutine, the return instruction returns to the calling program by using the link register.

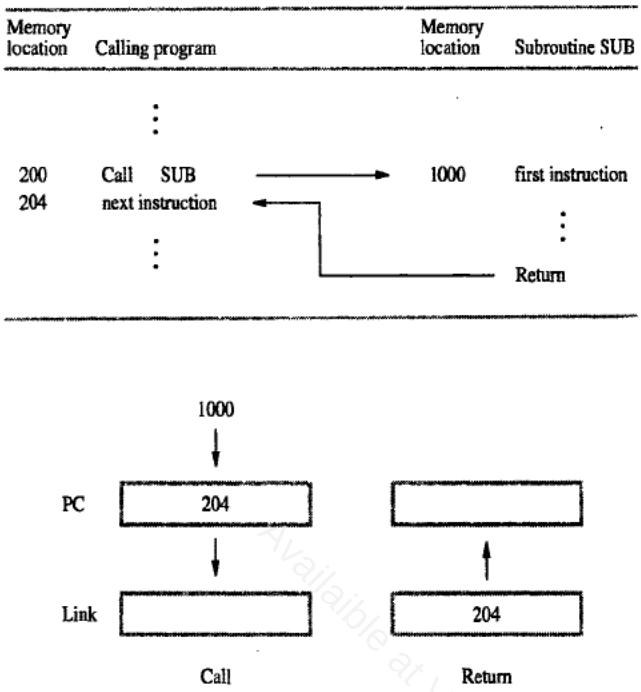


Figure 2.24 Subroutine linkage using a link register.

The Call instruction is a special branch instruction –

- It stores the content of PC in link register
- ores the specified subroutine address in PC and branch to that address.

The Return instruction of subroutine is a special branch instruction –

- It branches to the address contained in link register.

One subroutine can call another and that can call another subroutine. This is called subroutine nesting. When the second subroutine is called, the link register will have its address and the first subroutine address is lost. So it is stored in processor stack, pointed by SP (Stack Pointer).
When function is called, the content of PC is put to the stack, when return instruction is called , the content of stack is popped out.

Parameter passing – Parameters must be passed to a subroutine and also the results must be sent to the calling function. The parameters can be passed in the following ways –

- Place parameters in registers
- Place parameters in memory locations
- Place parameters on processor stack.

13. ADDITIONAL INSTRUCTIONS

There are 3 types

a. Logical instructions – NOT, AND , OR

b. Shift instructions – Logical Shift Left, Logical Shift Right, Arithmetic Shift right

c. Rotate instructions – Rotate Left with carry, Rotate Left without carry, Rotate Right with carry, Rotate Right without carry.

a. Logical instructions

The processors are designed to perform logical operations such as AND,OR and NOT operations.

i) **NOT instruction**

Format: opcode destination

The opcode specifies operation to be performed and destination specifies the operand. The operand can be register operand or memory operand.

Ex 1: NOT R0

It performs the function of complementation. It is the process of converting binary bit 0 into binary bit 1 and vice versa.

The illustration of this instruction is as follows.

Before instruction execution	**After instruction execution**
R0 = 10101100	R0 = 01010011.

ii) AND instruction

It performs the function of logical AND operation.

Format: opcode source, destination

The opcode specifies operation to be performed and source & destination specifies the operands. The operands can be register operand or memory operand.

Ex:1 AND R3, R0

This instruction logically ANDs the contents of R3 with the contents of RO and result is stored in R0 register.

The illustration of this instruction is as follows.

Before instruction execution	**After instruction execution**
R3 = 0011 0100	R3 = 0011 0100

R0 = 1010 1100 R0 = 00100100

iii) OR instruction

It performs the function of logical OR operation.

Format: opcode source, destination

Ex: 1 OR R3, R0

The opcode specifies operation to be performed and source & destination specifies the operands. The operands can be register operand or memory operand.

This instruction logically ORs the contents of R3 with the contents of RO and result is stored in R0 register.

The illustration of this instruction is as follows.

Before instruction execution	**After instruction execution**
R3 = 0011 0100	R3 = 0011 0100
R0 = 1010 1100	R0 = 1011 1100

b. Shift instructions

The shift instructions are designed to shift the contents of processor register or memory location to left or right according to the number of bits specified in the first operand (count).

There are 3 types of shift instructions.

1. Logical Shift Left.
2. Logical Shift Right.
3. Arithmetic Shift Right

1. Logical Shift Left

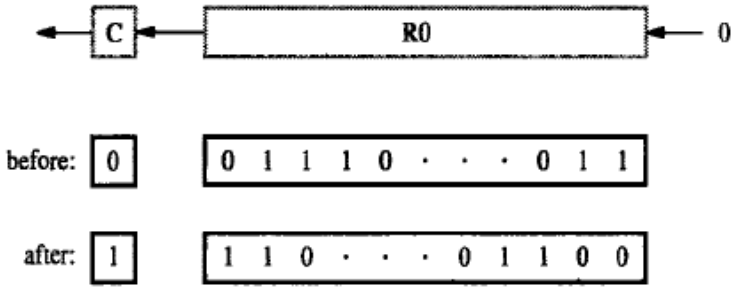
Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand.

Ex: LshiftL #2, R0

This instruction shifts the contents of register R0 to left through carry flag by 2 bits. The count value may be directly specified in the instruction as an immediate operand or stored in a register. The carry bit is initialized to '0'(zero).

The contents of R0 before and after the execution of this instruction are as shown in the fig. The **shifted positions are filled with zeros from right side** as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

C	R0
0	1010 1100

If the instruction is **LShiftL #2, R0**

After first time bit shifting R0 and carry flag values are

C	R0
1	0101 1000

After second time bit shifting R0 and carry flag values are

(final result of LShiftL #2,R0)

C	R0
0	1011 0000

2.Logical Shift Right

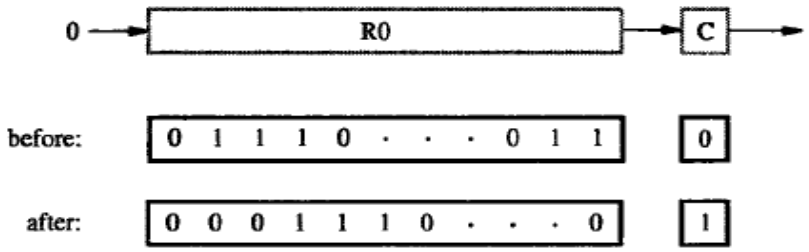
Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand.

Ex: LshiftR #2, R0

This instruction shifts the contents of register R0 to right through carry by 2 bits. The count value directly specified in the instruction as an immediate operand or stored in a register. The carry bit is initialized to '0'(zero).

The contents of R0 before and after the execution of this instruction are as shown in the fig. The **shifted positions are filled with zeros from left side** as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

R0	C
1010 1100	**0**

If the instruction is **LShiftR #2, R0**

After first time bit shifting R0 and carry flag values are

R0	C
0101 0110	**0**

After second time bit shifting R0 and carry flag values are

(**final result of LShiftR #2,R0)**

R0	C
0010 1011	**0**

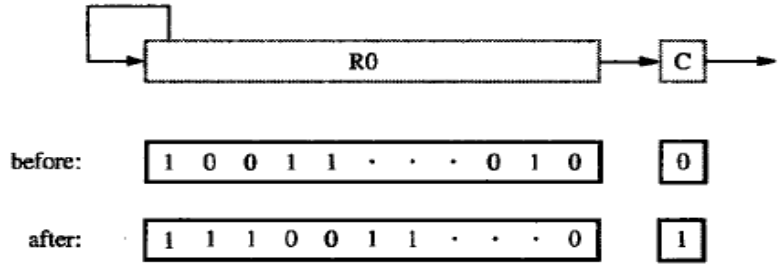
3.Arithmetic Shift Right

Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand.

Ex: AShiftR #3, R0

This instruction is designed **to preserve the sign bit**. This instruction shifts the contents of register or memory location to **right through carry**, by number of bits specified in the 'count'. After each shift it **copies leftmost bit to Most Significant Bit**. The contents of R0 before and after the execution of this instruction are as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

R0	C
1010 1100	**0**

If the instruction is **AShiftR #3, R0** (As first operand is #3, three bits shifting must be done)

After first time bit shifting R0 and carry flag values are

R0	C
1101 0110	**0**

After second time bit shifting R0 and carry flag values are

R0	C
1110 1011	**0**

After third time bit shifting R0 and carry flag values are

(final result of AShiftR #3,R0)

R0	C
1111 0101	**1**

c.Rotate instructions

The Rotate instructions are designed to rotate the contents of register or memory location to left or right according to the number of bits specified in the first operand (count).

There are 4 types of Rotate instructions.

1. Rotate left without carry

2. Rotate left with carry
3. Rotate right without carry
4. Rotate right with carry

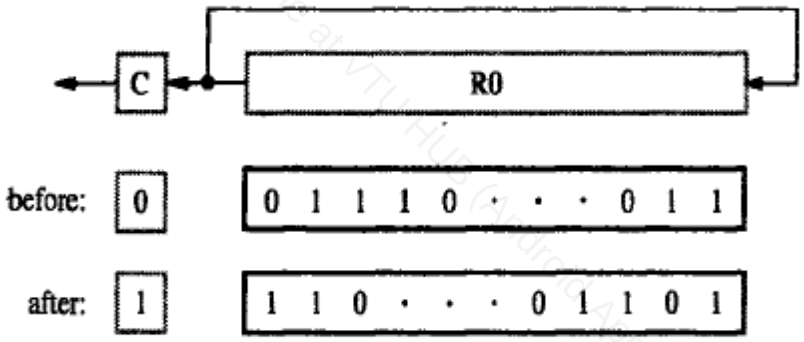
1. Rotate left without carry(RotateL, ROL)

Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand. The carry flag is initialized to 0(zero).

Ex: RotateL #1, R0

This instruction rotates the contents of register R0 **to left without carry** by 2 bits as shown in the fig. The Most Significant Bits are transferred to Least Significant Bits are as shown in the fig. The contents of register R0 before and after the execution of this instruction is as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

C	R0
0	1010 1100

If the instruction is **RotateL #1, R0** (only ones rotation of bits must be done)

After first time bit shifting R0 and carry flag values are

(f**inal result of RotateL #1,R0**)

C	R0
1	0101 1001

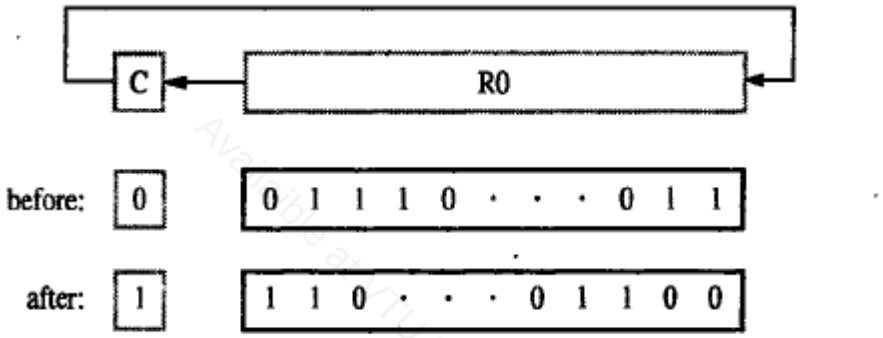
2. Rotate left with carry

Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand. The carry flag is initialized to 0(zero).

Ex: RotateLC #2, R0

This instruction rotates the contents of register R0 **to left with carry** by 2 bits as shown in the fig. **The Most Significant Bits are transferred to carry and then transferred to Least Significant Bits** are as shown in the fig. The contents of register R0 before and after the execution of this instruction is as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

C	R0
0	1010 1100

If the instruction is **RotateLC #2, R0** (two times rotation of bits must be done)

After first time bit shifting R0 and carry flag values are

C	R0
1	0101 1000

After second time bit shifting R0 and carry flag values are

(f**inal result of RotateLC #2,R0**)

C	R0
0	1011 0001

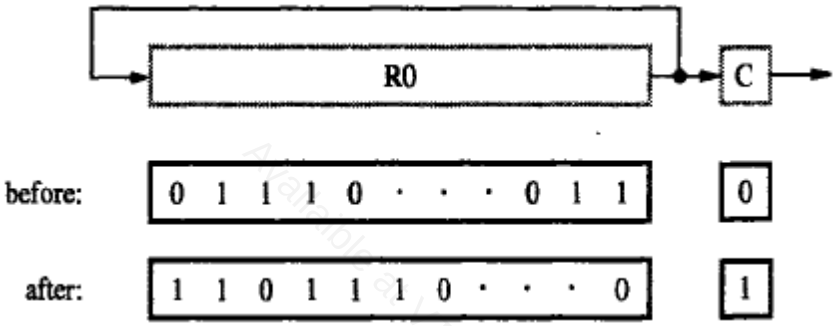
3. Rotate right without carry

Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand. The carry flag is initialized to 0(zero).

Ex: RotateR #2, R0

This instruction rotates the contents of register R0 to right without carry by 2 bits as shown in the fig. **The Least Significant Bits are transferred to Most Significant Bits** are as shown in the fig. The contents of register R0 before and after the execution of this instruction is as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

R0	C
1010 1110	**0**

If the instruction is **RotateR #2, R0** (As first operand is #2, two bits shifting must be done)

After first time bit shifting R0 and carry flag values are

R0	C
0101 0111	**0**

After second time bit shifting R0 and carry flag values are

(final result of RotateR #2,R0)

R0	C
1010 1011	**1**

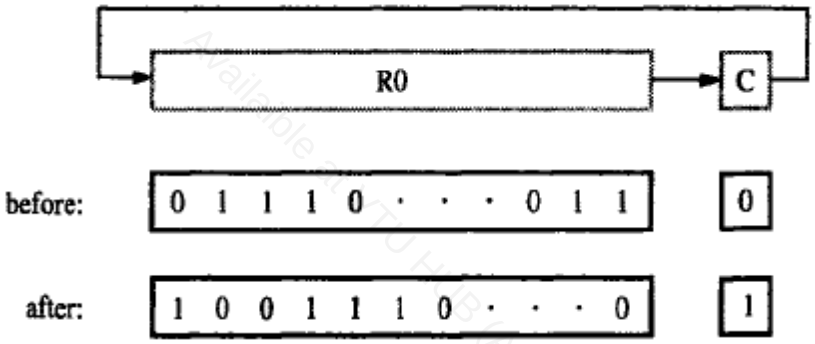
4. Rotate right with carry

Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand. The carry flag is initialized to 0(zero).

Ex: RotateRC #2, R0

This instruction rotates the contents of register R0 to **right with carry** by 2 bits as shown in the fig. **The Least Significant Bits are transferred to carry and then transferred to Most Significant Bits** are as shown in the fig.The contents of register R0 before and after the execution of this instruction is as shown in the fig.



The illustration of this instruction is as follows.

Before instruction execution

R0	C
1010 1110	**0**

If the instruction is **RotateRC #2, R0** (As first operand is #2, two bits shifting must be done)

After first time bit shifting R0 and carry flag values are

R0	C
0101 0111	**0**

After second time bit shifting R0 and carry flag values are

(final result of RotateR #2,R0)

R0	C
0010 1011	**1**

14. ENCODING OF MACHINE INSTRUCTIONS.

A list of instructions is called as program. To execute a program in processor, instructions must be encoded into a compact binary form. Such encoded instructions are called as Machine instructions. The instructions that use symbolic names are called as "Assembly Language ".The **Assembler** converts Assembly Language instructions into Machine Language instructions.

Consider few instructions to perform operations such as add, sub, multiply, shift, branch. These instructions may use operands of different size such as 32 bit , 8 bit or 16 bit. The type of operation to be performed and type of operand will be specified by encoded binary pattern.

The three instruction formats in computers -

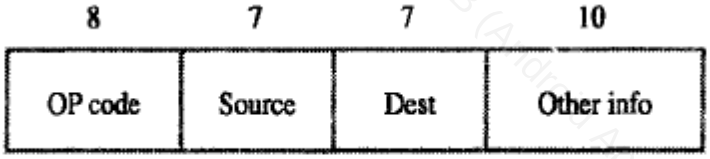
a) One word instruction format.

b) Two word instruction format.

c) Three word instruction format.

a) One word instruction format

The one word instruction format in 32 bit machine is as shown in the fig



Ex:1 ADD R0, R1

This instruction is an example for Register Addressing mode.

This instruction transfers 32 bit data from R0 to R1. Where R0 is the Source Register and R1 is the Destination Register. The encoding of this instruction according to the above instruction format is as follows.

8 bits → opcode

4 bits → Source Register.

3 bits → Source Register's addressing mode.

4 bits → Destination Register.

3 bits → Destination Register's addressing mode.

10 bits → Index value or immediate operand.

Ex:2 MOVE 24(R0), R5

This instruction is an example for Index Addressing mode.

This instruction transfers 32 bit data from Memory to Register R5. The effective address of the operand is the value stored in the Register R0 + offset value.

The EA of an operand is given by

EA = [R0] + 24.

The encoding of this instruction according to the above instruction format is as follows.

8 bits → opcode

4 bits → Source Register.

3 bits → Source Register addressing mode.

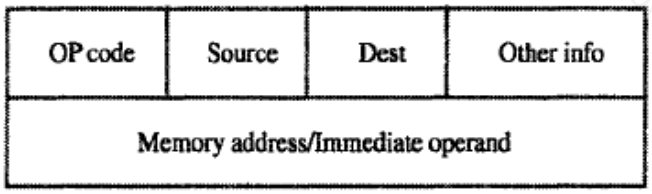
4 bits → Destination Register.

3 bits → Destination Register addressing mode.

10 bits → Index value or immediate operand.

b) Two word instruction format

The two instruction format is as shown in the fig.



Ex :1 MOVE R2, LOCA

This instruction is an example for Direct Addressing Mode.

This instruction transfers 32 bit data from Register R2 to Memory location whose symbolic name is given by LOCA.

The encoding of this instruction according to above instruction format is as follows

- This instruction format consists of 2 words.
- The 1st word is used to specify the opcode(8 bits), Source register(4 bits), Addressing Mode for Source(3 bits), Addressing Mode for Destination(3 bits) and index value or immediate operand as follows

8 bits → opcode

4 bits → Source Register.

3 bits → Source Register addressing mode.

3 bits → Destination Memory location addressing mode.

- The 2nd word is used to specify the 32 bit Memory address or 32 bit operand.

c) **Three word instruction format**

In instruction Move Loc1, Loc2 , both operands uses direct addressing mode

opcode	source	dest	Other info
Memory address / Immediate operand			
Memory address / Immediate operand			

This instruction transfers data from a memory location to another.

The encoding of this instruction according to above instruction format is as follows

- This instruction format consists of 3 words.
- The 1st word is used to specify the opcode(8 bits), Addressing Mode for Source(3 bits), Addressing Mode for Destination(3 bits) and index value or immediate operand as follows

  8 bits → opcode

  3 bits → Source Register addressing mode.

  3 bits → Destination Memory location addressing mode.

- The 2nd word is used to specify the 32 bit Memory address or 32 bit operand.
- The 3rd word is used to specify the 32 bit Memory address or 32 bit operand.

These types of instructions are used in CISC(Complex Instruction Set Computers).

The type of computer having only one word size instruction is called Reduced Instruction Set Computer(RISC).

The instruction format is as shown below -

OP code	R*i*	R*j*	R*k*	Other info

Ex: ADD R1, R2, R3

This instruction is an example for **Three operand Register Addressing Mode**.

This instruction adds the contents of Register R1 with the contents of R2 and result is placed in R3 Register.

The mathematical representation of this is as follows

[R3] ← [R1] + [R2]