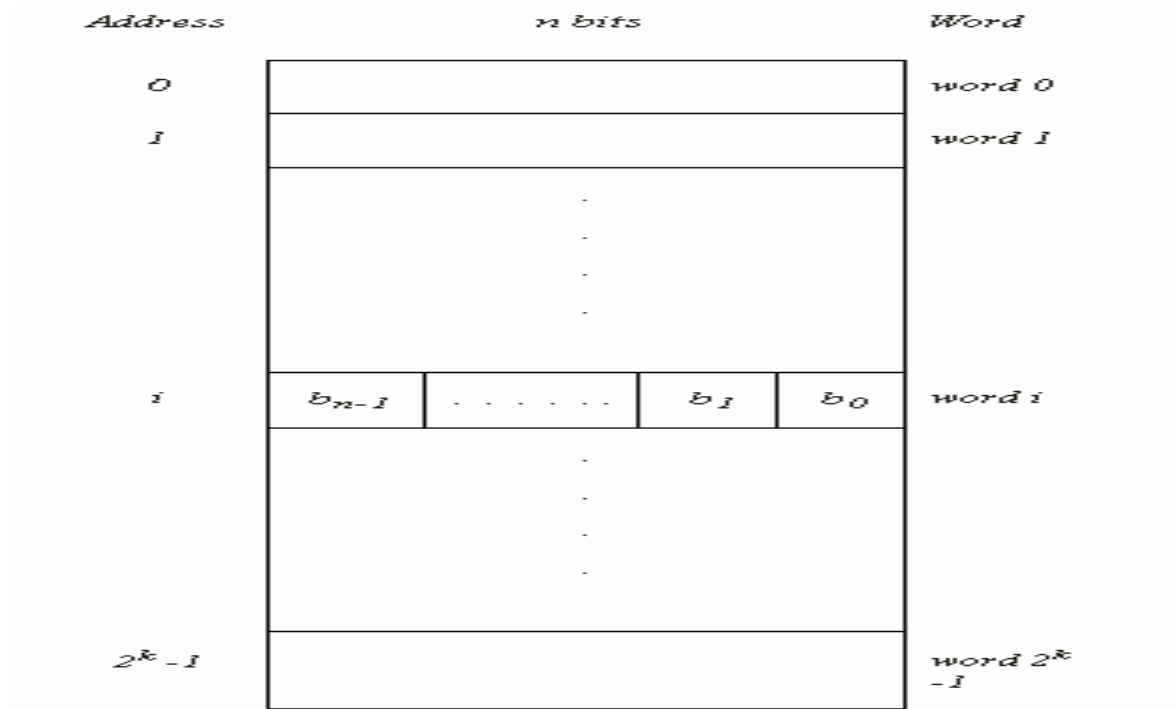


## MEMORY LOCATIONS, ADDRESSES & INFORMATION ENCODING



Computer stores data , instruction in the memory

- The data can be in the form of number or character
- Memory locations consist of millions of storage cells , Each cell is capable of storing 1 bit information having value 0 or 1.
- Single bit represents small amount of information
- For this reason memory is organized so that Group of n bits, called as a word of information can be stored and retrieved in a single basic operation.
- Here n is the word length – ranging from 16 to 64 bits
- Accessing the main memory requires address for each word location 0 to  $2^k - 1$ ;
- Main memory of this computer can have up to  $2^k$  words.

**Example:**

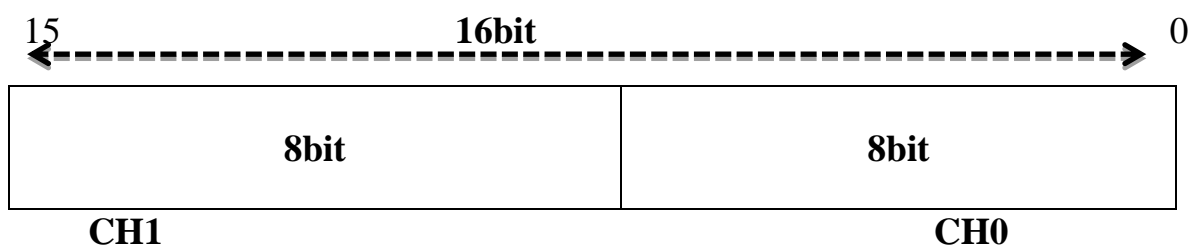
If processor has 16 address line it can address up to  $2^{16} = 65536$  memory location

If processor has 24 address line it can address up to  $2^{24} = 16777216$  16M memory location

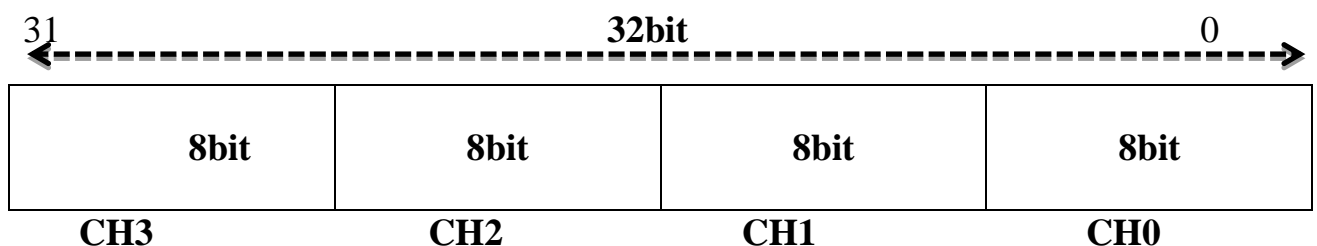
## CHARACTERS

- A Memory can also store character information such as text or string.
- These characters can be alphabets, symbols like #, @, &, \*, , " , ; , etc....
- Such characters are represented using 7 bit ASCII code.
- 8 bit can store 1 character
- 16 bit can store 2 characters
- 32 bit can store 4 characters.

### Character Format for 16bit

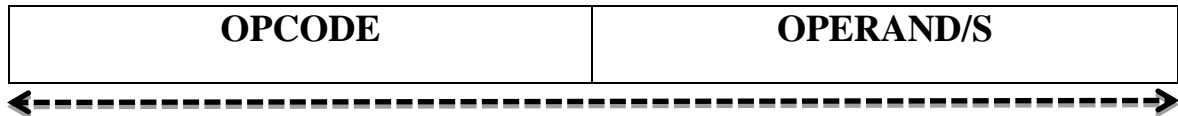


### Character Format for 32 bit



**INSTRUCTION:**

Program instruction are stored in memory . It usually consists of **OPCODE** and **OPERAND/S**.

**GENERAL FORMAT**

## Instruction

- 8 bit **OPCODE** specifies  $2^8=256$  different operations.
- **OPERAND** may be one of the register or it may be immediate value, or it may be a memory location.

**BYTE ADDRESSABILITY**

- The word length of 8 bit is known as 1 byte.
- The word length of 16 bit is known as half word.
- The word length range from 16 bit to 64 bit is known as word.
- Each successive byte are located at address 0, 1, 2, 3...
- Each Successive words are located at address 0, 4, 8.....

**BIG-ENDIAN & LITTLE-ENDIAN ASSIGNMENT****BIG-ENDIAN:**

<i>Word address</i>	<i>Byte address</i>			
0	0	1	2	3
4	4	5	6	7
$2^k-4$	$2^k-4$	$2^k-3$	$2^k-2$	$2^k-1$

**Big-endian order**

Lower byte address are used for most significant byte of the word.

**LITTLE ENDIAN:**

<i>Word address</i>	<i>Byte address</i>			
0	3	2	1	0
4	7	6	5	4
$2^k-4$	$2^k-1$	$2^k-2$	$2^k-3$	$2^k-4$

**Little-endian order**

Lower byte address is used for least significant byte of the word.

**MEMORY OPERATION**

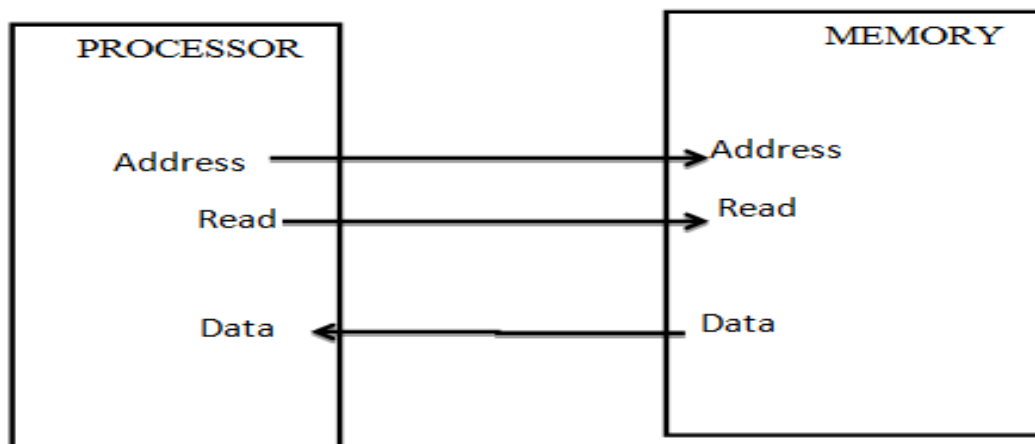
- We know that both program instruction and data operands are stored in memory.
- To execute the instruction processor reads the operand from memory
- After execution of instruction processor may store result in memory.

There are 2 types of operations

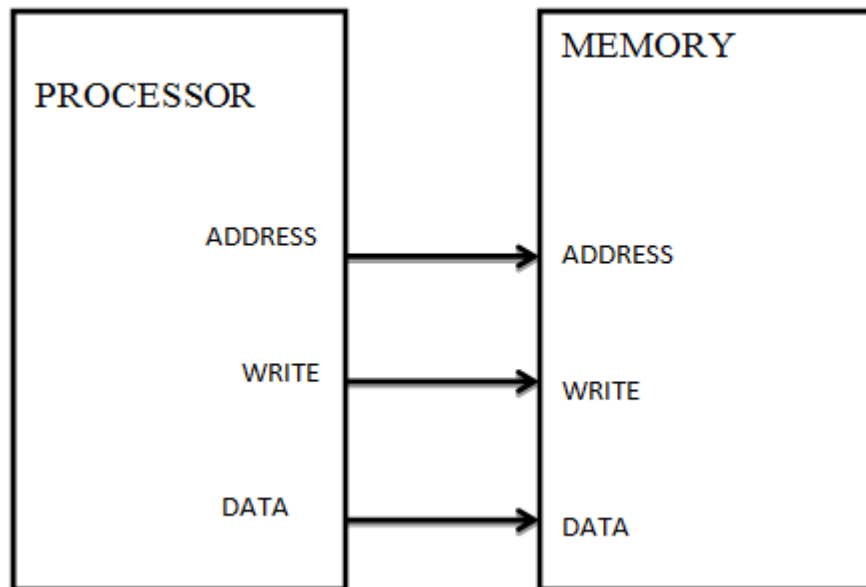
1) Load (read operation / fetch operation )

2) Store (write operation)

1) **LOAD OPERATION:**



- In load operation the content from specific memory location read by the processor.
- For load operation processor sends address of the memory location whose content is to be read and generate read signal to indicate read operation
- Memory identifies the address of memory location and sends the content to the processor.

**2) STORE OPERATION:**

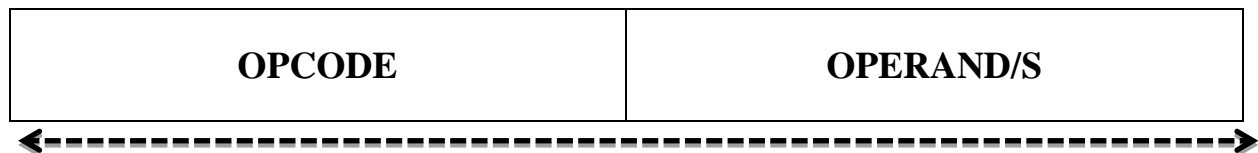
- For store operation processor sends the address of memory location where the data is to be written.
- It generates write signal to indicate write operation or store operation.
- Memory identifies the address of memory location and writes the data sent by the processor.

**INSTRUCTION AND INSTRUCTION SEQUENCING**

- The operation of computer is controlled by set of instruction.
- Instruction is a command to processor to perform a given task.
- Typically program consists of varieties of instruction such as add two number , compare two numbers or display character from keyboard

**Categories instruction:****1) Data movement instruction****Eg. MOV, PUSH, POP, XCHG****2) Arithmetic and logical instruction****Eg. ADD, SUB, MUL, DIV, OR, EX-OR****3) Data processing instruction****Eg. ADC-add with carry****SBC-subtract with carry****4) Data program control instruction**

- **JZ-jump if accumulator is zero.**
- **JC-jump if carry flag is 1.**

**INSTRUCTION FORMAT****INSTRUCTION**

Instruction usually consist of 2 parts

- 1) **OPCODE** (Mnemonics): Specifies operation to be performed.
- 2) **OPERAND**: The contents operated by OPCODE.

**REGISTER TRANSFER NOTATION**

- In computer system data transfer take place between processor & I/O system or between processor register & memory.
- Processor register represented by R0, R1, R2, ----Rn-1.
- Memory location represented by MER,LOC,M.....
- I/O Register are represented by DATAIN, DATAOUT.
- The content of register or memory location denoted by using placing around square bracket.

**Example: 1) MOV LOC, R2**

**This instruction moves the content of LOC moves to R2.**

**$R2 \leftarrow [LOC]$**

**2) MOV SUM, R0**

**$R0 \leftarrow [SUM]$**

**Moves the content of sum moves to R0.**

**3) ADD R1, R2, R3**

**$R3 \leftarrow [R1] + [R2]$**

**This instruction adds the content of register R1 and R3 and copies to R3**



**BASIC INSTRUCTION TYPES****1) THREE ADDRESS INSTRUCTION****2) TWO ADDRESS INSTRUCTION****3) ONE ADDRESS INSTRUCTION****4) ZERO ADDRESS INSTRUCTION****1) THREE ADDRESS INSTRUCTION:**

Three address instructions can be represented by symbolically.

ADD A,B,C
-----------

- This instruction adds the content of A&B & copies the computed result to OPERAND C

NOTATION:  $C \leftarrow [A] + [B]$

- In this instruction operand A&B are source operand & operand C is destination.

GF

OPCODE	SOURCE1	SOURCE2	DESTINATION
--------	---------	---------	-------------

**2) TWO ADDRESS INSTRUCTION:**

Two address instructions can be represented by symbolically.

ADD A,B
---------

- This instruction adds the content of A& B & copies the computed result to operand B & overwrites the previous content.

**NOTATION:**  $B \leftarrow [A] + [B]$

In this instruction operand A & B is both source B destination.

GF

OPCODE	SOURCE1	SOURCE2/DESTINATION
--------	---------	---------------------

### 3) ONE ADDRESS INSTRUCTION

One address instructions can be represented by symbolically.

ADD	B
-----	---

This instruction adds the content of B to the processor register called accumulator (R0) & copies the computed result to R0 and overwrites the previous content.

NOTATION:  $R0 \leftarrow [B] + [R0]$

GF

OPCODE	SOURCE
--------	--------

EXAMPLE: LOAD A

This instruction copies the content of memory location A to the accumulator register (R0)

NOTATION:  $R0 \leftarrow [A]$

Example : STORE B

This instruction copies the content of Accumulator register (R0) to the memory location B

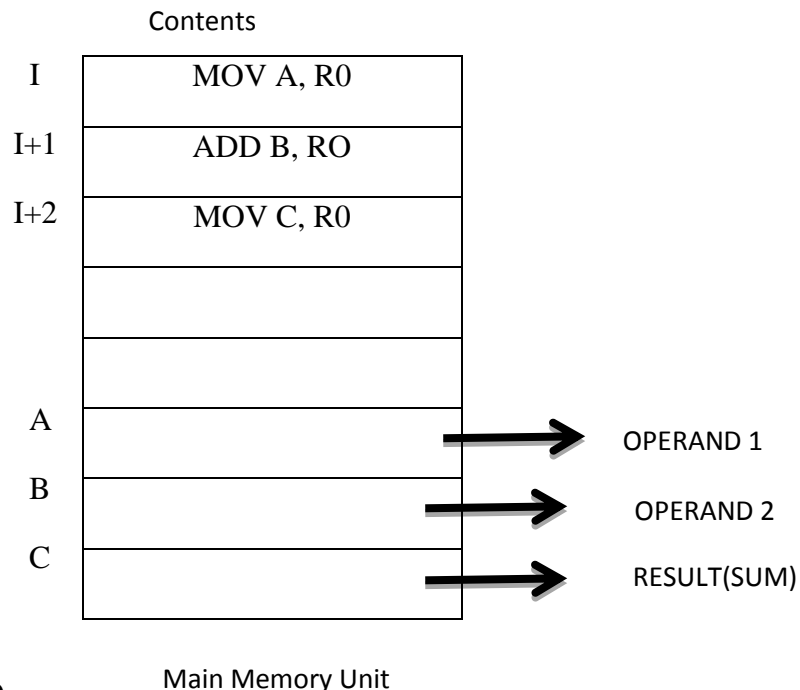
NOTATION:  $B \leftarrow [R0]$

4) ZERO ADDRESS INSTRUCTION: in zero address instruction operands are define implicitly.

Example: PUSH A [1 operand instruction]

PUSH B[1 OPERAND instruction]

ADD [Zero operand instruction].

**INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCE**

- 1) MOV A, R0
- 2) ADD B, R0
- 3) MOV C, R0

- The above three instruction computes  $C \leftarrow [A] + [B]$
- These are stored in continues memory location such as I, I+1, I+2...
- Operand & sum appears at a different memory location.
- Processor executes the program with the help of PC(program counter)
- PC holds address of next instruction to be executed,
- To begins the execution, program counter points to the address of first instruction.
- Then CPU control circuit uses the information of PC to fetch & executes the instruction.
- Instruction executed in the order of increasing address in known as straight line sequence.

**INSTRUCTION EXECUTION CYCLE/PHASE**

**Processor carried out 3 cycle to execute instruction**

**1) INSTRUCTION FETCH CYCLE:**

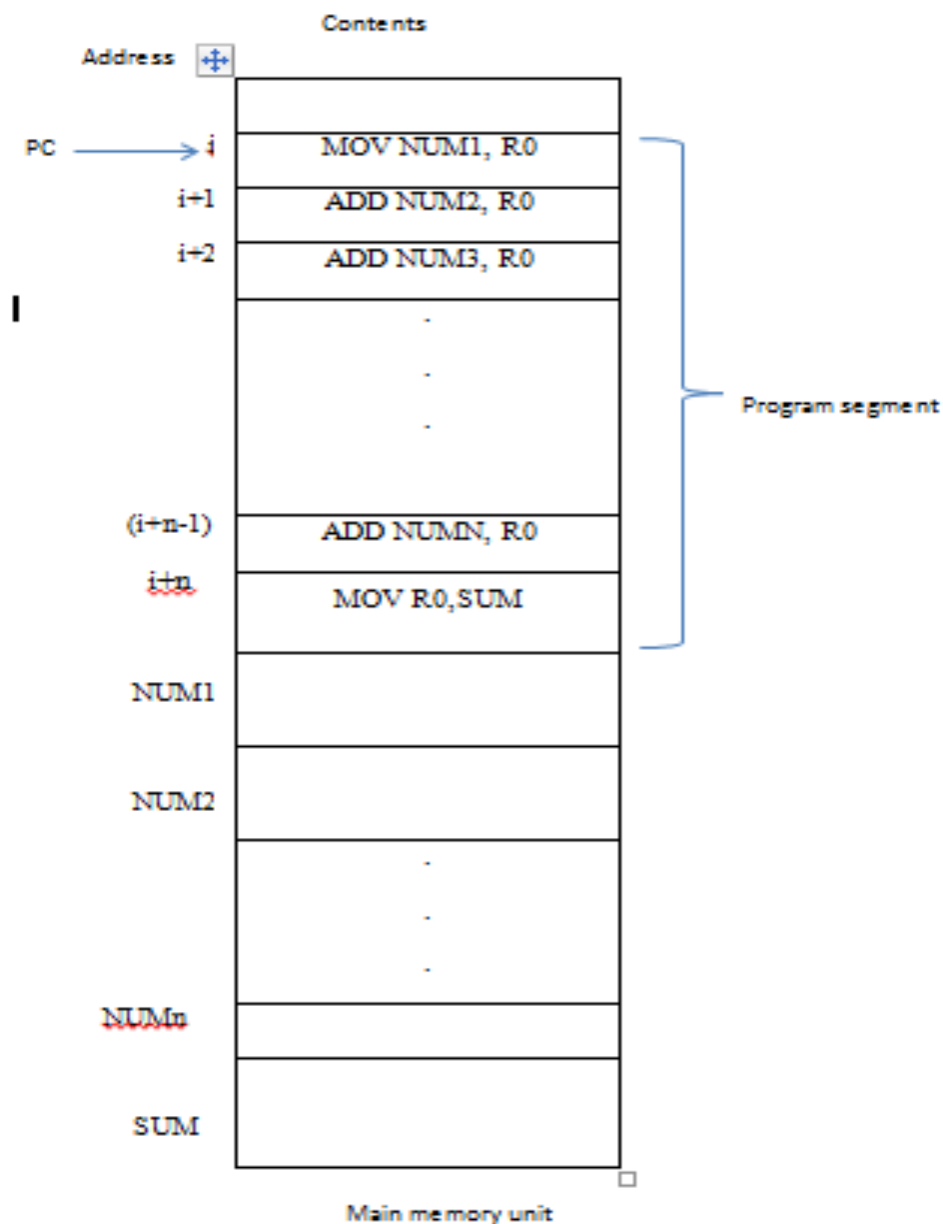
- In this cycle instruction is fetch from memory location who's address is in PC.
- This instruction is placed in the IR of the processor.

**2)INSTRUCTIONS DECODE CYCLE:**

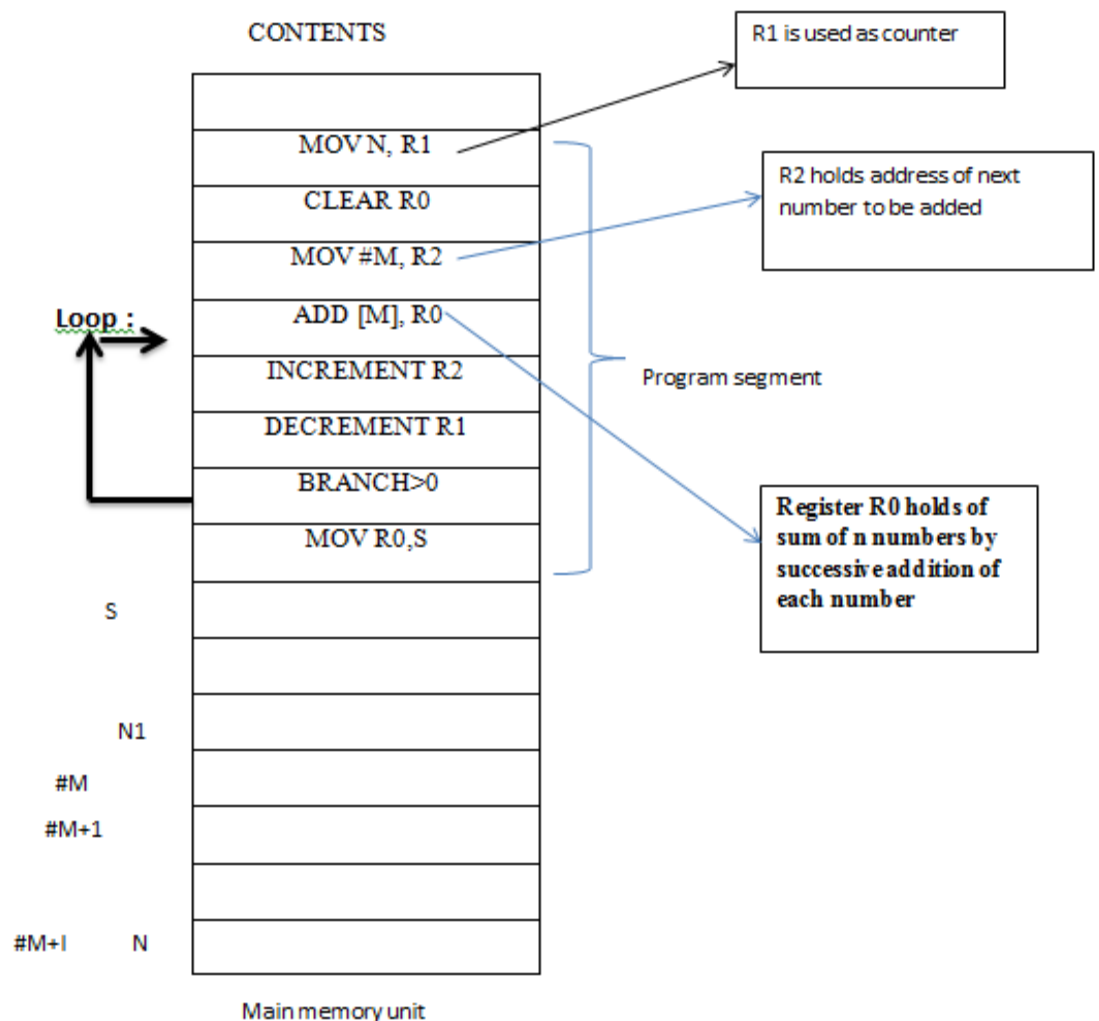
- OPCODE of the instruction is stored in IR & is decoded to determine which operation is to be performed

**3)INSTRUCTION EXECUTION CYCLE:**

- In this cycle, specified operation is performed by the processor.
- The arithmetic operation like Addition ,subtraction, multiplication ,division & logical operation like OR, AND, NOT, EX-NOR .....etc.

**BRANCHING**

- Suppose , you would like to add 'n' numbers stored in continues memory location such as num1, num2, num3,... num n
- A separate add instruction is used to add the content of R0.
- After all number has been added result is places in memory location sum.



- We can avoid repeated use of ADD instruction for adding ‘n’ numbers in memory by using conditional branch instruction.
- This reduces program length.
- Very less amount of memory is required to store the program.
- The Register R0 holds of sum of n numbers by successive addition of each number to it,
- Register R1 is used as a counter to determine number of time loop is executed.

- Register R2 holds address of next number to be added
- M represents content of memory location
- The program execution continues in straight line sequence until encounter branch >0
- If condition is true PC points to ADD [M],R0 & execute the instruction and also decrement R1 because R1 holds total number of count
- R2 is incremented to point to next number ,This process is repeated until R1 s zero (0)

### **CONDITION CODES [FLAG REGISTER/FLIP FLOP]**

- It shows status of the result when you perform arithmetic and logical operation

Four commonly used flags are

N(negative) → Set to 1 if the result is negative; otherwise, cleared to 0

Z(zero) → Set to 1 if the result is 0; otherwise, cleared to 0

V(overflow)→ Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0 (over flow occur when the result of arithmetic operation is outside the range)

C(carry) → Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

## **Addressing modes**

**Definition :** The way in which operand of the instruction is specified

### **Types of addressing mode .**

- **Register addressing mode**
- **Absolute addressing mode**
- **Immediate addressing mode**
- **Indirect addressing mode**
- **Index addressing mode**
- **Relative addressing**
- **Auto increment mode**
- **Auto decrement mode**

### **Register addressing mode:**

- **Operand is content of CPU Register , Name of the register given in instruction.**

**Example : MOV R1,R2**

**This instruction copies the content of register R1 to R2.**

**Example : ADD R1,R0**

**This instruction adds the content of register R1 and R0**

### **Absolute addressing mode**

- **Address is Explicitly specified in the instruction.**

**Example : MOV 2000,A**

**This instruction copies the content of memory location 2000 to A**

**Example : MOV LOC,R2**

**Copies the content of memory location LOC into Register R2**



**Example : ADD 3000,B**

**This instruction copies the content of memory location 3000 to B**

**Immediate addressing mode**

- **Operand is given explicitly specified in the instruction.**

**Example: MOV #2000,R0**

**This instruction copies immediate value 2000 to the register R0.**

**A common convention is use , the pound sign (#) in front of value of the operand to indicate that this value is to be used as immediate operand**

**Indirect addressing mode**

- **The effective address of the operand is content of register or main memory location , whose address is given explicitly in the instruction**

**Example : MOV (R0),A**

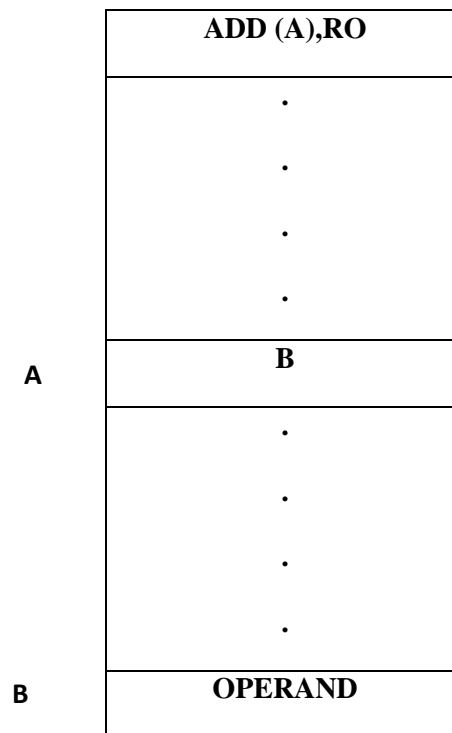
**This instruction copies the content of memory location pointed by register R0 to A**

**Example : MOV B,(LOC)**

**This instruction copies the content of B to the memory location pointed by (LOC).**

**Example: ADD (R1),R2**

**This instruction adds the content of memory location pointed by register R1 to R2**

**Example: indirect addressing through memory location**

- This instruction fetches the operand from the address, pointed by the A and adds them to R0

**Example :** Add (R1), R0 (this mode is often called as register indirect mode)

This instruction fetches the operand from the address, pointed by the contents of the register R1 and adds them to R0

**Example :** Add (B), R0

This instruction fetches the operand from memory location pointed by 'B' and adds them to R0.

**Index addressing mode**

- In this scheme effective address is generated by adding constant value to the specified register.
- This constant value is called displacement or offset.
- Index register symbolically represented as  $X(R)$
- Where X is displacement or constant and R is any register

**Example : MOV 20(R1),R0**

**Where R1=2000**

**EA=X+R (i.e EA → effective address)**

**20+2000=2020**

**Relative addressing**

**Effective address is determined by using PC in place of GPRS.**

→ **Back: ADD (M),R0**  
.....  
.....  
.....  
→ **JNZ Back**

**Auto increment mode**

- The 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 the next value.
- This increment is 1 for byte sized operands, 2 for 16 bit operands and so on.

**E.g. Add (R2) +, R0**

**Here are the contents of R2 are first used as an E.A. then they are incremented.**

**Auto decrement mode:**

- The 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 and then the value is accessed.

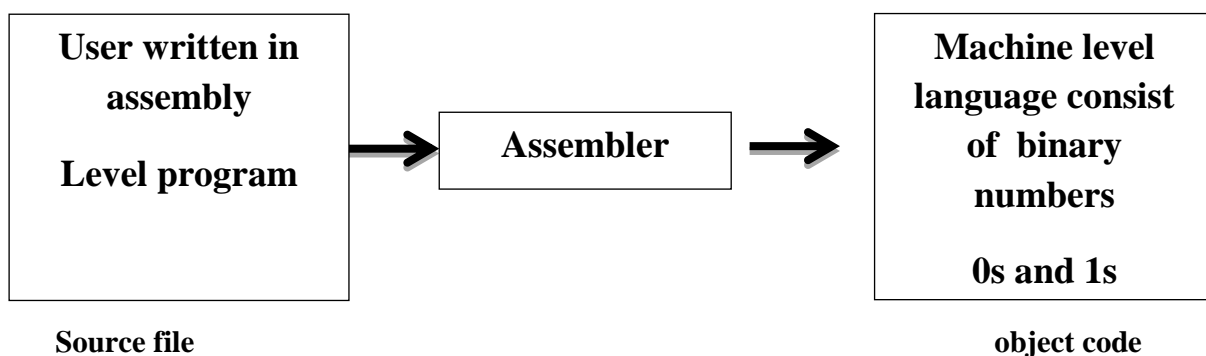
E.g. Add - (R2), R0

Here the contents of R2 are first decremented and then used as an E.A. for the operand which is added to the contents of R0.

The auto increment addressing mode and the auto decrement addressing mode are widely used for the implementation of data structures like Stack

**Assembly language**

**Define assembler :** It translates assembly level language into equivalent machine language program.



## Formats of Assembly level language

### Syntax

**Label : Mnemonics operand1, operand2; comment**

- Here each instruction separated by space.
- Every new instruction start from new line.
- If label is not used then : should not used .
- Mnemonics represents operation to be performed .
- If there are 2 or more operand they should be separated by comma(,)
- Semicolon(;) indicates comment

## Assembler directives

**Definition :** Assembly level language program composed of 2 statement

- Assembler
- Directive

**Assembler:** It translate assembly level language into equivalent machine language program.

**Directive :** That directs the assembler during assembly process for which no machine code is generated.

**Types of Assembler directives****➤ Data control directive**

DB (define byte)	→	(1Byte)
DW (define word)	→	(2 Byte)
DD (define double word)	→	(4Byte)
DQ (define quad)	→	(8 Byte)
DT (define ten byte)	→	(10 Byte)

**Example :**

**Total        DB    0**

**Above statement reserve 1 byte of memory for variable total and initialize value is 0**

**Total        DB        ?**

**Above statement reserve 1 byte of memory for variable total and assign value is unknown**

**Total DB 10H,20H,30H,40H**

**Above statement reserve 4 byte of memory for variable total and initialize value is 10,20,30,40**

**MES DB “welcome”**

**Above statement reserve 7 byte of memory for the variable MES and initialize the value is welcome**

➤ **DUP(Duplication directive )**

This directive can be used to initialize several locations and assign values to these locations.

GF:

**Variable\_Name data\_type num dup(value)**

**Total DB 2 dup(00)**

It reserves 2 bytes for the variable total and initializes by value 0

**BLIST DB 3,4,5, 5dup(99),88,88**

It reserves 10 bytes for variable BLIST and initializes by values 3,4,5, 99,99,99,99,99,88,88

➤ **EQU (equate )**

It is used to declare symbols, to which some constant value is assigned. Such a symbol is called a macro symbol.

GF:

**Symbol\_name equ expression**

**Example :**

**NUM EQU 100**

It declares the symbol NUM with value 100

Assembler replaces 100 when symbol NUM appears

➤ **ORIGIN**

- This assembler directive tells the assembler that where to place the block of data in memory or where to start loading of object program in memory.
- It specifies starting memory location for data or object code  
**ORIGIN 100**

This directive loads the object program or data in memory starting from location 100

➤ **PROC (procedure)**

This directive is used to define the procedure , procedure name must be present and must be unique

There are 2 types of procedure

- **NEAR** :it can be called within segment.
- **FAR**: it can be called from any segment

**Procedure\_name PROC[NEAR/FAR]**

**Example :**

**IFACT PROC FAR**

It define IFACT as FAR type it cannot be called from any segment.

**HEX2ASCCI PROC NEAR**

It define HEX2ASCCI as NEAR type it cannot be called within segment.

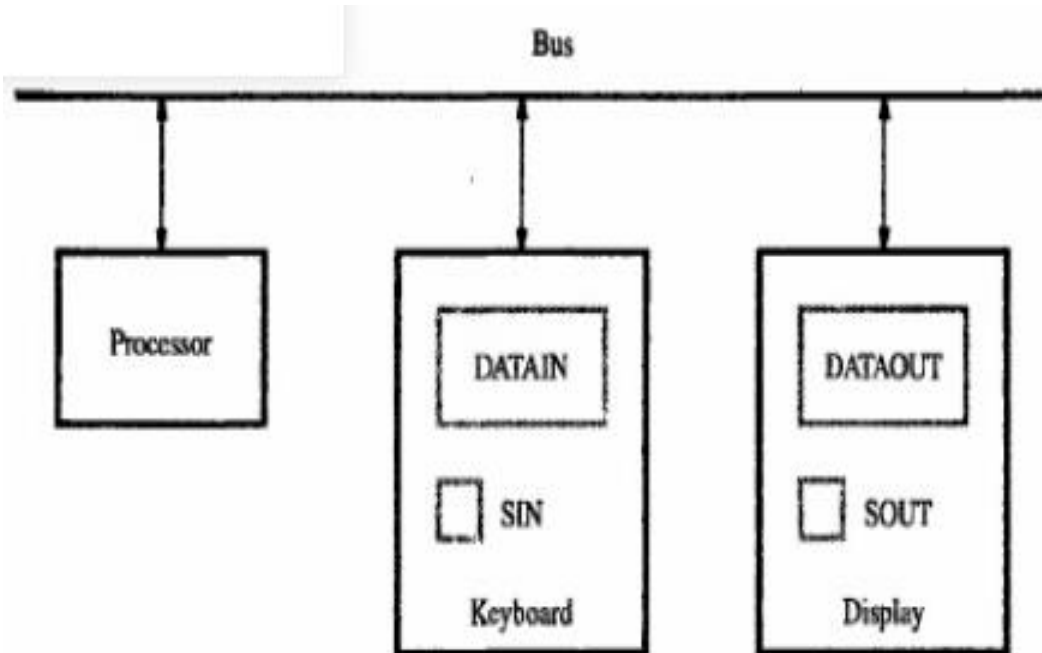
**HEX2ASCCI PROC**

.....

**HEX2ASCCI ENDP**



## Simple basic input output operation



- User can give the information to the processor using keyboard.
- User can see the result or output with the help of display unit.
- The transfer of data between keyboard ,processor, display unit is called input output data transfer or I/O data transfer

Fig shows typical bus connection for keyboard ,processor and display device .

- **DATAIN** and **DATAOUT** are the register by which processor reads the content from keyboard and send the data for the display .
- Data transfer rate from keyboard to CPU typically 5 character /sec.
- Data transfer rate from CPU to display unit 50,000 character /sec.
- Both of the rate much slower than speed of the processor that execute million of instruction per sec.

- When key is pressed corresponding character code stored in DATAIN register and SIN status bit is set to 1, to indicate value character code is available in DATAIN register .
- Processor check the SIN bit when it is found SIN=1, It reads the content of DATAIN register after completion of read operation SIN is automatically cleared to 0.
- When character is transferred from processor to DATAOUT register SOUT status bit are used.
- Processor check SOUT bit , if SOUT is 1 processor transfer the data out register
- The display device reads the content from data out register .