Chapter-2 (Unit-2)

Machine Instructions and Programs :-

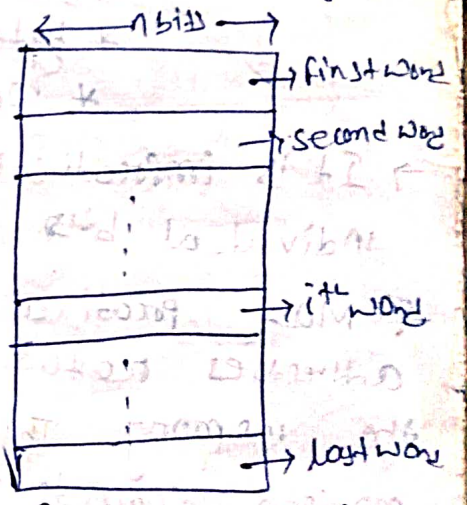
Memory locations and Addresses :

→ Memory consists of many millions of storage cells, each of which can store a bit of information having the value of 0 or 1.

→ Because a single bit represents a very small amount of information, bits are seldom handled individually. (often)/rarely

→ The useful approach is to deal with them in groups of fixed size. for this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation.

→ Each group of n bits is reffered to as a word of information, and n is called the word length.



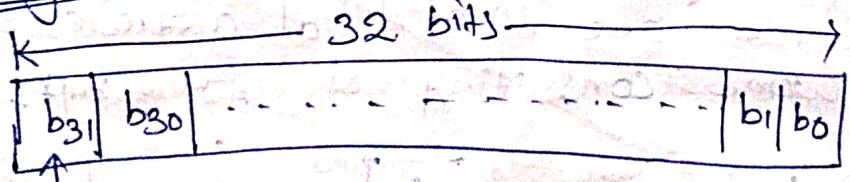
(Memory words)

→ Modern computers have word length range from 16 to 64 bits

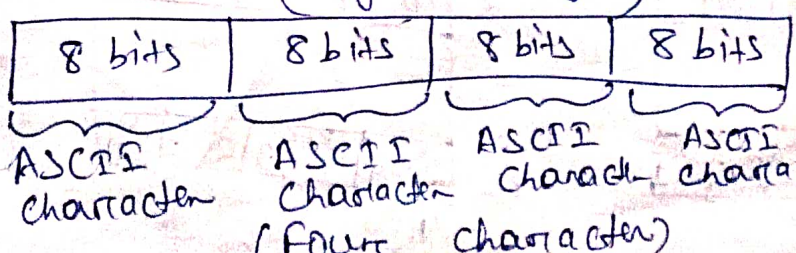
→ If a word length of computer is 32 bits,

* a single word can store a 32-bit 2's complement number or
* 4 ASCII characters, each occupying 8 bits.

→ Each unit of 8 bits are called a byte.



sign bit : $b_{31} = 0$ for positive no.
$b_{31} = 1$ for negative no.
(signed Integer)



(Four character)

→ A K-bit address creates the address space of this computer is 2^K, and the ~~address~~ numbers from 0 to $2^K - 1$.

→ A 32 bit address creates an address space of $2^{32} = 4G$ locations.

→ A 16 bit addresses creates an address space of $2^{16} = 64K$ locations.

→ A 24 bit → $2^{24} = 16M$ locations

Byte Addressability :-

→ 3 basic information quantities →

* bit
* byte → 8 bits
* word → ranges from 16 to 64 bits.

→ It is impractical to assign distinct addresses to individual bits locations in the memory.

→ The Most practical assignment is to have successive addresses refers to successive byte locations in the memory. This is the assignment used in most modern computers. and termed as byte addressable memory.

→ If the word length of the m/c is 32 bits, successive words are located at addresses 0, 4, 8, ... with each word consisting of four bytes.

0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
	⋮			
$2^{k+2}-4$	$2^{k+2}-4$	$2^{k+2}-3$	$2^{k+2}-2$	$2^{k+2}-1$

0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
⋮				
2^k-1	$2^{k+2}-4$	$2^{k+2}-3$	$2^{k+2}-2$	$2^{k+2}-1$

$4(2^k-1)$

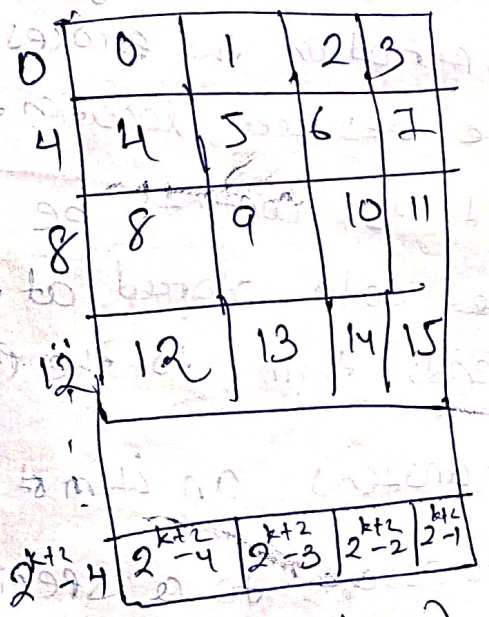
$2^{k+2}-4$

Big-Endian and Little Endian

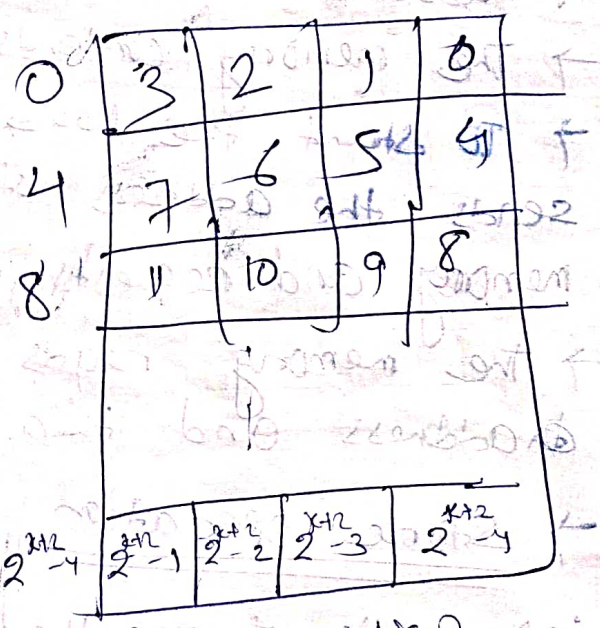
→ There are 2 ways that byte addresses can be assigned across words.

* big-endian: When lower byte addresses are used for the more significant byte (leftmost bytes) of the word.

* Little-endian: When lower byte addresses are used for the less significant byte (rightmost bytes) of the word.



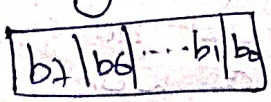
(Big endian)



(Little endian)

→ In both cases, byte addresses 0, 4, 8, ... are taken as the addresses of successive words in the memory and are the addresses used when specifying memory read and write operation for words.

→ The ordering of bits within byte is left to right



Numericals on Big Endian and Little endian

① How "HELLO_WORLD" will store in machine if m/c uses

(a) Big Endian
- (i) word length = 16 bits
- (ii) word length = 32 bits
- (iii) word length = 64 bits

(b) Little Endian
- (i) W.L = 16 bits
- (ii) W.L = 32 bits
- (iii) W.L = 64 bits

Starting address = 1000.

Ans:-

(a) Big Endian format

(i) word length = 16 bits = 2 Byte

1000	H	E
1002	L	L
1004	O	_
1006	W	O
1008	R	L
1010	D	

(ii) word length = 32 bits = 4 Bytes

1000	H	E	L	L
1004	O	_	W	O
1008	R	L	D	

(b) Little Endian format

(i) word length = 16 bits = 2 Byte

1000	E	H
1002	L	L
1004	_	O
1006	O	W
1008	L	R
1010		D

(ii) word length = 32 bits = 4 Bytes

1000	L	L	E	H
1004	O	W	_	O
1006		D	L	R

L	L	E	H
O	W	_	O
	D	L	R

(vii) word length = 64 bits
= 8 Bytes

1000	H	E	L	L	O	_	W	O
1008	R	L	D					

(iii) word length
= 64 bits = 8 Bytes

O	W	_	O	L	L	E	H	1000
					D	L	R	1008

② Write a message "COVID_PANDEMIC" with the same previous question information

Memory ~~locations~~ operations

→ Both program ~~and~~ instructions and data operands are stored in memory.

→ Two basic operations involving the memory are needed,

* Load (Read / Fetch)
* Store (Write)

→ Load operation transfers a copy of the contents of a specific memory location to the processor.

→ The memory contents remain unchanged.

→ To start the load operation, the processor sends the address of the desired location to the memory and requests that its contents be read.

→ The memory reads the data stored at that address and then sends them to the processor.

→ Store operation transfers an item of information from the processor to a specific memory location,

→ destroy the former contents of that location

→ Processor sends the address of the desired location to the memory, together with the data to be written into that location.

→ An information item, of either one word or one byte can be transferred betn the processor and the memory in a single operations.

→ Processors contains a small no. of registers, each capable of holding a word. These registers are either source or destination of a transfer to or from the memory.

Instructions and Instruction sequencing

→ A computer must have instructions capable of performing four types of operations:

* Data transfers between memory and processor registers.
* Arithmetic and logic operations on data.
* Program sequencing and control.
* I/O ~~control~~ transfers.

Register transfer Notation

→ The possible locations that may be involved in ~~exec~~ informal transfers are memory location, processor registers or registers in the I/O subsystem.

→ Example :- for memory location → LOC, Place, A, Varz
processor register names → $R_0, R_1, \ldots R_5$
I/O register names → DataIn, OUTSTATUS, ...

→ The contents of a memory location is denoted by placing square brackets around the name of the location.

Ex :- $R_1 \leftarrow [LOC]$

$R_3 \leftarrow [R_1] + [R_2]$

This type of notation is known as register transfer notation (RTN)

→ Right side of RTN as value.

→ Left side of RTN → where the ~~data~~ value is to be placed overwriting the old contents of that location.

Assembly Language Notation :-

→ Assembly language is a type of notation to represent machine instructions and programs.

Ex :- MOVE LOC, R_1 : $R_1 \leftarrow [LOC]$

Add R_1, R_2, R_3 : $R_3 \leftarrow [R_1] + [R_2]$

Basic Instruction types :-

→ 4 types of instructions

* 3 - address instruction
* 2 - " "
* 1 - " "
* 0 - " "

OPCode : Operation performed by the instruction

Original program is $C = A + B$

3 - address instruction :- | OPCode | source1 | source2 | Dest |

operation source1, source2, Destination.

Ex :- | Add A, B, C | : $C \leftarrow [A] + [B]$

Operands A and B are called the source operands

C is called as destination operand.

→ If k bits are needed to specify the memory address of each operand then the encoded form of the above instruction must contain 3k bits for addressing purposes in addition to the bits needed to denote the Add operation.

→ In 32 bit address space, a 3 - address instruction is too large to fit in one word for a reasonable word length.

2 - address instruction :- | OPCode | Source | Source/Destn |

operation source, Destination

Ex :- Add A, B : $B \leftarrow [A] + [B]$

~~This is not the soln for~~ This instn can't be used to solve for the original ~~program~~ / problem.

C = A + B	C ← [A] + [B]
Move B, C	C ← [B]
Add A, C	C ← [A] + [C]

→ Even 2-address instruction will not normally fit into one word for usual word lengths and address size.

1-address instruction :-

opcode	source/Destination

→ Accumulator is used ~~for this~~ in this type instruction and this register is implied.

Add A : add the content of A to Accumulator.

C = A + B	
Load A	ACC ← [A]
Add B	ACC ← [ACC] + [B]
Store C	C ← [ACC]

→ Operand may be source/destination depending upon the type of operation.

→ No. of general purpose registers are 8 to 32.

→ Access to data in these registers is much more faster than to data stored in memory locations because the registers are inside the processor.

→ The no. of registers are relatively small, only a few bits are needed to specify the register.

Ex:- 32 registers → only 5 bits are needed.

→ This is much less than the no. of bits needed to give the address of a location in the memory.

→ Because of faster processing and less no. of bits are required to represent the register, registers are used to store temporary data in the processor during processing.

Load A, R_i : $R_i \leftarrow [A]$

Store R_i, A : $A \leftarrow [R_i]$

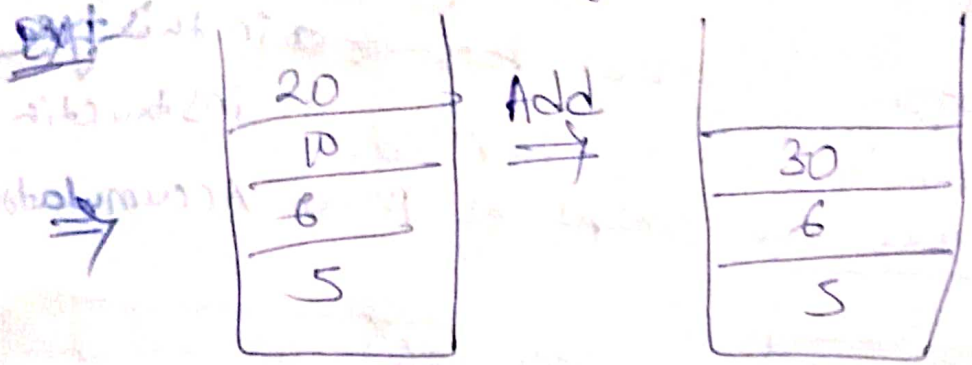
R_i is the general purpose register

Add A, R_i : $R_i \leftarrow [A] + [R_i]$

Zero-Address instruction :-

→ It has no address field. Here stack is used implicitly.

~~Ex:~~



Convert the following expression $X = (A+B) * (C+D)$ into three, two, one and zero

3-address instn :-

Add A, B, R_0
Add C, D, R_1
MUL R_0, R_1, X

2-address instn

MOV A, R_0
~~MOV B~~
Add B, R_0
MOV C, R_1
Add D, R_1
MUL R_0, R_1
MOV R_1, X

One-address

Load A
Add B
Store R_0
Load C
Add D
MUL R_0
Store X

Zero-address

PUSH A
PUSH B
ADD
PUSH C
PUSH D
ADD
MUL
POP X

① $X = (A - B + C * (D * E - F)) / (G + H * K)$

(i) Using 3-address instrn

Mult D, E, R_0 : $R_0 \leftarrow [D] * [E]$

Sub R_0, F, R_1 : $R_1 \leftarrow [R_0] - [F]$

Mult C, R_1, R_2 : $R_2 \leftarrow [C] * [R_1]$

Sub A, B, R_3 : $R_3 \leftarrow [A] - [B]$

Add R_3, R_2, R_4 : $R_4 \leftarrow R_3 + R_2$

Mult H, K, R_5 : $R_5 \leftarrow [H] * [K]$

Add G, R_5, R_6 : $R_6 \leftarrow [G] + [R_6]$

Div R_4, R_6, X : $X \leftarrow R_4 / R_6$

(ii) using 2-address instruction :-

~~MOV D, R_0~~
~~Mult R_0~~

MOV D, R_0 : $R_0 \leftarrow [D]$

Mult E, R_0 : $R_0 \leftarrow R_0 * [E]$

Sub F, R_0 : $R_0 \leftarrow R_0 - [F]$

Mult C, R_0 : $R_0 \leftarrow R_0 * [C]$

MOV A, R_1 : $R_1 \leftarrow [A]$

Sub B, R_1 : $R_1 \leftarrow R_1 - [B]$

Add R_1, R_0 : $R_0 \leftarrow R_1 + R_0$

MOV H, R_2 : $R_2 \leftarrow [H]$

Mult K, R_2 : $R_2 \leftarrow R_2 * [K]$

Add G, R_2 : $R_2 \leftarrow R_2 + [G]$

~~Div R_2, R~~

Div R_2, R_0 : $R_0 \leftarrow R_0 / R_2$

MOV R_0, X : $X \leftarrow R_0$

(iii) Using 1-address instruction :-

Load D	: ACC ← [D]
Mult E	: ACC ← [ACC] * [E]
Sub F	: ACC ← [ACC] − [F]
Mult C	: ACC ← [ACC] * [C]
Store R_0	: R_0 ← [ACC]
Load A	: ACC ← [A]
Sub B	: ACC ← ACC − [B]
Add R_0	: ACC ← ACC + R_0
Store R_0	: R_0 ← ACC
Load H	: ACC ← [H]
Mult K	: ACC ← [ACC] * [K]
Add G	: ACC ← [ACC] + [G]
Store R_1	: R_1 ← [ACC]
Load R_0	:- ACC ← R_0
Div R_1	: ACC ← ACC / R_1
Store X	: X ← ACC

(iv) Using 0-address instruction

$X = (A - B + C * (D * E - F)) / (G + H * K)$

$= AB - CDE*F-*+GHK*+/$

Push A	Push G	
Push B	Push H	
Sub	Push K	
Push C	Mult	
Push D	Add	
Push E	Div	
Mult	POP X	
Push F		
Sub		
Mult		
Add		

RISC Instruction :-

→ In RISC, any operation can be done by taking the operand register only.

→ The data can be access from memory by load and store

Ex :- $X = (A+B) * (C+D)$

```
Load  A, R1
Load  B, R2
Load  C, R3
Load  D, R4
Add   R1, R2, R5
Add   R3, R4, R6
Mul   R5, R6, R7
Store R7, X
```

Q.2) $Z = (A - B + C) * (D / E * F) / G$

three-address instruction :-

Sub A, B, R_0
Add R_0, C, R_1
Div D, E, R_0
Mult R_0, F, R_2
Mult R_1, R_2, R_0
Div R_0, G, Z

two-address instruction :-

MOV A, R_0
Sub B, R_0
Add C, R_0
MOV D, R_1
Div E, R_1
Mult F, R_1
Mult R_0, R_1
Div G, R_1
MOV R_1, Z

One-address instruction :-

Load A
Sub B
Add C
Store R_0
Load D
Div E
Mult F
Mult R_0
Div G
Store Z

zero-address instruction

AB - C + DE/F * * G/

Push A
Push B
Sub
Push C
Add
Push D
Push E
Div
Push F
Mult
Mult
Push G
Div
POP Z

memory reference :-

→ To fetch one instruction from memory for execution requires one clock cycle.

→ If any instruction is having one memory operand then total no. of ~~memory reference~~ clock cycles is 2.

→ If the result need to be store in memory for the above case then total no. of memory reference will be 3 clock cycle

Ex :-

		Total clock cycle
Add R_0, R_1	:	1 (1 for instn fetch)
Sub A, R_1	:	2 (1 for instn fetch + 1 for operand fetch)
MUL R_1, A	:	3 (1 + 1 + 1 for result store)
INC R_1	:	1 (1 for instn fetch)
INC A	:	3 (1 + 1 for operand fetch + 1 for result store)

Problems

Consider a hypothetical CPU which support one-address and two-address instructions. The 16-bit instruction is stored in 128 word memory. If there exist 2 two-address instruction, what will be the one-address instructions supported by the CPU.

Ans :- memory size = 128 word = 2^7

→ 7 bit is required to address a memory

OP code	Add$_1$
9	7

16 bit

OP code	Add$_1$	Add$_2$
2	7	7

16

$2^9 = 512$ one-address instn can possible

$2^2 = 4$ two-address instn can possible

→ If there exist 2 two-address instn then

1 - two-address instn can accumulate $= 2^7$ one-address instr

2 - two address instn can accumulate

$2 * 2^7 = 256$ one-address instn

∴ 256 one-address instn presents in CPU.

$$P + Q * 2^7 = 2^9$$

(Total possible one-address instn)

P → (valid one-address instn)

$Q * 2^7$ → (Invalid one address instn due to Q no. of two-address instn)

$P \to P_{min}$ if $Q \to Q_{max}$ $\quad 1 \leq Q \leq 3$

$P \to P_{max}$ if $Q \to Q_{min}$

if $Q = 1$, $P = 384$ $\quad$ if $Q = 2$, $P = 256$, if $Q = 3$, $P = 128$

$$P + Q * 2^7 + R * 2^{14} = 2^{16}$$

All the bits are operands.

P → valid zero-address instruction

Q → Invalid zero address instn due to Q no. of one-address instruction.

R → Invalid one-address instn due to R no. of two-address instn.

if $R = 2Q = 250$, 6 remaining one address can accumulate $6 * 2^7$

$P + 250 * 2^7 + 2 * 2^{14} = 2^{16}$

$\Rightarrow P = 2^{16} - 2^{15} - 250 * 2^7$

$= 2^7(2^9 - 2^8 - 250) = 2^7(512 - 256 - 250)$

$= 2^7(6) = 6 * 2^7$

② A computer has 32 bit instrn and 12 bit address. If there are 250, two-address instruction then how many one-address instruction can be formulated.

Ans :-

opcode	Add_1
20	12

32 bit

opcode	Add_1	Add_2
8	12	12

32 bit

$2^{20} = 4K$ no. of one-address

$2^8 = 256$ no. of two address.

6 no. of two address can accumulate $= 6 * 2^{12}$ no. of one-address instrn

$$P + Q * 2^{12} = 2^{20}$$

$$P + 250 * 2^{12} = 2^{20}$$

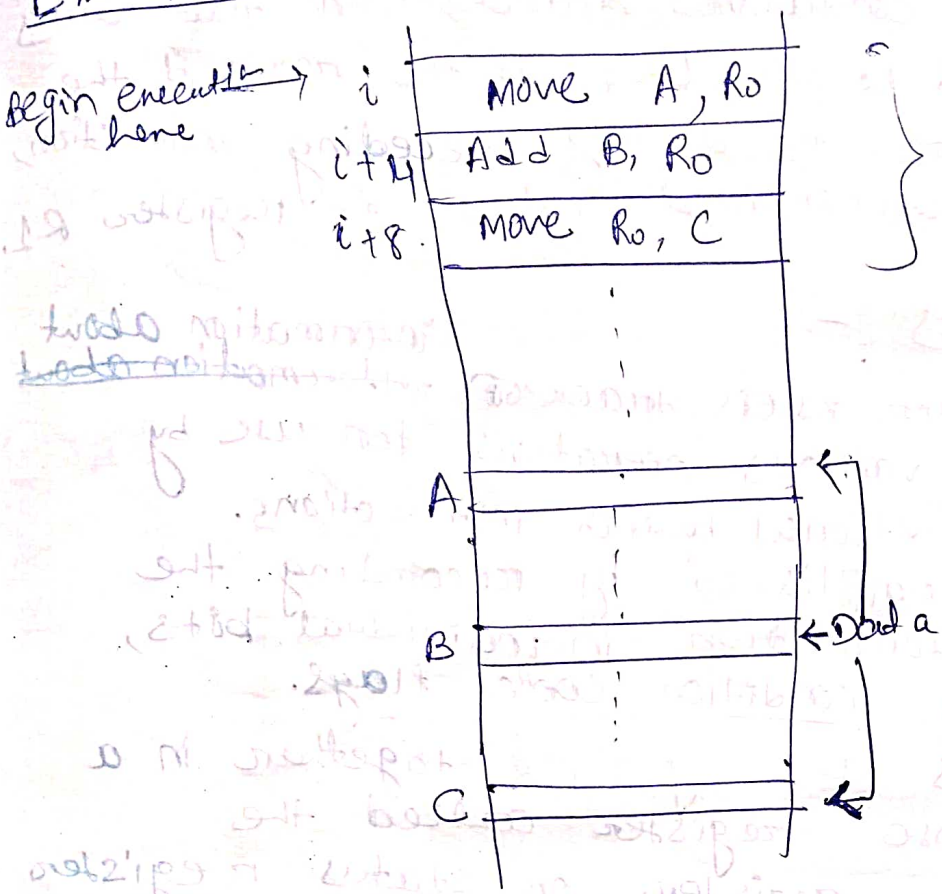
$$\Rightarrow P = 2^{20} - 250 * 2^{12}$$

$$= 2^8 * 2^{12} - 250 * 2^{12} = 6 * 2^{12}$$

Instruction execution and straight line sequencing :-

Example :- $C \leftarrow [A] + [B]$

→ word length = 32 bit
→ memory is byte addressable.
1 word = 4 B



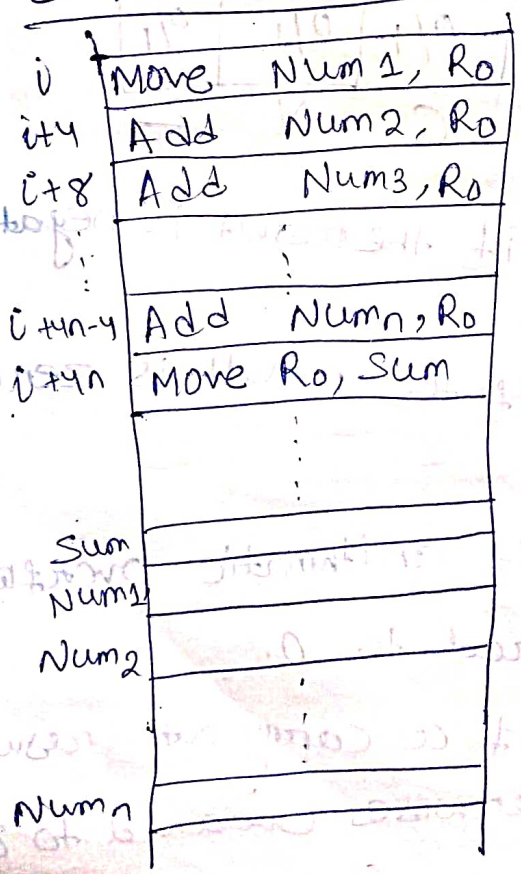
3 instruction

Assume 1 instrn can contain 1 memory operand.

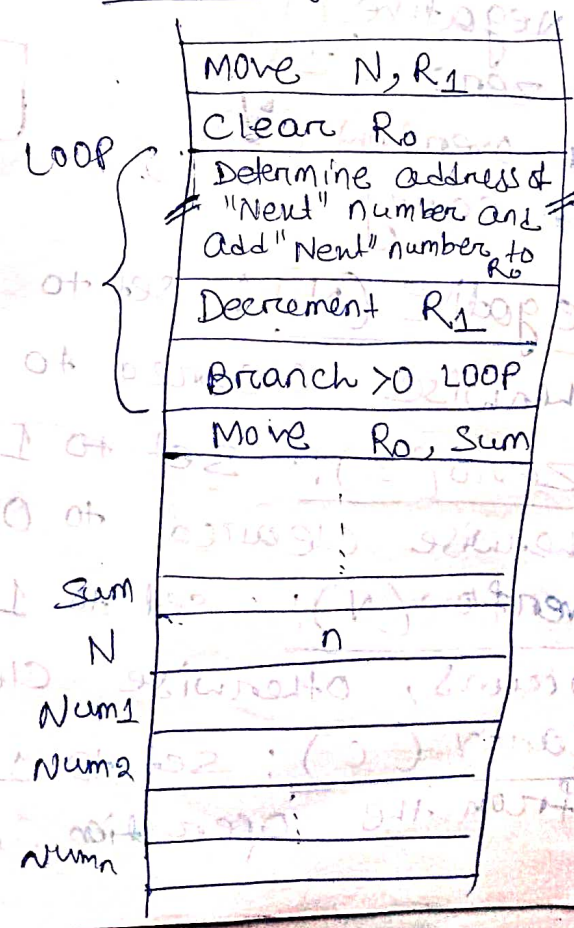
Branching :-

Ex :- task of adding a list of n numbers.

Sequence-execution



Branching



Branch >0 LOOP

↳ it is a conditional branch instr that causes a branch to location LOOP if the result of the result of the immediately preceding instruction, which is the decremented value in register R1.

Condition codes :-

→ The processor keeps track of information about ~~information about~~ the results of various operations for use by subsequent conditional branch instructions.

→ This is accomplished by recording the required information in individual bits, often called condition code flags.

→ These flags are grouped together in a special purpose register called the condition code register or status register

→ There are 4 commonly used flags are

* Negative (N)
* zero (Z)
* overflow (V)
* carry (C)

0/1	0/1	0/1	0/1
N	C	V	Z

→ Negative (N): set to 1 if the result is negative otherwise cleared to 0.

→ zero (Z): set to 1 if the result is zero otherwise cleared to 0.

→ Overflow (V): set to 1 if arithmatic overflow occurs, otherwise cleared to 0.

→ Carry (C): set to 1 if a carry out results from the operation, otherwise cleared to 0

CPU organization :-

→ There are 3 types of CPU organization.
* General register organization.
* Single accumulator organization
* Stack organizatization.

→ 3-address instrn & 2-address instrn belong to general register organization.
→ One-address instrn belongs to single accumulator organization.
→ Zero-address instrn belongs to stack organization.

Addressing Modes :-

→ The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

→ There are different types of addressing modes :
* Implied mode
* Register mode
* Absolute mode. (Direct mode)
* Immediate mode
* Indirect mode
  → Register indirect mode.
  → memory indirect mode.
* Index mode
* Relative mode
* Base mode.
* Auto increment mode
* Auto decrement mode.

① Implied Addressing mode :-

→ In this mode, operands are specified implicitly in the definition of the instruction.

Ex :- COM ; complement accumulator.
ADD in zero-address instruction.

→ All register reference instruction that uses accumulator and all zero address instruction are implied mode instruction.

② Register Addressing mode :-

→ In this mode, the operand is the contents of the processor register.

Ex :- ADD R_0, R_1 → Register addressing mode

③ Absolute/Direct Addressing mode :-

→ In this mode, the operand is the memory location or memory address.

Ex :- ADD A, B → Absolute addressing mode

ADD A, R_0 → Register and Absolute A.M.

④ Immediate Addressing mode :-

→ In this mode, the operand is given explicitly in the instruction.

Ex :- MOV #5, R_0
MVI 5, R_0

→ These are useful to initilize the register with some constant value.

High level language :- A = B+6

Move B, R_1
Add #6, R_1
Move R_1, A.

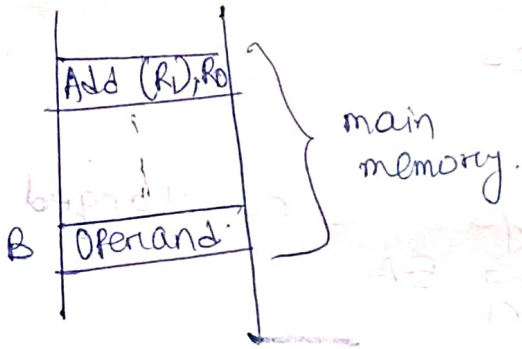
(5) Indirect addressing mode :-

Effective Address → Provides information from which the memory address of the operand can be determined.

→ In this mode, the effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

(i) Register Indirect addressing mode :-

→ In this mode, the effective address of the operand is the contents of a register



R_1 [B] register

(Through a general purpose register)

Ex R_0 [10]

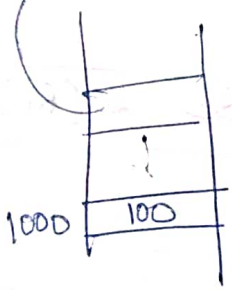
R_1 [1000]

Add (R_1), R_0

$\Rightarrow R_0 = R_0 + (R_1)$

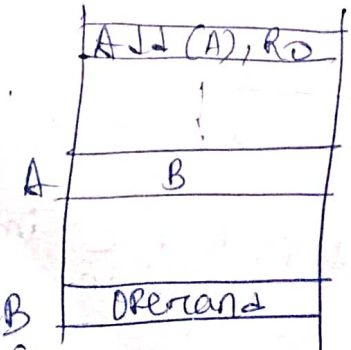
$= 10 + 100$

$= 110$



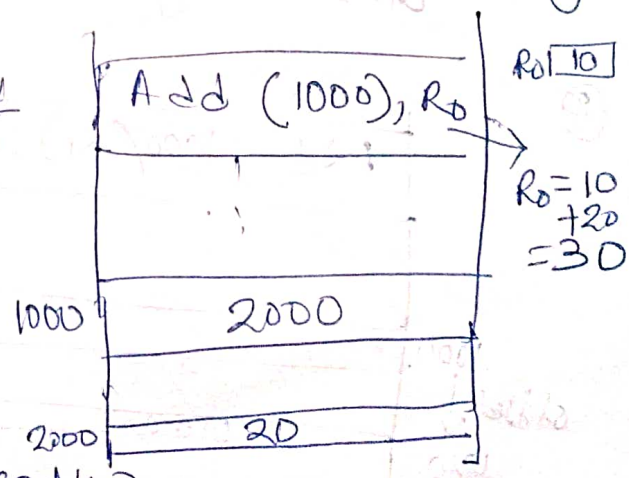
(ii) Memory Indirect addressing mode :-

→ In this mode, the effective address of the operand is the content of a memory location.



(Through a memory location)

Ex



⑤ Index addressing mode :-

→ In this mode, the effective address of the operand is generated by adding a constant value to the contents of a register.

→ This addressing mode is useful in arrays and lists.

→ Here, one register is used called as index register. It may be a general purpose register or a special register.

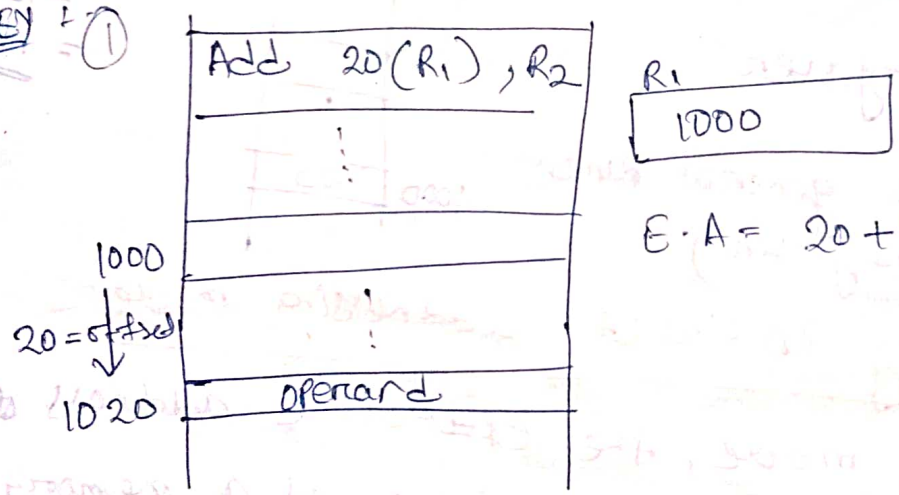
→ symbolically indicated as,

$X(R_i)$: Where X denotes as a constant value contained in the instruction and R_i is the name of register involved.

$$E.A. = X + [R_i].$$

→ The content of index register is not changed in the process of generating EA.

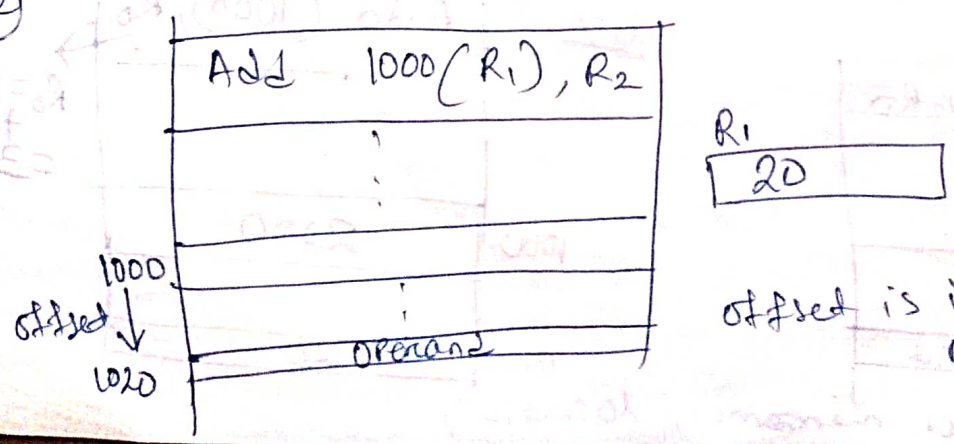
Ex :- ①



E.A = 20 + 1000 = 1020

offset is given as a constant.

②



offset is in the index register

7 Relative Addressing mode :-

→ In this mode, the effective address is determined by the index mode using the program counter in place of the general purpose register.

→ symbolically indicated as X(PC)

→ to address a memory location that is X bytes away from the location presently pointed to by the program counter.

→ This mode is mostly use to specify the target address in branch instruction.

Ex :- Branch > 0 LOOP

→ if condition satisfy then the execution starts from LOOP location. This location can be computed by specifying it as an offset from the current value of the program counter

→ offset may be +ve or -ve i.e. signed number.

8 Base addressing mode :-

→ In this mode, the content of the base register is added to the ~~address of~~ another register presented in the instruction.

(Ri, Rj) → EA = Ri + Rj

Rj → base register → it contains offset value (X).

Ri → index register

→ This mode provides more flexible because both components of the effective address can be changed.

X (Ri, Rj) : EA = Ri + Rj + X → useful in 3D array.

(9) Auto-increment addressing mode :-

→ In this 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 point to the next item in a list.

→ indicated as $(R_i)+$

→ implicitly increment amount is 1.

(10) Auto decrement addressing mode :-

→ In this mode, the contents of a register specified in the instruction are first automatically decremented and are then used as the effective addressing of the operand.

→ indicated as $-(R_i)$

Ex:- WAP to add a list of N numbers using indirect addressing.

```
        Move    N, R1        } Initialization.
        Move    # Num1, R2   }
        Clear   R0           }
LOOP :  Add     (R2), R0
        Add     # 4, R2
        Decrement  R1
        Branch >0  LOOP
        Move    R0, SUM
```

NUM1 : Address of 1st no. in the list

Sub $(-R_1), R_2$

R_1 [1000] $R_2 = 10$

1000 [100] 99

Ex-2:- WAP to add a list of N numbers using auto increment addressing mode.

```
        Move      N, R1
        Move      #NUM1, R2     } Initialization
        Clear     R0
LOOP:   Add       (R2)+, R0
        Decrement R1
        Branch >0 LOOP
        Move      R0, SUM
```

Q

A two word instrn Load is stored at location 1000 with its address field at location 1001. The address field has the value 2000 and the value stored at 2000 is 5000 and at 5000 is 6500. The words stored at 2200, 3002 are 3500, 4000 respectively. An index register has value 200. Evaluate the effective address and operand if addressing mode of the instrn is as follows:

I. Memory Indirect addressing mode
II. Relative addressing mode
III. Index addressing mode

Ans :-

1000	Load
1001	2000
PC → 1002	Next instrn
2000	5000
2200	3500
3002	4000
5000	6500

Index Register R_1 = 200

I. Memory Indirect A.M

(A) = (2000)

E.A = 5000, Operand = 6500

II. Relative A.M

2000 (PC) PC = 1002

E.A = 2000 + 1002 = 3002

Operand = 4000

III. Index A.M

2000 (R_1)

E.A = 2200 Operand = 3500

(i) Load (2000), AC

AC = 6500

E.A = 5000

Load R_1 (2000), AC

E.A = 200 + 2000 = 2200, AC = 3500

(ii) Load PC (2000), AC

E.A = PC + 2000

= 1002 + 2000 = 3002

AC = 4000

Q. Let a processor operates by a frequency 20 MHZ and it executes a program having 200 instrn out of which

100 ← 50 % are register referenced instrn
60 ← 30 % are memory " "
40 ← 20 % are branch instrn.

The register referenced, memory referenced and branch instrn will take 4, 8, 6 clock cycles respectively, then find out the total time required by the processor to execute the program.

Ans :-

Register reference → $100 * 4 = 400$
memory " → $60 * 8 = 480$
Branch instrn → $40 * 6 = 240$
Total cycles = 1120 clock cycles

20 MHZ frequency

1 clock cycle $= \frac{1}{20\,M}$ sec $= \frac{1}{20 * 10^6}$ sec

$= \frac{1000}{20}$ nsec

$= 50$ nsec.

Total time $= 1120 * 50$ nsec
$= 56000$ nsec $= 56$ μsec.

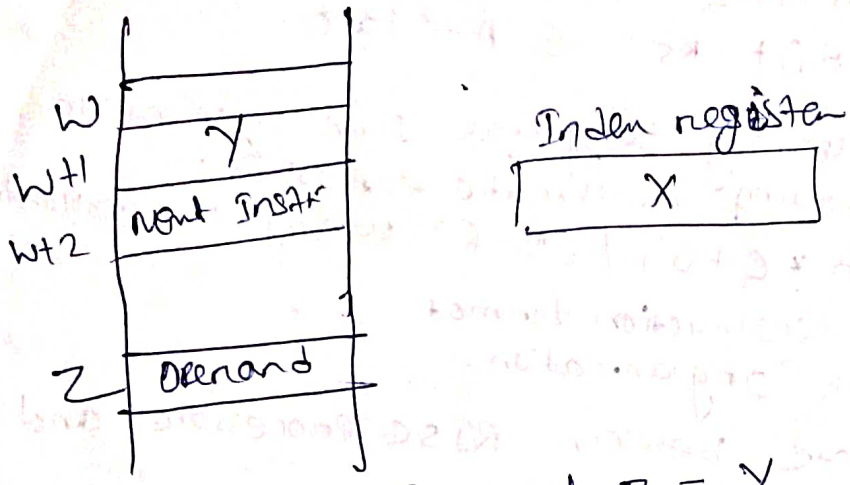
Q// There are two 8 bit register 'R1' and 'R2' contains the values −5 and −125 respectively. Find out the values of the status bit of V (overflow), S (sign), Z (zero), C (carry) flags when the following instrn is executed. Add R_1, R_2

	128	64	32	16	8	4	2	1
−5 =	1	1	1	1	0	1	1	
−125 =	1	0	0	0	0	1	1	
	1	0	1	1	1	1	1	0

V	S	Z	C
1	1	0	1

Problems on Addressing modes :

Q. A two word instruction is stored at an address designated by a symbol W. The address field of the instruction is stored at W+1, is designated by Y. The operand used during the execution is stored at an address designated by Z. An index register contain the value x. State how Z is calculated if the addressing modes are direct, Indirect, relative, index addressing mode



Direct A.M ∴ E.A of Z = Y

Indirect A.M E.A of Z = [Y]

Relative A.M E.A of Z = Address Part + [PC]
= Y + W + 2

Index A.M E.A. of Z = Address Part + [Indexreg]
= Y + x

Q. What must be the address field of an index addressing mode instruction to make it same as the register Indirect addressing mode instn

Ans

EA = X + [R1]

EA = [R1]

⇒ X + [R1] = [R1]

⇒ x = 0

Q. Calculate the total no. of memory reference to execute the following program.

```
      MOV  #5, R1      → 1
Loop  Add  (R0), R2    → 2
      DEC  R1          → 1
      BR   R1>0  Loop  → 2
```

Total time = 1+5(2+1+2)
= 1 + 5*5
= 26

↳ memory reference

Q.

	Mode	Load to AC
200		
201	Address = 500	
202	Next Instruction	
	⋮	
399	450	
400	700	
	⋮	
500	800	
	⋮	
600	900	
	⋮	
702	325	
	⋮	
800	300	
	⋮	

PC [200]
R1 [400]
XR [100]
(Index Register)
BR [202]
(Base Register)
AC [?]

Find the effective address and the contents of Accumulator for all addressing modes.

(1) Implied addressing mode : This is not an implied mode

(2) Immediate addressing mode

Load #500, AC
E.A = 201 , AC = 500

(3) Register Addressing mode

Load R1, AC
, AC = 400 , NO effective address

(4) Direct / Absolute Addressing mode

Load 500, AC
EA = 500, AC = [500] = 800

(5) Register Indirect Addressing mode

Load (R1), AC , EA = 400 [400] = 700 or AC = 700

6) Memory Indirect Addressing mode

load (500), AC

EA = 800

AC = [800] = 300

(7) Index Addressing mode

load X (XR), AC

EA = X + [XR] = 500 + 100 = 600

AC = 900

(8) Relative Addressing mode

load X(PC), AC

EA = X + [PC] = 500 + 202 = 702

AC = [702] = 325

(9) Base Addressing mode

load X(BR), AC

EA = X + [BR] = 500 + 202 = 702

AC = [702] = 325

(10) Auto Increment mode

load (R1)+, AC

EA = [R1] = 400 R1 = 401

AC = [400] = 700

(11) Auto Decrement mode

load -(R1), AC

~~EA = [R1] = 400~~ R_1 = 399

~~AC = [399] = 450~~

EA = 399, AC = [399] = 450

Assembly Language

Mnemonics :- symbolic names to represent the patterns.

Move → MOV
Add → ADD
Increment → INC
Decrement → DEC
Branch → BR

→ A complete set of symbolic names and rules for their use constitute a programming language, referred ~~to~~ as an **assembly language**.

→ The set of rules for using the mnemonics in the specification of complete instructions and programs is called the **syntax** of the language.

→ Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an assembler.

Assembler : Assembly language → m/c instn

Source programme → Object program

ADD #5, R_3 → Immidiate addresing mode

or

ADDI 5, R_3 (I → Immediate addressing mode)

→ A given instr^n has different OP-code mnemonics for different addressing mode

→ In some assembly language, the intended addressing mode is indicated in the OPcode mnemonics.

MOVE #5, (R_2)

or MOVEI 5, (R_2)

<u>Difference betn</u>

Move #5, R_2 and Move #5, (R_2)

~~R_2 is the~~

<u>O/P</u> R_2 [~~105~~] 5

<u>O/P</u>

R_2 [1000]

1000 [~~105~~]

Assembler Directives :-

Source Program

SUM = 200

Assembler Program

SUM EQU 200

~~It is not an instruction that will when object prog~~

→ This statement does not denote an instruction that will be executed when the object program is run; in fact it will ~~never~~ not even appear in the ~~source~~ object program.

→ It simply informs ~~that~~ the assembler that the name SUM should be replaced by the value 200 whenever it appears in the program.

→ Such statements called assembler directives (or commands), are used by the assembler while it translates a source program into an object program.

Example

→ Add N numbers present in list

→ Assume Each instn takes 1 word
→ Byte addressable
→ 1 word length = 32 bits.

100	Move N, R1
104	Move #NUM1, R2
108	Clear R0
112	Add (R2), R0
116	Add #4, R2
120	Decrement R1
124	Branch >0 LOOP
128	Move R0, SUM
132	
SUM 200	
N 204	100

NUM1 208	
NUM2 212	
	⋮
Num_n 604	

	Memory address label	Operation	Addressing or data information
Assembler Directives	SUM	EQU	200
		ORIGIN	204
	N	DATA WORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate m/c instructions	START	MOVE	N, R_1
		MOVE	#NUM1, R_2
		CLR	R_0
	LOOP	ADD	(R_2), R_0
		ADD	#4, R_2
		DEC	R_1
		BGTZ	LOOP
		MOVE	R_0, SUM
Assembler directives		RETURN	
		END	START.

(Assembly language representation for the program)

→ Equate Directive (EQU): is used to inform the assembler the name replace by a value throughout the program (EX: SUM = 200)

→ ORIGIN Directive: is used to inform the assembler in which memory ~~location~~ to place that data block

→ DATAWORD directive: is used to inform the assembler that value 100 is to be placed in the memory word at address 204.

→ RESERVE directive: declares that a memory block of 400 bytes is to reserved for data and the name NUM1 is to be associated with address 208

Assembly and Execution of programs :-

→ **Symbol table** :- When assembler program scans a source program, it keeps track of all names and numerical values that corresponds to them in a symbol table.

→ **Two pass assembler** :-

* During first pass, it creates a complete symbol table. At the end of this pass, all names will have been assigned numeric values.
* In second pass, the assembler substitute values for all names from the symbol table.

such type of assembler is called two-pass assembler.

→ **Loader** :- It is an utility program which transfer the object program from the magnetic disk to the specified main memory.

Debuggers :- It is a system software which is able to find errors (syntax error) in the program

Basic Input / Output Operations :-

Program - controlled I/O :-

A task that reads characters input from a keyboard and produces characters output on a display screen. To perform such I/O tasks is to use a method known as program-controlled I/O.

→ DATAIN is a buffer register that contains character entered through keyboard.

→ SIN is a status control flag is set to 1 if DATAIN contain valid character and it informs to processor.

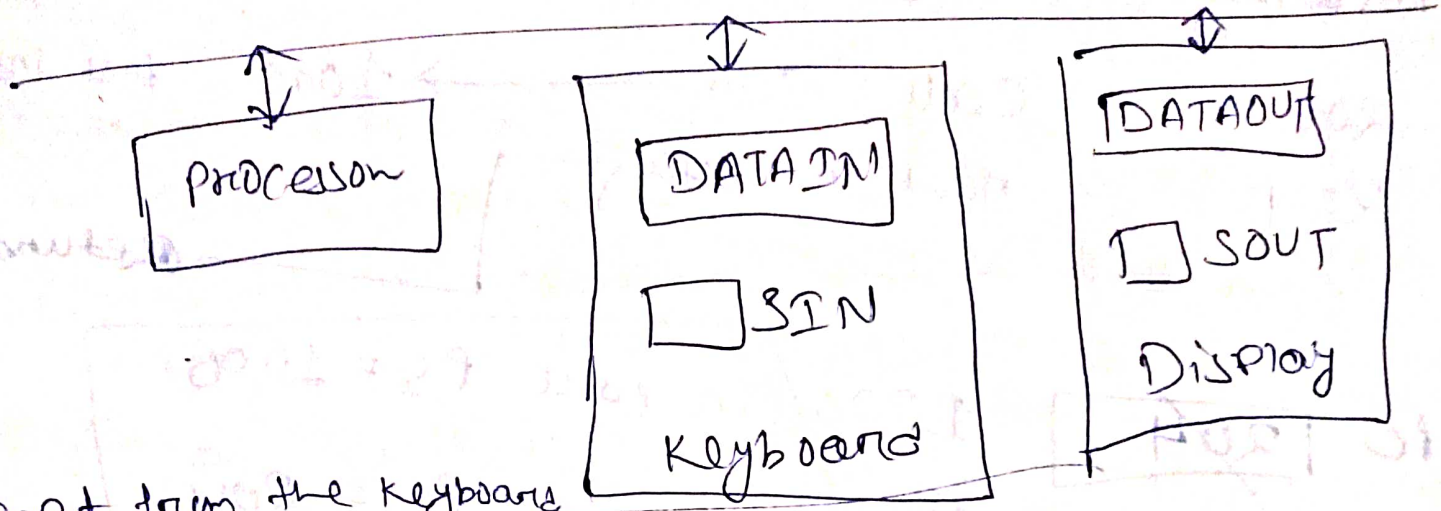
→ A program monitors SIN and when SIN is set to 1, the processor reads the contents of DATAIN.

→ When the character is transferred to the processor, SIN is set to 0.

→ A buffer register, DATAOUT and a status control flag, SOUT are used for the data transfer between processor and display.

→ When SOUT is set to 1, the processor transfers a character code to DATAOUT.

→ After transfering the character to DATAOUT, clears SOUT is 0, when the display device is ready to receive a second character. Then again SOUT is set to 1.



Input from the keyboard

READWAIT Branch to READWAIT if SIN = 0 Input from DATAIN to R1.

Output to Display

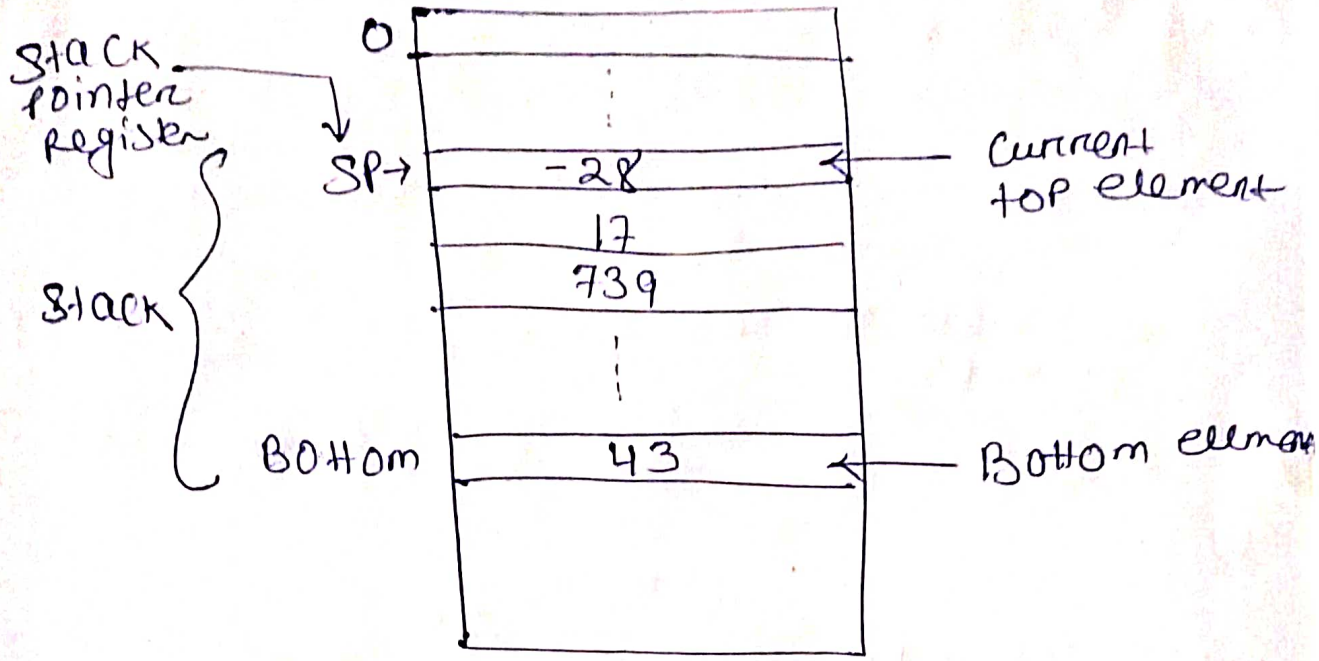
WRITEWAIT Branch to WRITEWAIT if SOUT = 0 Output from R1 to DATAOUT

Memory mapped I/O :-

Some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT.

instr

Stack and Queues :-



PUSH operation

Subtract #4, SP	SP ← SP - 4
MOVE NEWITEM, (SP)	[SP] ← New item

or

MOVE NEWITEM, -(SP)	Auto decrement SP = SP - 4 [SP] ← New item.

POP operation

MOVE (SP), ITEM	ITEM ← [SP]
Add #4, SP	SP ← SP + 4.

or

MOVE (SP)+, ITEM	Item ← [SP] SP ← SP + 4

Compare instruction :-

compare src, dst : [dst] - [src]

and sets the condition code flags according to the result. It does not change the value of either operand.

SAFEPOP	Compare Branch>0	#2000,SP EMPTYERROR	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare Branch≤0	#1500,SP FULLERROR	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Move	NEWITEM,−(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

(b) Routine for a safe push operation

Subroutines :- A Branch and Link (BL) instruction is used to call subroutine

→ A particular subtask are performed many times in a programs. such a subtask is called as subroutines.

→ **Call instruction :-**

* Store the content of PC in the link register
* Branch to the targed address specified by the instruction.

→ **Return Instruction :-**

* Branch to the address contained in the link register.

Memory Location	Calling Program	Memory location	Subroutine SUB
200	Call SUB →	1000	first instr
204	next instruction ←		⋮ Return

PC $\boxed{204}$ 1000

Link $\boxed{204}$

in call PC = 1000

in return PC = 204

Example

Calling program:-

```
Move    N, R1
Move    #NUM1, R2
Call    LISTADD
Move    R0, SUM
  ⋮
```

Subroutine

```
LISTADD   Clear   R0
LOOP      Add     (R2)+, R0
          Decrement R1
          Branch>0  LOOP
          Return
```

Subroutine nesting and the processor stack

Subroutine nesting → One subroutine call another subroutine.

→ Problem is the previous information of link register when calling a new subroutine.

→ So there is need to save the contents of the link register in some other location before calling another subroutine

→ last in first out order.

→ Stack is used to save the content of link register

→ Stack pointer (SP) is a register which points to a stack called the processor stack.

→ The call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC.

→ The return instruction pops the return address from the processor stack into the PC.

Parameter passing :-

→ parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. (Ex previous page)

→ or the parameters may be placed on the processor stack used for saving the return address.

Ex :-

```
Move    #NUM1, -(SP)
Move    N, -(SP)
call    ListAdd
Move    4(SP), SUM
Add     #8, SP
:
ListAdd
```

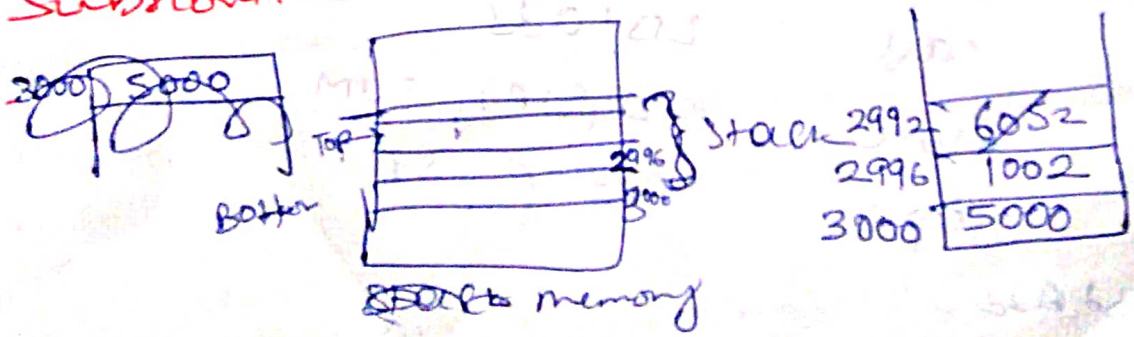
The content of the top of memory stack is 5000. The content of the stack pointer SP is 3000. Assume you want to organize a nested subroutine calls on a computer as follows: the routine main calls a subroutine SUB1 by executing a two-word call subroutine instruction located in memory at address 1000 followed by the address field of 6000 at location 1001. Again subroutine SUB1 calls another subroutine SUB2 by executing a two-word call subroutine instruction located in memory at address 6050 followed by address field of 8000 at location 6051. What are the contents of PC, SP and top of stack?

(i) After the subroutine call instruction is executed in main routine? SP=2996, TOS=1002, PC=6000

(ii) After the subroutine call instruction is executed in the subroutine SUB1? SP=2992, TOS=6052, PC=8000

(iii) After the return from SUB2 subroutine. SP=2996, TOS=1002, PC=6052

Ans: (i)



Q. Suppose you want organize a nested subroutine call for which all the instn and its corresponding memory address are given below. Here, the main program makes a subroutine call named 'SUB1' and 'SUB1' makes another subroutine called named 'SUB2'. The content of the stack pointer (SP) and top of memory stack (TOS) is 4000 and 5000 respectively.

Main Prog: Address	Instn	SUB1 Address	Instn	SUB2 Address	Instn
1000	Add R_1, R_2	2000	Sub R_0, R_3	3000	Mul R_1,
1004	Call SUB1	2004	Call SUB2	3004	Div R_5, R_6
1008	Sub R_3, R_2	2008	Mul R_6, R_3	3008	Return
1012	Return	2012	Return		

What is the content of PC, SP and TOS for the following

I. After the subroutine call instn is executed in the main program?

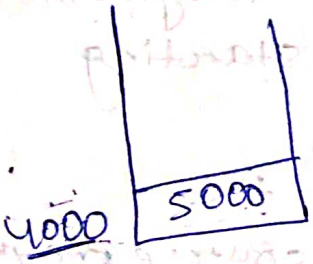
II. After the subroutine call instn is executed in the subroutine SUB1?

III. After return from the subroutine SUB2?

~~PC = 1008 2000~~

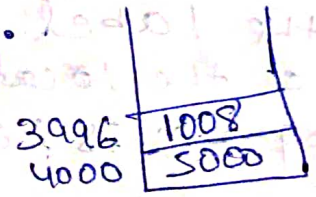
~~2008 3000~~

~~PC = 2008~~



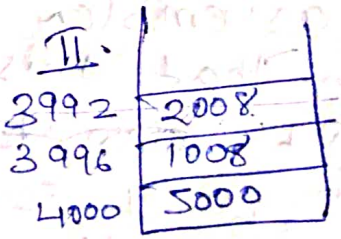
SP = 4000
TOS = 5000

I.



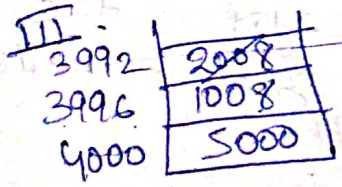
SP = 3996
TOS = 1008
PC = 2000

II.



SP = 3992
TOS = 2008
PC = 3000

III.



SP = 3996
TOS = 1008
PC = 2008

8/ Consider a processor with 64 registers that ~~operands~~ supports twelve instrns. Each instrn has 5 distinct fields, namely opcode, two source register fields, one destination register field and a twelve bit immediate value. Each instrn must be stored in memory in a byte aligned fashion. If a program has 100 instrn. What is the amount of memory (in bytes) consumed by the program

Ans:- 64 reg $= 2^6$ → 6 bit for register

OP code	SRC Reg 1	SRC Reg 2	Dest Reg	Imm value
4 bit	6	6	6	12

12 instrn → 4 bit for opcode

Total bits $= 4 + 6 + 6 + 6 + 12 = 34$ bits

$= \frac{34}{8}$ bytes $= 4.25$ bytes.

1 instrn → 5 bytes (.75 bytes waste)

100 instrn → $5 * 100 = 500$ bytes.