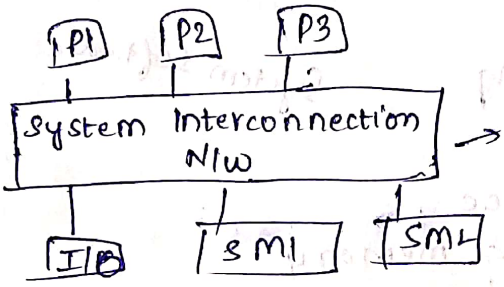
Computer Architecture

Performance
- Technology (VLSI → speed, storage etc)
- parallelism
  - → Instruction set
  - → multiprocessors
  - → multicores

(~~multiple processors~~)

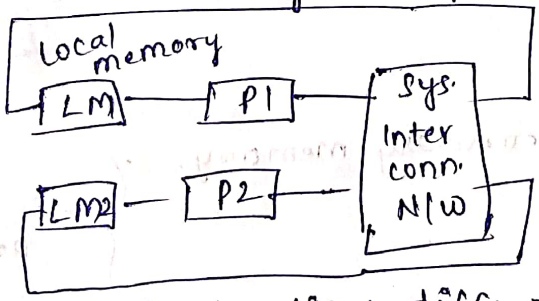
multicomputers & multiprocessors
- → UMA (Uniform Memory Access)
- → NUMA (Non Uniform " ")
- → COMA (Cache Only " ")

UMA:



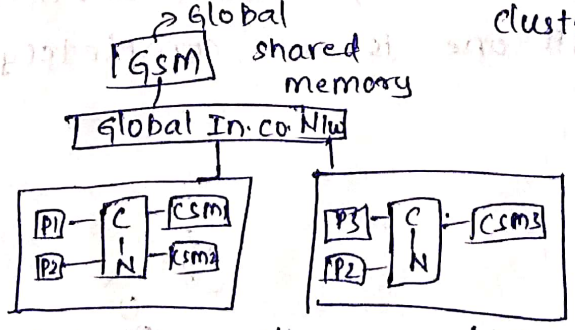
Here each processor takes the same time to access a memory (eg. SM1) (uniform time)

NUMA: Here memory is seperated physically to each processor



P1 → sys. In. co. Nlw → LM2
P1 → LM1
P2 → sys. In Nlw → LM1
Δt

Here there will be time diff. for accessing diff memories by a processor, due to memory location.

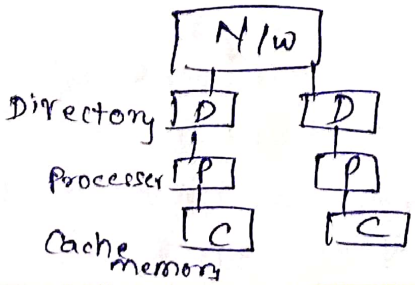


CIN - cluster Intercon. Nlw

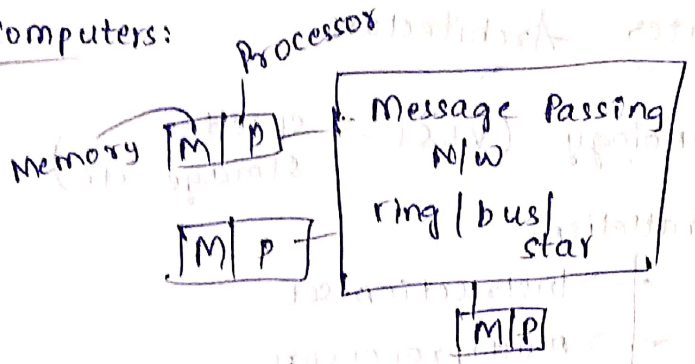
Cluster Hierarchical structure.

↳ Here each cluster acts as UMA

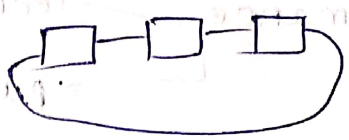
COMA:



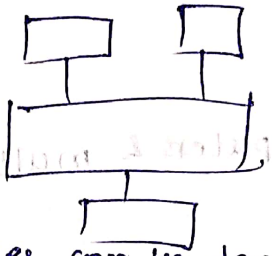
Multi Computers:



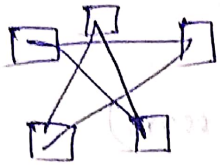
ring



bus



star:



UMA → Symmetric → If all the processors are done by all processors then they are symmetric

UMA → Asymmetric

System software:

Software → entering/editing
Software → linking
Software → I/O interface
Software → managing memory
Software → Transfering high level lang. to low level lang.

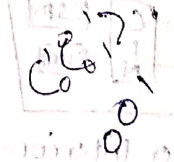
OS - interface b/w user & hardware.

Multiprogramming:

1. Get application from secondary memory.
2. exec → read data
   compute
   write

} for any pro.

Whenever the CPU is idle, it loads another application even when the 1st one is not completely executed.



Number Representation

$$B = b_0 b_1 b_2 \cdots b_n$$

$$V(B) = b_0 \times 2^{n-1} + b_1 2^{n-2} + \cdots + b_n 2^0$$

Sign magnitude
1's complement
2's complement
} Representations

→ Subtract the binary no. from 2^n

→ " " " $2^n - 1$

So, add 1 to get 2's from 1's

Memory operations: load, store

Memory organization: word

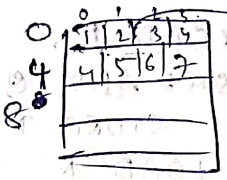
n bits - word
↳ 16 to 64 bits

1 byte
ASCII ~~value~~ of length 0-8 characters can be stored in one word

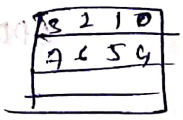
32 bit

byte addressable

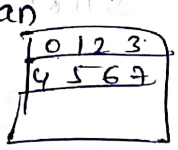
0 → 1 byte
1 → 2 byte
2 → 3 byte
⋮



byte addressable
- Little Endian
- Big Endian



Word alignment



2^k - addressable locations if we have a k-bit register

Sign Extension: Adding 1's on the left hand side of a 2's complemented bin. no. doesn't make any effect.

Instructions: (Based on operations: Datatransfer; conversions, logical, arith operations)

OPCODE OPERAND

Eg: ADD R1, R2

	unsigned	signed
	110101	110101
logical left →	101010	101010
right →	011010	111010

shift: logical, arithmetic (left, right)

ari. left: 110001 → 10011

ari. right: 11001 → 11100

ADD R1, R2
source, desti.

ADDI #25
Add Immediately → will be sent to accumulator.

Sources — registers, memory locations, immediate → destination

Instructions: (Based on # of operands)

(1) 3 address — OPCODE Mem/reg M/r M/r

Eg: $X = (P+Q) \times (R+S)$

```
ADD   R1, P, Q
ADD   R2, R, S
MUL   X, R1, R2
```

(2) 2 address — OPCODE M/r M/r

```
MOV   R1, P
Add   R1, Q
MOV   R2, R
Add   R2, Q
Mul   R1, R2
MOV   X, R1
```

(finite) no. of bits required for opcad, operand

(If we have definite then it becomes easy for operation)

(3) 1 address — LOAD, STORE

```
LOAD  P
ADD   Q
STORE T
LOAD  R
ADD   S
MUL   T
STORE X
```

(4) zero address (no operand references) are used (stack)

We use when we have stack organization in the system

operation – top 2 ele.

```
PUSH P
PUSH Q
ADD
PUSH R
PUSH S
ADD
MUL
POP X
```

Sequencing:

$C \leftarrow A+B$

Here instructions will be executed linearly

i	LOAD R1, A
i+4	ADD R1, B
i+8	STORE C, R1
i+12	

considering each i/n to take 4 bytes. (assuming)

i.e., after the execution of i^{th} instruction, $(i+4)^{th}$ will be executed (address)

Branching / Looping:

```
LOOP  LOAD  R1, N
loop  { add element to reg R0
        and increment
        decrement R1
```

(Arithmetic overflow)

i.e., checks if $R1 > 0$ ← the last instru. ← BRANCH > 0 LOOP

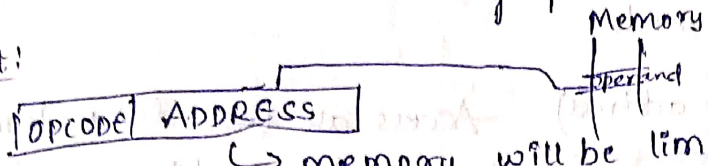
o/p will be checked MOVE SUM, R0

Addressing modes: → Immediate
→ direct
→ indirect

Immediate: MOVE A,#50 #50

OPCAD OPERAND
↳ memory will be limited for data
↳ accessing speed will be fast

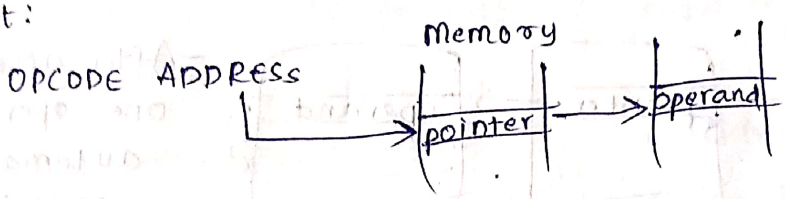
Direct:



↳ memory will be limited for address space

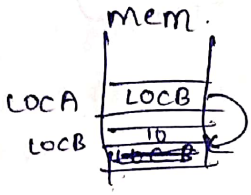
ADD R1, 4016H

Indirect:

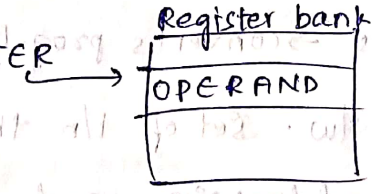


not restricted with range of the memory.

Eg: ADD R, LOCA

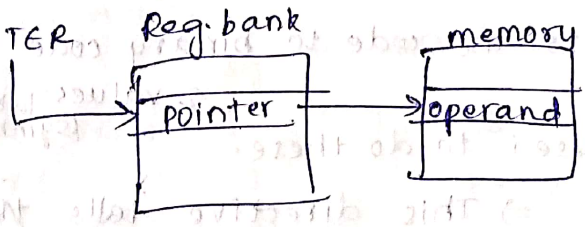


Register direct:



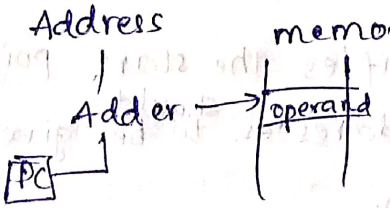
range is limited
faster access.
No need access data from any loc.

Register indirect:

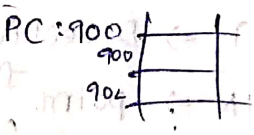


not limited in address range

PC mode
Relative:



Eg: Address: 30



Now: PC = 900+2+30
↳ Addr
= 932
↳ operand at this address

Stack mode:

Operates on top two data values.
Stack pointer → points to the top
ADD
↳ Adds top 2 ele.

Base addressing mode:

Here we add the address we have to the base address to access the operand.

OPCODE ADDRESS

adder – operand

Base register
(or) Segment register

array of data elements.

index register addressing mode:

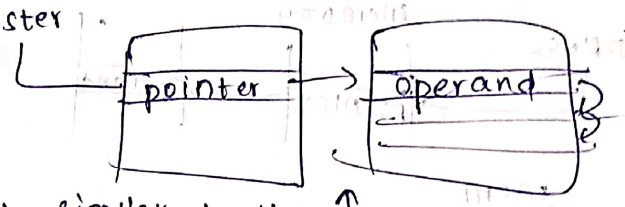
OPCODE address (base address)

adder

index

→ Access all the data elements from base address (address) to index

auto increment:



→ After accessing one operand it automatically access the next operand

auto decrement: similar to this ↑

Assembly Language:

Mnemonics: ADD, MOV, BR, LOC, INC

mnemonics (symbols) + rules → converts prog. lang. to assembly lang.

Assembler: part of system s/w. Set of i/n that helps is stored in memory.

A.L → must specify the addressing mode.

assembler converts the code to binary code

↓

Directives are used to do these.

& values. labels (symbols)

MOV A, #40

MOVI A, 40

LOOP EQU 100 ⇒ This directive tells that to the value of LOOP is 100.

START ⇒ Directive that specifies the start point of execution (code)

ORIGIN ⇒ point from where the addresses should to be given for i/n

END ⇒ end of the execution.

define storage ⇒ reserves memory for some data items.

define constant ⇒ allows us to reserve some memory before the program execution.

ORIGIN → how to interpret the names → Equate directory ive

↳ where to allocate → origin directive

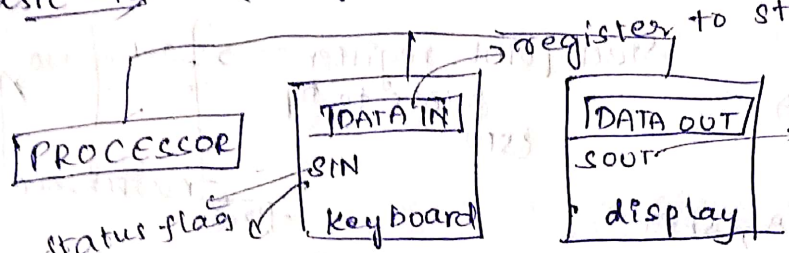
Symbol table: contains the list of symbols along with their respective values.

scans → two scan assemble / two pass assembler.

forward branch (fwd branch);

loader

(Basic I/o) {Program Controlled I/o:



becomes '1' when DATA IN has some values stored & processor senses this & reads the data, once the data is taken out of register to processor,

SIN = 0

READWAIT · BRANCH TO READWAIT if SIN=0

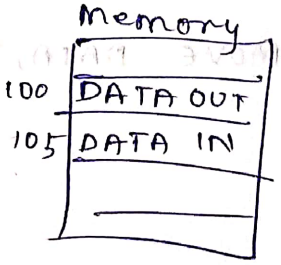
Move input data into R from DATAIN

WRITEWAIT BRANCH WRITEWAIT if SOUT=0

Output to DATAOUT from R.

Initially, SIN=0, SOUT=1

Memory mapped I/o:



MoveByte R, DATAIN

MoveByte DATAOUT, R

READ Testbit INSTATUS, #2 → eg: @0100 #2

BRANCH=0 READ

MoveByte R, DATAIN

'0' so, the status flag=0

Status registers tells the status of each device.

WRITE Testbit OUTSTATUS, #2

BRANCH=0 WRITE

MoveByte DATAOUT, R

MOV R0, LOC

READ Testbit INSTATUS, #3

BRANCH=0 READ

MoveByte R0, DATAIN

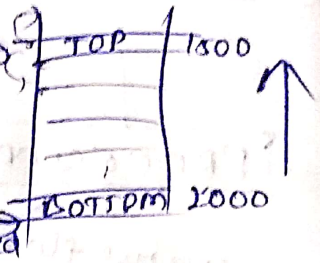
WRITE Testbit OUTSTATUS, #3

BRANCH=0 WRITE

MOVEBYTE DATAOUT, #0

compare CR, #0+ (Auto increment)

STACKS: Memory will allocated from higher to lower addresses



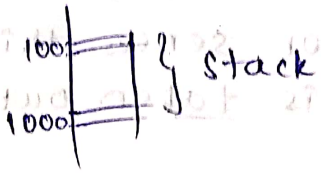
push:

move (SP)-, DATA

pop:

MOVE DATA, (SP)+

Compare (SP), #1000



(destination, source)

dest - sour

returns either +ve or -ve

if o/ps +ve → stack → underflow

~~-ve~~

COMPARE (SP), #1000

BRANCH>0 ~~COMPARE~~ STACK UNDERFLOW

MOVE DATA, (SP)+

↳ POP

MOVE -(SP), ITEM

supports auto increment & auto decrement

SUBSTRACT SP, #4

MOVE SP, ITEM

COMPARE (SP), #100

BRANCH <0 STACK OVERFLOW

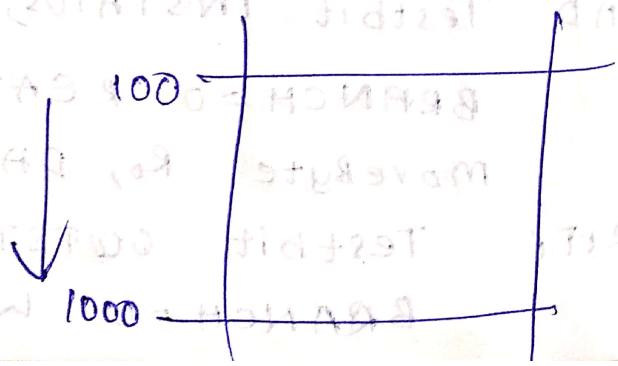
MOVE -(SP), ITEM

Queues: Memory allocation → lower to higher

Circular queue

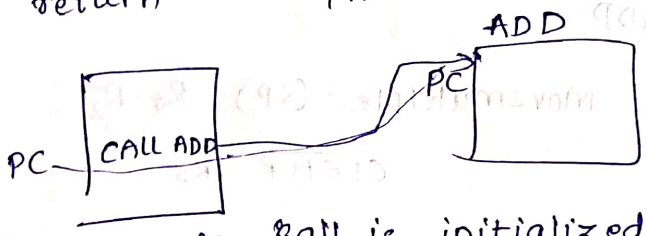
insertion →

deletion →



subroutine:

Set of instr which are to be performed many times at different locations, then we store them in a subroutine & branch to them whenever necessary.
(similar to fns in prog.)
call instruction is used to call the subroutine
return state/instr.



Here the ADD fn. returns to the address stored in LR.

As the fn. call is initialized, the PC now points to the instr. in the ADD fn. (subroutine). So, after the fn. returns, PC to point to the nxt instr. we store the next instr. address in 'Link Register' before pointing to the sub Ro.

```
100 CALL ADD
104 MOVE SUM, R0
```

PC 104 → return.

Link Register (LR) 104

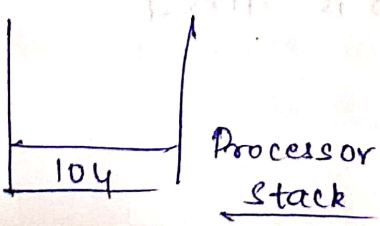
But if the subrout. fn again calls sm other subrout. then the LR now gets replaced with the new address. So, we now lost our previous addresses.

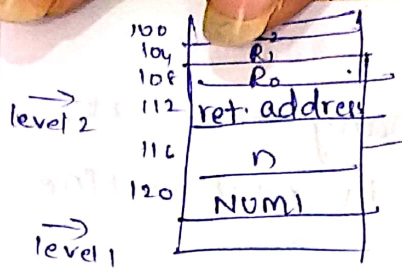
To avoid this we use stacks to store the addresses.
↓
'processor stack'

Passing parameters → registers
→ memory loc.
→ stack

```
MOVE  R1, N
MOVE  R2, NUM1
100 CALL  ADD
104 MOVE  SUM, R0
```

```
ADD   CLEAR R0
LOOP  ADD   R0,(R2)+
      DEC   R1
      BRANCH>0  LOOP
      RETURN
```





→ Stack frame

→ # of elements

4(SP) = mem(SP+4)

16+SP
16+100 → 116 → n

MOVE R1, 16(SP)

MOVE R2, 20(SP)

CALL ADD

MOVE (SP), NUM1

MOVE

ADD Movemultiple -(SP), R0-R2

Move R2, 20(SP)

move R1, 16(SP)

clear R0

LOOP Add R0, (R2)+

decrement R1

Branch >0 loop

Move 20(SP), R0

Movemultiple R0-R2, (SP)+

return

Movemultiple -(SP), R0-R2

CLEAR R0

MOVE (SP), NUM1

MOVE -(SP), N

CALL ADD

MOVE SUM, 4(SP)

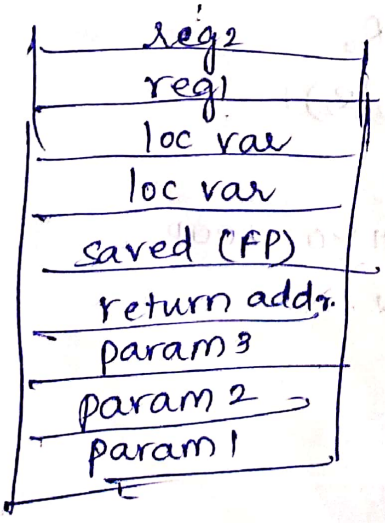
ADD SP, #4

(using return address)

Movemultiple -(SP), R0-R2 → Addressing mode - index mode

Stack frame - used when subroutine is called & is freed up when the subroutine returns.

frame pointer → used to access local variables & the parameter

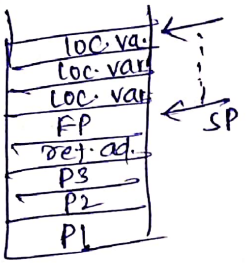


↓
declared inside the subroutine

SP's location changes, if we push / pop

FP's location is fixed

↓
general purpose register

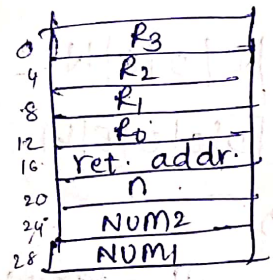


MOVE -(SP), FP

MOVE FP, SP

SUBSTRACT SP, 12

$\sum_{i=1}^{n-1} A(i) \times B(i)$



DOTPRODUCT Movemultiple -(SP), R_0-R_3

Move R_3, 28(SP)

MOVE R_2, 24(SP)

MOVE R_1, 20(SP)

MOVE R_4, R_1+

MUL R_4, (R_2)+

ADD R_0, R_4

R_1 10 bits

R_2 10 bits

MOVE R1, ALOC

MOVE R_2, BLOC

MOVE R_3, N

CLEAR R_0

LOOP Move R_4, (R_1)+

MUL R_4, (R_2)+

ADD R_0, R_4

Decrement R_3

Branch > 0 loop

Move ANS, R_0

~~LSHIF~~

LShifL dest, count

LShifR dest, count

AShifL

AShifR

Mach. instr. encoding:

Assembly lang $\xrightarrow{\text{Assembler}}$ Mach. instr.

OPCODE Dest Src OtherInfo ← Addressing mode etc..

CISC RISC → Reduced Instruction Set Computer

↳ fixed length instr.

Complex Inst. Set Comp. → variable length instr.

Move (R3), (LOC) → No. of memory references ⇒ 3

↓ ↓ Indirect register addressing mode

↳ 2

M = (P+Q) + (R+S)

LOAD R_1, P

LOAD R_2, Q

Pipelining:

Each problem (inst.) is divided into sub problems (inst.)

CISC instr. is converted into RISC

↳ complex ↳ simpler bcoz of fixed length

F | D | C | O | E | W

F | D | C | O | E | W

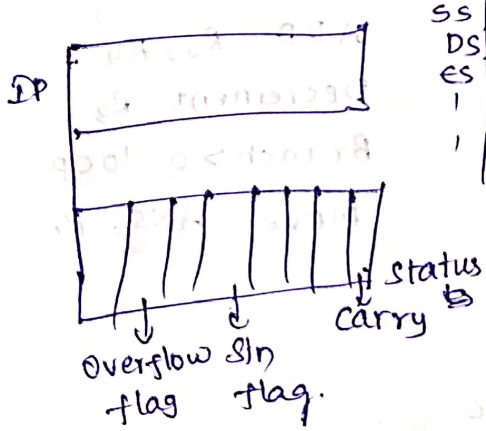
CISC
- IBM 360/370
- Intel X86/pentium
- VAX 11/780
- IA32

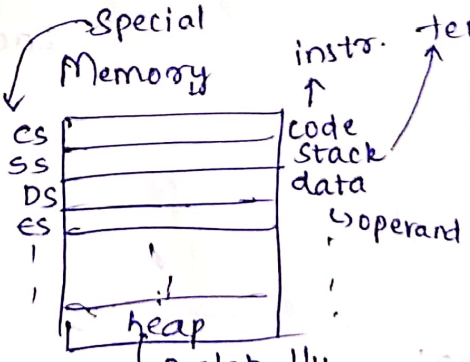
RISC (doesn't support all addressing modes)
- ARM
- MIPS
- CDC 6600

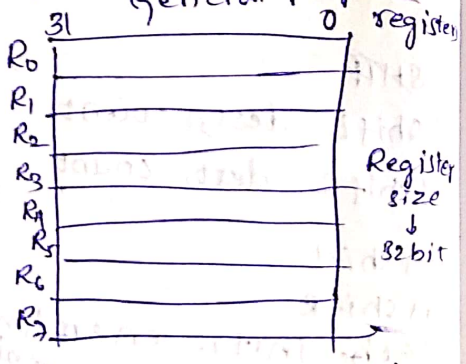
Backward compatibility (even the old pro./inst. can be executed after the updates also)

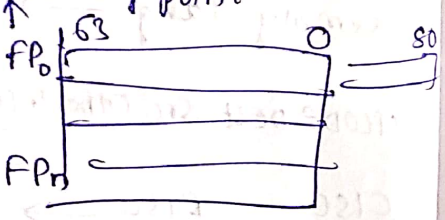
IA32:

Registers:









double word registers

Selectors → to point to certain mem. point (Eg: code, stack)
↳ 6 registers

Instr. pointer

Effective address (address which will be achieved by performing some operations during execution)

Eg: EA = Mem [R_1 + 12]

Addressing modes:

- Immediate — 8/32 bit operands

Direct: 32 bit

Register direct: we can use any of the 8 general pur. regs

Register Indirect: We place 32 bit integer in the register (any of the 8 gen. p.r)

Base with displacement: 8/32 bit

↳ Base address will be stored in a register (1 of 8)

EA = content of base + Displacement

Index with disp.:

Scale factor – to access the array of elements by some scale factor (gives the bottom limit (range).

EA = content of index (reg (8 1 of 8)) * scale factor + displacement

Base with index:

2 out of 8 gen. reg. are used

EA = Base + index * scale

Base with index & displacement:

EA = Base + (index * scale) + displacement

OPCODE Dest. Src

MOV R, #15 — # – immediate

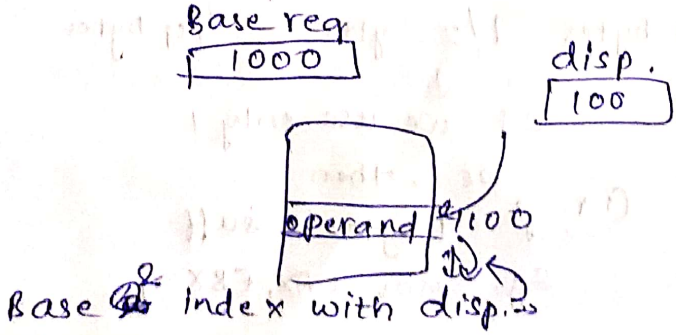
MOV R, [LOC] — [ ] → indicates direct add. mode

Add R_1, $[R_2]$

$R_1 \leftarrow R_1 + Mem[R_2]$

Register direct

Base with displacement

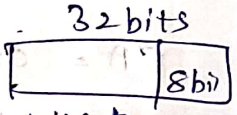


Base & Index with disp.

OPcode dest, src

we can atmost have any one of them (dest, src) as a memory reference.

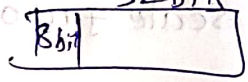
8-bit	16-bit	32-bit	
A	AX	EAX	gen. pur. regis.
B	BX	EBX	
C	CX	⋮	
D	DX	⋮	
E	⋮	⋮	
F	⋮	⋮	
G	⋮	⋮	
H	⋮	⋮	



MOVE AL, 10

↳ moves the src (10) to the lower 8-bits

AH → moves the content to the higher 8-bits



OFFSET

```
MOV EAX, OFFSET LOC
MOV EBX, [EAX]
```

LEA → Lowered Effective Address → for dynamic

```
LEA EAX, [EBX+50]
```

JZ LOOP

↳ Jump to location loop if the value of branch is '0'

JG loop

↳ branch > 0

EDI - Index Register

Loop directing : LOOP ADD

M/c i/p format:

OPCODE — Addressing mode → Registers used — displacement → & to 32 bits — Immediate

OPCODE	Addressing mode	displacement	Immediate
1/2 bytes	1/2 bytes	1/4 bytes	1 or 4 bytes

If we use only 'l' loc., then (1 reg) 1 byte is suff.

Eg: MOV EAX, EBX → 1oc

2 bytes → 2 registers → Eg: ADD EAX, [EBX + EDI * 4]

loc

MOV EAX, 820 → 32 bits → 4 bytes
↓
1 byte

~~MOV EBX,~~
MOV AL, 8 → 8 bits → 1 byte
↓
1 byte

push
We can push/pop memory locations into the stack (data in it gets stored)

```
LEA   EBX, NUM1
MOV   ECX, N
MOV   EAX, 0
MOV   EDI, 0
ADD   EAX, [EBX+EDI*4]
      INC EDI
      DEC ECX
JG    ADD
MOV   SUM, EAX
```

n n-1 n-2 . . . 0

(>0)

```
LEA EBX, NUM1
MOV ECX, N
MOV EAX, 0
SUB EBX, 4
ADD EAX, [EBX+ECX*4]
LOOP ADD
MOV SUM, EAX
```

data: (data section)

```
DD NUM1   (15, 7, 6, 3, -21)
DD N      (15)
DD SUM    (0)
```

MAIN: → i/p that r translated to M/c Lang.

END MAIN

SF, ZF, OF, CF → conditional flags.

- Status flag
- Zero flag
- overflow flag → arithmetic overflow
- carry flag

in sub:
CF = 1 → borrow signal