ALU

General Purpose Register (GPR)

Special Registers → Memory Address Register (MAR)
→ Memory Data Register (MDR)

MAR → is used to read / hold the (address) *

MDR → is used to read / hold / write the (data) *

Operation Operand 1, Operand 2 .
Instrⁿ ~~Dest~~ SRC₁ / SRC₂ ~~Dest~~
Destnⁿ

→ Sum
b = 5 a = 3

eg → Add Loc A, R₀

Steps ⊢

① Get operand's value into ALU

a) Register → ALU

b) MAR → MDR → ALU
(perform read opⁿ)

① Register
② Memory locⁿ
(Loc A) eg.
2000

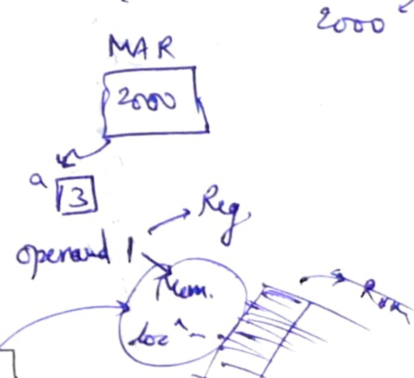
② Perform the operⁿ.

③ Store the result

a) Reg. ALU → Register

b) Memory ALU → MDR (WB) MAR

write back operⁿ
to delete prev. data & write new one.



Example:

Add LocA, R_0

→ Instructions:

PC → 2000 → Program counter.

Special purpose register (SPR)

2000 | Add LocA, R_0

PC (2000)

↓

MAR

↓ (read operation) → Read

MDR [2000]

MAR | 2000

" content inside address 2000 ".

which is,

Add LocA, R_0.

Read ↓ MDR → ALU

MAR ← LocA

ALU ← R_0

← IR (instruction register)

↓

ALU

↓

MDR

↓ WB.

MAR (LocA).

↓

PC = PC + 1

Ex - Add R_0, R_1

PC | 2000 → Add R_0, R_1

PC (2000)

↓

MAR

↓ Read

MDR ↓ [2000] → (Add R_0, R_1)

ALU ← R_0

ALU ← R_1

← IR

↓

ALU

↓

MDR

↓ WB

MAR

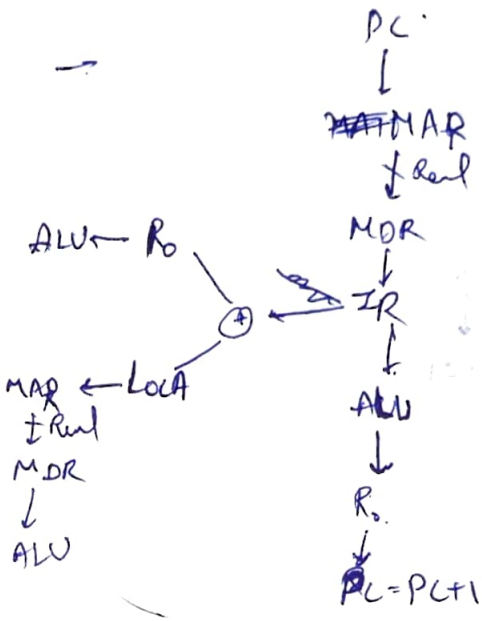
↓

PC = PC + 1

Ex - Add R_0, LocA

PC
2000

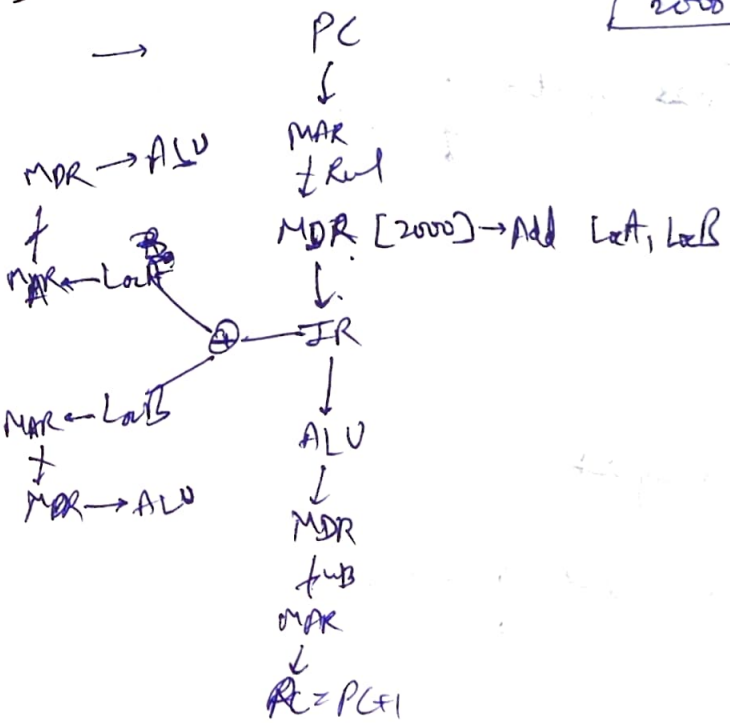
→



Ex - Add LocA, LocB

PC
2000

→



Number System Representation

+ve → 0
-ve → 1

→ Sign and magnitude

+5 → 0101
-5 → 1101

→ 1's & 2's compliment

eg → -3 → +ve no (+3)

0 011

1 100 → 1's compliment

+ 1

1101 → 2's compliment of (-3)

eg → -5 → +ve no. (+5)

0101

1010 1's compliment

+ 1

1011 → 2's compliment of (-5)

✡ Arithmetic Operations:

Ex: Add +7 and -3.

```
 0111   (+7
+1101    -3
------  ----
10100    +4
```

→ ignore the carry signal.

↓ Carry bit

(*) If answer lies within the range, then accept it, otherwise Cond^{n} of overflow will occur.

Rules is 2's complement Arithmetic.

① Addition

Ignore carry bit signal if the answer lies within this range. -2^{n-1} to $2^{n-1}-1$

then we accept the answer

range → Accept

else → overflow.

If $n = 3$,

no of bits.

-2^{2} to $2^{2}-1$

-4 to 3

② Subtraction.

Add $X + (-Y)$

↓

1's complement

↓

2's complement.

-2^{n-1} to $+2^{n-1}-1$

$-5 \Rightarrow +5$

0101

1's C 1010

+1

$-5 =$ 1011

Add +2 and +3

0010

+0011

+5 → 0101

for n, range → -2^{n-1} to $+2^{n-1}-1$

for n=4, -2^{4-1} to $+2^{4-1}-1$

-2^3 to 2^3-1

-8 to 7

Q Add +4 and −6

0100

+0110

1010

1's C 0101

+1

0110

1001

+1

1010

0100

1010

1110

1's C 0001

+1

2's C 0010

Q Sub −3, −7

~~1011~~

~~+1111~~

~~1010~~

$-3 \Rightarrow +3$

0011

1's C 1100

+1

−3 → 1101 ← 2's C.

$-7 \Rightarrow +7$

0111

1's C 1000

+1

2's C 1001

−7

0110

+1

0111

∴ 1101

+1001

0110 → 6

~~1001~~

1101

+0111

0100 → +4

Q] Sub +2, -4

~~0110~~ ← -4 ⇒ +4

0100

1's C → ~~1~~011

+1

2's C 1100 ← ~~-~~4

∴ 0010

+ 0100

0110 → (+6)

~~0001~~

~~++~~

~~0010~~.

2's C → 0011

+1

0100

Q] Add +2, -7

↳ -7 ⇒ +7

0111

~~1000~~

1000

+1

-7 → 1001

0010

+1001

1011

↳ 0100

+1

0101 ← -5

Dusky Arithmetic Op^n,

(*) When one of the operand is negative then computed answer should follow 2's C method for verifi^n.

Q] Add -3, -7

↳ -3 ⇒ +3

0011

↳ 1100

+1

(-3) 1101

-7 ⇒ +7

0111

1000

+1

(-7) 1001

1101

+1001

(-) 0110 ← (+6)

Carry Signal

∴ -3-7 ⇒ (-10) → out of range of -8 to 7

∴ Overflow Condition

Q) Add 7, 3.

=>

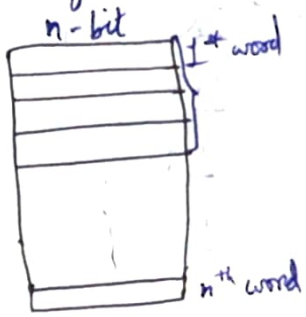
$$\begin{array}{r} 0111 \\ +\ 0011 \\ \hline 1010 \end{array} \leftarrow 10$$

* When two numbers having same sign, 2's C repr" system might give incorrect result.

Memory

→ trillions / billions of storage cells are used to store the bits

→ Access the bits in the form of word



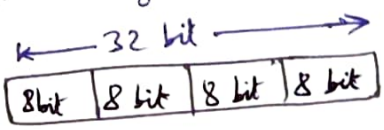
→ Data is usually accessed in n-bit group according to word length.

a) Sign and magnitude

+ve → $b_{31} = 0$

-ve → $b_{31} = 1$

Characters → each of 8 bit



ASCII

Byte Addressable memory. → each word = 4 Bytes.

WL = 0, 4, 8, ---

word length

K bit address = 2^K memory loc^n

32 bit address = 2^{32} mem. loc^n

$= 2^{30} \cdot 2^2$

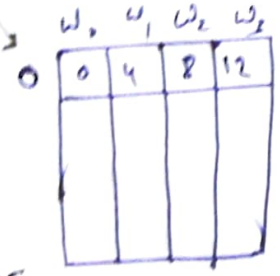
$= 4$ GB

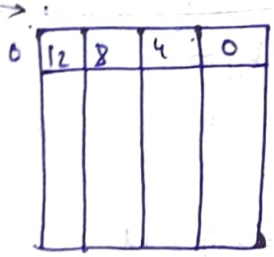
- $2^{10} = 1K$
- $2^{20} = 1M$
- $2^{30} = 1G$
- $2^{40} = 1T$

Memory Representation

→ Big Endian ⇒ Lower bit is used to denote MSB address.

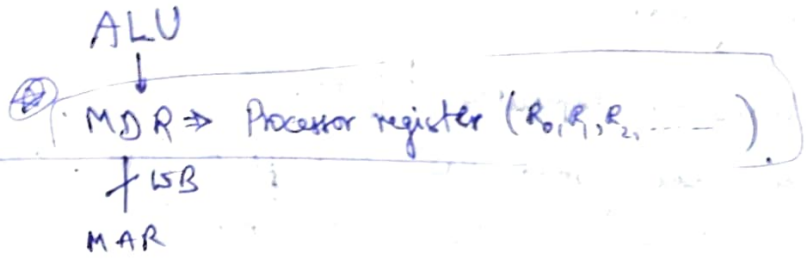
→ Little Endian ⇒ Lower bit is used to denote LSB address.





Memory Operation

- Load: transfer the contents of memory location to processor register.
  - eg ⇒ $R_1 \leftarrow M[A]$
  - → Load R_1, M[A]
  - Assembly language not^n.
- Store: transfers the content of the processor's registers into memory location.
  - eg ⇒ $M[A] \leftarrow [R_1]$
  - Store M[A], R_1



Instruction and Instruction Sequencing ..

i) Data transfer between memory location and register
ii) ALU operations on data → Instr" format
iii) Program sequencing and control → conditional codes
iv) I/O transfers → Branching instructions.

- 1.1 Register Transfer Not"
- 1.2 Assembly lang. Not".

1)

* Register Transfer Not"

→ contents inside the register are denoted by placing [ ] .

mem. loc" or register ← $M/R \leftarrow [R_1]$

* Assembly Lang. Not"

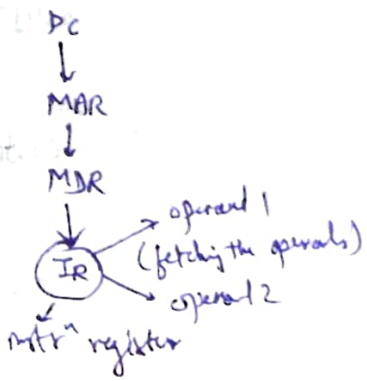
→ used to understand low-level language like machine instr"s.

Operation	Destination	Source
Opcode	Operand	Operand

2) ALU operation on Data

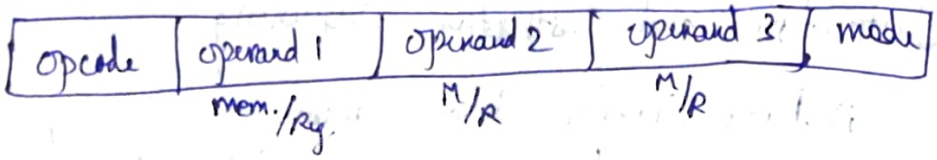
Instruction format :-

IR : Variable size.
↳ 3, 2, 1, 0 address instr".



3-Address Instr^n format

→ All addresses are explicitly defined.
Hen the instr^n format has 3 different fields, which specify either a memory or register loc^n.



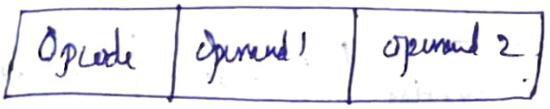
Ex:- Add R_1, A, B

$R_1 \leftarrow M[A] + M[B]$

Advantages:
1) Useful for shorter codes

Disadvantages:
1) no of instr^n reqd. to specify operand > # of Instr^n operand reqd. to decify opcode.
2) Takes up large space to specify the instr^n.

2-Address Instr^n format



Ex:- Add A, B

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

Advantages
① It is good for shorter code.

Disadvantages
① # of instr^n reqd. to specify operand > # instr^n reqd. to specify opcode.
② More no of bits are reqd.

1-Address Instr^n format

→ It uses implied accumulator register for data manipulation.

↓

CPU will use its own accumulator Reg., so there is no need to specify the address.

ex → Load A

Acc ← M[A]

ex → Add A,B → can be converted to 1-address instr^n.

① The more no of bits we reduce → more effort our CPU will be.

0-Address Instr^n format

→ No operand address.

→ Only opcode → Push / Pop

opcode

Stack Organization → push / pop

ex → Push (≡ Push Acc.)

→ Add

Push

Advantages →

1) very fast
2) Less no of bits are reqd.

Disadvantages →

① We need stack organ^n, which is compulsory. (Queue, linked lists, etc can't be used)

② Bcoz it's used stack org^n, it is necessary to convert into post-fixed ~~oper^n~~ format

↳ refer back

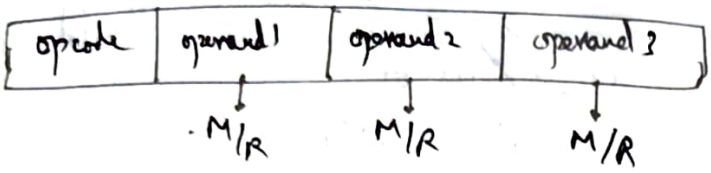
eg → ~~3x+1~~ → x + 3, 2, 1

② we need to convert all the instr into post fixed format

① Evaluate the expression using 3, 2, 1, 0 instr format.

$(A+B) * (C+D)$

Sol" ⇒ * 3-Addr. instr



Add R_0, A, B

Add R_1, C, D

MUL ~~ADD~~ R_2, R_0, R_1

By means Instr.

Add $R_1, M[A], M[B]$ ← $R_1 \leftarrow M[A] + M[B]$

Add $R_2, M[C], M[D]$ ← $R_2 \leftarrow M[C] + M[D]$

Mul R_3, R_1, R_2 ← $R_3 \leftarrow R_1 * R_2$

Notations

$R_1 \leftarrow M[A] + M[B]$

~~$M[A] \leftarrow M[A] + M[B]$~~

Add A, B

not allowed in 3-addr.

② Add Z, A, B

it requires 2 clock cycles whereas using R_1 requires only 1 clock cycle

we will choose, which one takes less clock cycle for getting result

Eg: Add A, B ← takes 6 clock cycle

* 2-Addr. Instr format

Instr

Add M[A], M[B]

Add M[C], M[D]

Mul M[A], M[C]

Notations

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

$M[C] \leftarrow M[C] + M[D]$

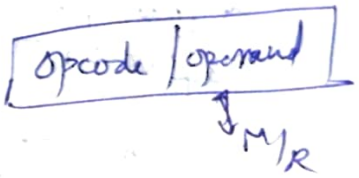
$M[A] \leftarrow M[A] * M[C]$

* 2 Addr. Instⁿ format

Instⁿ	Notations
Load R_1, A	$R_1 \leftarrow M[A]$
Add R_1, B	$R \leftarrow R_1 + M[B]$
Load R_2, C	$R_2 \leftarrow M[C]$
Add R_2, D	$R_2 \leftarrow [R_2] + M[D]$
~~MOV X, R~~	$M[X] \leftarrow [R_2]$
MUL R_1, R_2	$(R_1) \leftarrow [R_1] \times [R_2]$
MOV X, R_1	$(X) \leftarrow [R_1]$

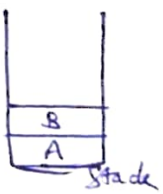
* 1 Addr. Instr"

Load A	$Acc \leftarrow M[A]$
Add B	$Acc \leftarrow [Acc] + M[B]$
MOV X	$M[X] \leftarrow [Acc]$
Load G.	$Acc \leftarrow M[C]$
Add D	$Acc \leftarrow [Acc] + M[D]$
MUL X	$M[X] \leftarrow [Acc] * M[X]$



* 0 Addr. Instr"

Push A	$TOS \leftarrow M[A]$
Push B	$TOS \leftarrow M[B]$
Add	$TOS \leftarrow (A+B)$.
Push C	$TOS \leftarrow M[C]$
Push D	$TOS \leftarrow M[D]$
Add	$TOS \leftarrow C+D$
Mul	$TOS \leftarrow (A+B) * (C+D)$



④ In stack organiz", push & pop operations are performed by-default operation along w/ address, however in 0-addres instr" Add SUB MUL will not carry any address

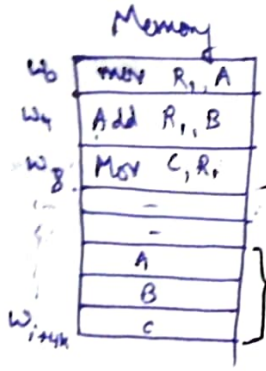
☆ Instruction Sequencing

Add C, A, B

C = A + B ←

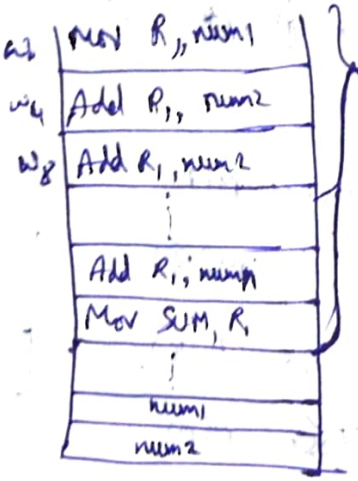
- mov R1, A
- Add R1, B
- mov C, R1

R1 ← M[A]
R1 ← [R1] + M[B]
M[C] ← [R1]



32-bit → 2^{32} M L.
Byte addressable memory
$\frac{3.2}{8} = 4B$
each word length is 4B

☆ What if, there are n numbers for sum.



for n=100,
we would need 100 times 4B' more loc^{n},
so that to accomodate all instr^{n} & also the data for the instr^{n}., so the sol^{n} for this is,

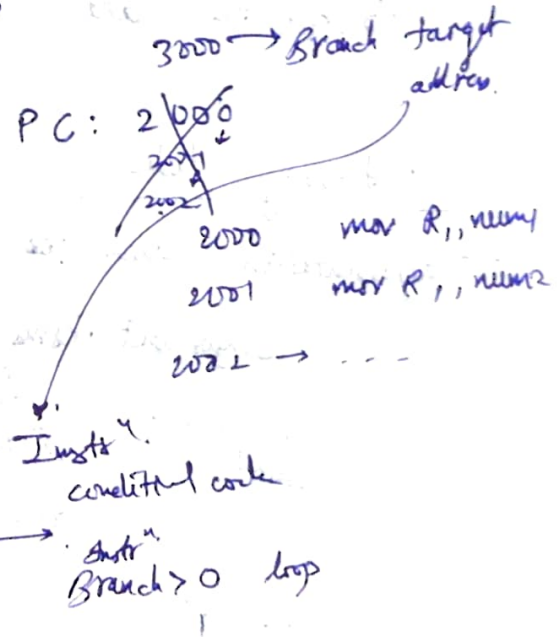
② Instruction Sequencing : Branch Instr^{n}.

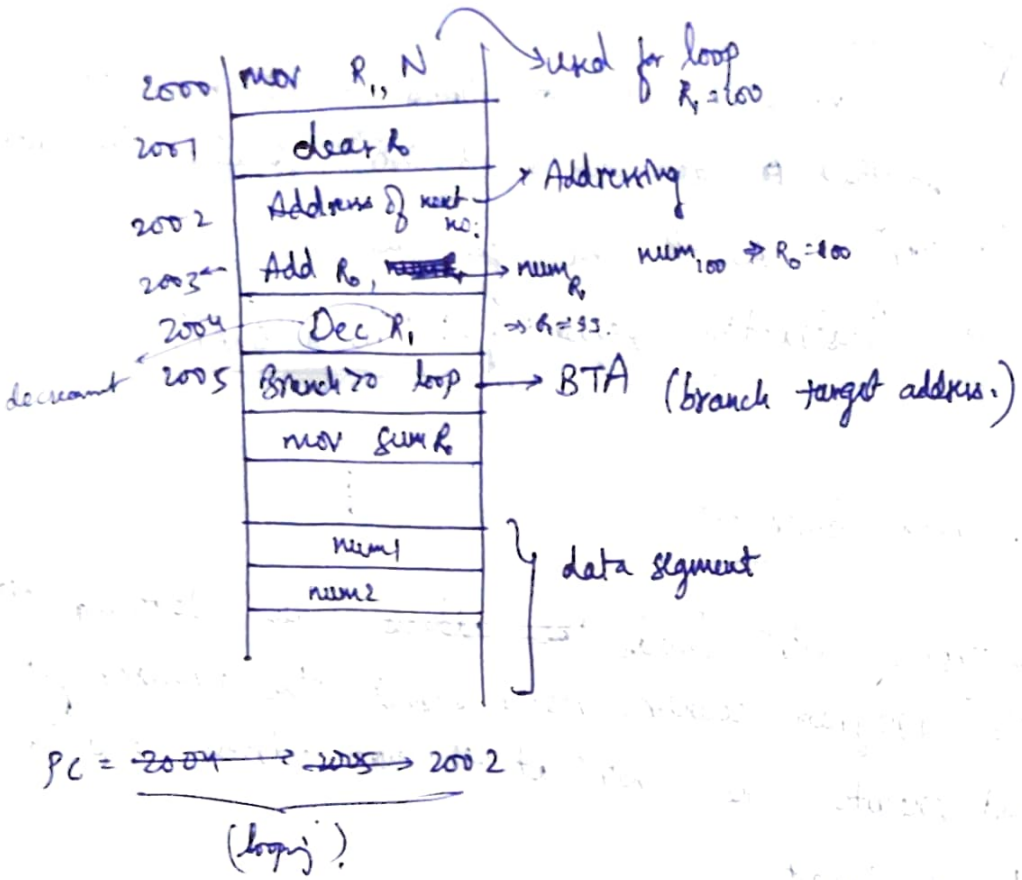
*. Branch Instruction

This type of instr^{n} loads ~~the address~~ a new address into the program counter. As a result the processor fetches and executes the instr^{n} at the new address called the branch target.

Conditional Code: A conditional branch instr^{n} causes a branch only if a specified cond^{n} is satisfied.

eg → Branch > 0 loop





This is the sol^n for reducing no of instr^n in memory,

but there is no sol^n for reducing memory reqd. to store the data.

★ Addressing Modes.

→ Representation in which loc^n of an operand is specified within instr^n.

Important terms:

- 8 bit/16 bit value.
- i) Displacement
- 2) Index Register (source index or destination index) → SI, DI
- 3) Base Register → BX, BP (Base pointer)

i, 2, 3 → EA (Effective address)

eg - EA = [R]

EA = [[R1]] ⇒ eg → R1 [R2] or R [2000]

R2 [3] 2000 [5]

V. Very Important ⭐⭐⭐ Numericals come from here

1) Implied AM
2) Immediate AM
3) Register direct AM
4) Register Indirect AM
5) Direct AM
6) Indirect AM
7) Relative AM
8) Index AM
9) Auto ~~increment~~ AM
   increament
10) Auto decreament AM
11) Base Register AM

1) Implied AM → operand is specified within, instr.
   ↳ data is stored as 8 bit / 16 bit displacement

CLC

0
carry flag reset to 0.

2) Immediate AM →

opcode	Data

Add 3.
Acc ← Acc + 3.

3) Register direct AM

EA = [R_i]

opcode	R_i

4) Register Indirect AM.

EA = [[R_i]]

Opcode	R_i

5) Direct AM

EA = [ML] (ML → memory location)

opcode	ML
mov	A

used to declare Variable in C programming eg → int a;

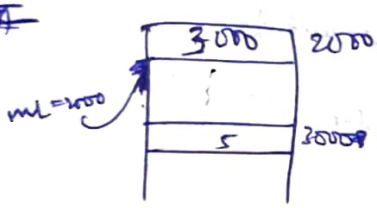
ACC ← m[A]

6) Indirect AM

↳ used to declare pointer in C programming

EA = [[ML]]

opcode	ML
mov	A



7) Relative AM

EA = [PC] + Address field of the instr^n.

PC ⇒ 2000

= 2000 + 3

= [2000+3]

EA = [2003]

opcode	Address (data) -3

→ Program Relocation

→ Calc^n of BTA.

PC ⇒ 2000

= 2008 + (-3)

EA = [2008 - 3]

= [2005]

8) Index AM

→ Index Register used to calc. EA

EA = ~~[PC] + [IR]~~ ~~Address~~ . [IR] + Address part of instr.

EA = [3000+2]

= [3004].

→ If R_{auto} contains the MA, then MR is required.

mem. addr. mem. reference.

9) Auto Increament AM.

→ we use general purpose register, which has functionality to ~~be~~ changes on its own. (R_{auto})

$R_{auto} = [R_{auto}] +$ step-size → depend on the size of operand.

→ used to perform loops.

10) Auto-Decreament AM

$R_{auto} = [R_{auto}] -$ step-size.

11) Base Register AM

EA = ~~[PC] + [BR]~~ [Address part + [BR]]

→ Array Relocation.

Q)

mov	m[500]	mode

AM	EA	Content
① Direct	m[500]	800
② Immediate	-	500
③ Indirect	[[500]]	800
④ Relative	[702]	325
⑤ Index	[600]	900
⑥ Register Direct	[R1]	400
⑦ " Indirect	[[R1]]	700
⑧ Auto Increment	[[R1]]	701
⑨ Auto Decrement	[[R1]]	699

memory reference required

1	200	Load to AC | Mode
	201	Address = 500
	202	next instr^n
	399	450
1	400	700
0	500	800
2	600	900
	702	325
1	800	300

I_1, I_2

given:
PC = 202
R_1 = 400
IR = 100
Step size = 1

PC = 200
(instr^n 1) is fetches 200 & 201
then PC → 202

② → no memory address req^d in Immediate,
it will treat operand as constant

③ → 2 mem. references req^d.
→ 500 → 800

④ $EA \rightarrow [202 + 500] = 702$.

202 → content of PC
500 → Address part of instr^n

500, not 800

(5) EA → [200] + 100 = [300]

[IR] + Address part of instr = 100 + 500 = 600

(6) EA = [R1]

& content ⇒ data of R1 → 400.

(7) EA = [[R1]] = [400]

write this as EA

content → data at 400 = 700

(8) [Rauto] + step-size = 400 + 1 = 401

[[Rauto]] + step-size ⇒ [400] + 1 = 700 + 1 = 701

(9) [[Rauto]] + step-size ⇒ 700 − 1 = 699

Q] Consider a 3 word machine instr. Identify # cycles needed during the exec of instr.

Add A[R0], @B

A[R0] → Indexed AM when R0 is IR

@B → Indirect AM

Add ⇒ first part A[R0]

EA = [[IR] + Address field of the instr]

EA = [[R0] + Address] → 1 memory reference is required.

2nd part

$@B = M[[B]]$ → indirect

↳ 2 memory references are required.

Add $A[R_0], @B$

- Add → opcode
- $A[R_0]$ (1) → SRC1 / Destn
- $@B$ (2) → SRC2
- $A[R_0], @B$ → operands

$A[R_0] \leftarrow A[R_0] + @B$

(1 ref. + 1 ref. + 2 ref.s)

∴ Total ref = 1+1+2
= 4 memory ref. are required.

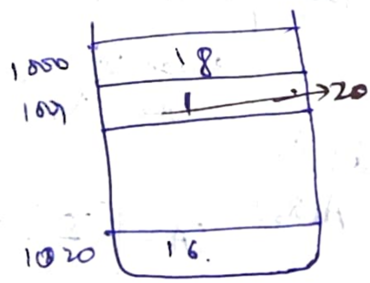
GATE PYQ

1) Memory loc's 1000, 1001, and 1020 have data values 18, 1, and 16. Consider the following instrn's:

MOV I	R_s, 1	Move Immediate
Load	R_d, 1000(R_s)	Load from memory [AM → Index addressing]
ADD I	R_d, 1000	Add Immediate.
Store I	0(R_d), 20	Store Immediate

- I → Immediate addressing mode
- @A, @B → indirect addressing mode
- $A[R_0]$ → index addressing mode ($A(R_0)$)

What will be the value in locn 1001 after execn?



(1) $R_s \leftarrow 1$; R_s = [1]

(2) $R_d \leftarrow [1000 + [R_s]]$ (content of)

$R_d \leftarrow [1000 + 1] \rightarrow [1001]$ (EA → content of 1001.)

∴ $R_d \leftarrow [1001]$

$R_d \leftarrow 1$; R_d = [1]

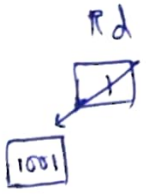
(3)

$R_d \leftarrow [R_d] + 1000$

Here 1000 is not memory address, ∴ Immediate addressing mode.
∴ 1000 is constant.

$R_d \leftarrow 1 + 1000$

$R_d \leftarrow 1001$



(4) $[0 + [R_d]] \leftarrow 20$

$[0 + 1001] \leftarrow 20$

Q] Consider the following memory values & one address machine organization. Calc. the ~~Fig~~ output of each instr".

20	40
30	50
40	60
50	70

Instr"
Load I 20
Load D 20
Load @20.
Load I 30
Load D 30
Load @30.

O/p

20 Acc ← 20
40 Acc ← [30] → 40
60 Acc ← [[30]] : [40]
30 Acc ← 30.
50 Acc ← [30] → 50
70 Acc ← [[30]] : [50]

Acc ← 20

Ans:
① ⇒ Acc ← 20
② ⇒ Acc ← [20]
Acc ← 40
③ ⇒ Acc ← [[20]]
Acc ← [40]
Acc ← 60.
④ ⇒ Acc ← 30
⑤ ⇒ Acc ← [30]
Acc ← 50.
⑥ ⇒ Acc ← [[30]]
Acc ← [50]
Acc ← 70.

Q) Consider a RISC machine where each instrⁿ is 4 bytes long (exactly). Here, conditional branch instrⁿ w/ PC relative AM is used. Offset is specified in bytes for the calcⁿ of target locⁿ of the branch instrⁿ. Consider the following sequence of instrⁿ:

Instrⁿ no.	Instrⁿ
i	Add R_2, R_2, R_4
i+1	Sub R_5, R_6, R_7
i+2	Cmp R_1, R_9, R_{10}
i+3	beq R_1, offset

(beq → branch on equal to)

If the target branch eqⁿ is i, find the value of offset.

→ BTA : i

Q is asking to calc' the offset (jumping value)

EA = [PC] + offset

no need of [ ]

∴ no computⁿ is needed → so no content of EA is needed.

∴ each instrⁿ is 4 bytes long (say)

100	i
104	i+1
108	i+2
112	i+3
116	i+4
	i+...

(1). Direct AM.

PC = 100 (say)

EA = [100].

EA = [PC] = 100

* PC value will hold the next instrⁿ address, while executing the current instrⁿ so that there is no lag in CPU.

EA = [PA] + offset

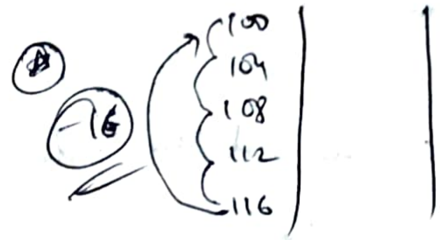
100 = 116 + offset

offset = 100 − 116 = (−16)

So it is like after exec of i+3 instrⁿ (PC will be holding 116 while executing it), we need to go back to iᵗʰ instrⁿ (100)

PC (→ instrⁿ address)

so offset = −16 → it needs to go back 16 bytes.



* Expanded Opcode Technique

32 bit

opcode	A_1	A_2	A_3
11 bits	a	b	c

3-address instr^n.

opcode	A_2	A_3

Opcode	A_3

\# of 3 AI = x

\# of 2 AI = y

\# of 1 AI = z

\# 2 AI possible $= (2^{11} - x)\,2^{a}$

\# 1 AI possible $= ((2^{11} - x)2^{a} - y)\,2^{b}$

\# 0 AI " $= (((2^{11} - x)2^{a} - y)2^{b} - z)\,2^{c}$

Q] In an 11-bit instr^n format, the size of address bit is 4-bits. The computer uses expanding opcode technique & contains 5 (2)-address instr^n & 32 1-address instr^{n}s. Calc. the no. of zero address instr^n it can support.

If the types of Qs, no. of Address instr^n = highest no. given.

⇒

11 bit

opcode	A_1	A_2
3 bits	4 bits	4 bits

5 – 2 AI

32 – 1 AI

? – 0 AI

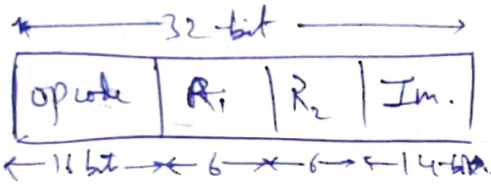
1 AI possible from given instr^n format $= (2^3 - 5)\cdot 2^4 = 48$.

0 AI " " " " $= (48 - 32)\cdot 2^4$

$= 16 \times 16 = 256$

* Gate exam Qs based on this topic.

Q] A machine has 32-bit Architecture of one word long instr. It has 64 registers. Each of which 32-bits long. It needs to support 45 instr. And immediate operands is also there along of 2 register operands. Assuming that immediate operand is an unsigned integer. Then find out the max value of the immediate operand.



$0 \to 2^n - 1$

000 → +
001 → ×
010 → −

$= 32 - (6-6-6)$

$= 32 - 18$

$= 14$

$\log_2 45 = 6$ (rounded off)

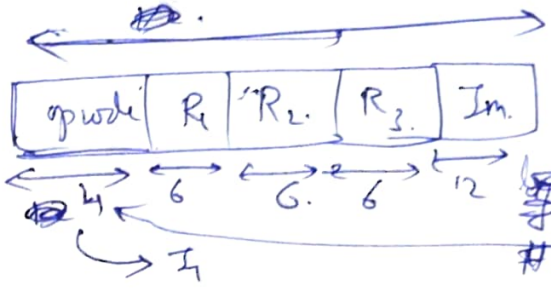
$= 2^{14} - 1$

$= 16383$

bits $R_1 = R_2 = \log_2 64 = 6$.

Q] Consider a processor of 64 registers & instr set of size = 12. Each instr has opcode, 2 src. registers, 1 destination register & 12 bit immediate value. Each instr is stored by aligned fashion. If a program has 100 instr, the amount of memory consumed by the program in bytes are ____?

Soln ⇒

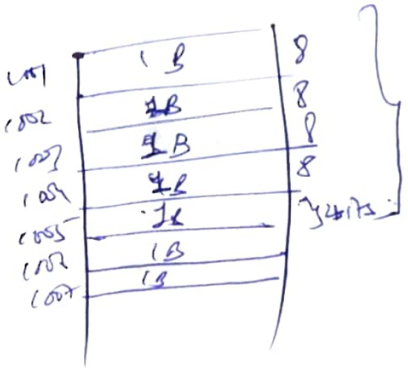


bits of $R_1 = R_2 = R_3 = \log_2 64 = 6$.

bits opcode $= \log_2 12 \approx 4$

$= 4 + 6 + 6 + 6 + 12$

$= 34$



$8+8+8+8+2 = 34$ bits.

$= 5$ blocks $= 5$ bytes.

1 instr $\longrightarrow$ 5 Bytes.

100 instr $\longrightarrow 5 \times 100 = 500$ bytes

If asked in bits then

$= 34 \times 100 \times 8$ bits.

9) A CPU is designed to have 58 ? AI, the CPU is able to address a max of 60 many bytes. The length of machine code is 32 bits for all the instr's. Determine the no of bits allocated for each address field.

(1) Determine the max no of bits allocated for the opcode field of 2 address instr'n.

(2) Determine the max no of bits allocated for the opcode of 2 address instr'n.

* Number Representation

- └→ sign and magnitude
- └→ IEEE floating point representation.

2's complement method.

-2^{n-1} to $+2^{n-1}-1$

→ overflow condition

$n = 8$

-2^{8-1} to $+2^{8-1}-1$

-2^{7} to $2^{7}-1$

-128 to $+127$ (+128)

It allows large range of no. using less no. of bits.

sign bit → S → +ve / → -ve | exponent

no. of 0's power in the no.

Mantissa → actual no. in binary.

Standard format

S	E	M
sign	expo.	Mantissa.

Exponent → biased.

no. should start from 0.

* **Mantissa** always, normalised. eg:-
  - └→ (1) If no. is 0.001 then (Mantissa) ~~0.1×10~~ 0.1×2^{-2} → base is binary. (Right shift → -ve power)
  - (2) ~~100~~ 100.1 then 0.1001×2^{3} (Left shift → +ve power) → Mantissa (after pt.)

Mantissa → Implicit, Explicit

Range: -2^{5-1} to $+2^{5-1}-1$

-2^4 to $+2^4-1$ (0 to 31)

-16 to +15 → 32 bit unsigned format

$-16+16=0$

↳ bias.

	bias
-16	16
-15	15
-14	14
-13	13
.	
.	
.	
-1	1
0	0

bias is such that the ~~num~~ should lie in (0-31) i.e. bias is min. to make the no. 0.

* Excess 16 code → will work for all nos. (-16 to 0)

If k bits are allocated in E, then bias formula is 2^{k-1}.

∴ Bias $= 2^{k-1}$

for the above,

$k=5$

$\therefore 2^{5-1} = 2^4 = (16)$ ← bias

If $k=7$

$2^{7-1} = 64$

∴ ⇒ Excess 64 code.

Mantissa

Mantissa contains fraction part in binary but in normalized form.

101.11 → Explicit mode (by default) / Implicit mode.

In the explicit mode, after the pt., the 1st bit should be 1.

eg(i): $.1$

$\Rightarrow 0.10111 \times 2^{+3}$

$E = e + \text{bias}$; $M = 10111$; e ; $E = 3 + \text{bias}$

value

S	E	M

address

so E ≠ E — one is address, other is value.

0	3+bias	10111

Implicit mode:

After the pt., it can be either 0 or 1.

(i.e., $1.X$ → anything)

The 1st bit before pt. should be 1.

$\Rightarrow 1.0111 \times 2^{+2}$; e ; $M = 0111$

$E = 2 + \text{bias}$

0	2+bias	0111

Eg: $(19.25)_{10}$

$\Downarrow$

$(10011.01)_2$

$\longrightarrow$ $M: 0.10011101 \times 2^{+5}$; $e = +5$

$M = 10011101\,00$

S	E	M

16 bits.

1 | 6 | 9

(k (given) exponent bit) then

$\text{bias} \rightarrow 2^{k-1} = 2^{6-1} = 2^5 = 32$

$\therefore E = e + \text{bias} = 5 + 32 = (37)_{10}$

$\Downarrow$

32 16 8 4 2 1

$(1\,0\,0\,1\,0\,1)_2$

∵ Mantissa has 9 bits given, then we append 2 zeroes at last

→ we can do it, ∵ Mantissa is after pt., so it won't make any difference.

S	E	M
0	100101	100110100
1	6	9.

$(0100101\,1001101\,00.)_2$ to hex.

0.25 → 4

0.1875 → 3.

0.6875 → 11 → B

4

$(43B4)_H$.