:COA:-

#) Memory organization:

*) RAM

MM	Secondary M	RPM	SLAV
Physical	Logical M		
Primary	virtual M		
	Auxiliary M		

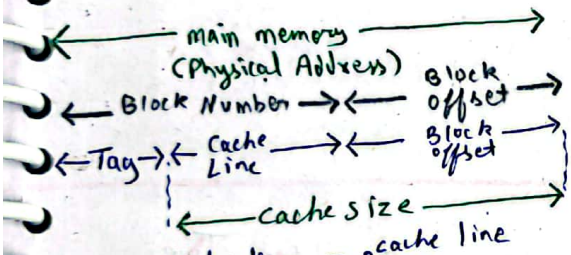
*) Lines (CM) Blocks (MM) Pages (SM)

*) Direct mapping
Fully Associative "
Set-Ass. "

*) 2^{10} — Kilo
2^{20} — Mega
2^{30} — Giga
2^{40} — Tera
2^{50} — Peta

*) Direct mapping:

Block no. / No. of cache line = Remainder is Address of block in cache

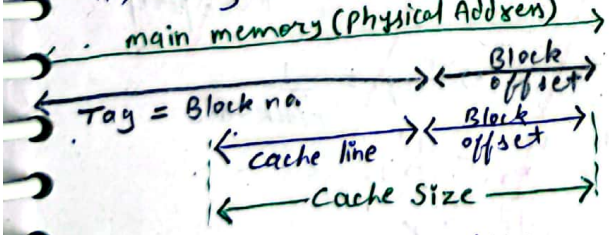


No. of cache lines = $2^{\text{cache line}}$

Tag dir size = No. of cache lines × Tag bits

Each cache Line has Tag

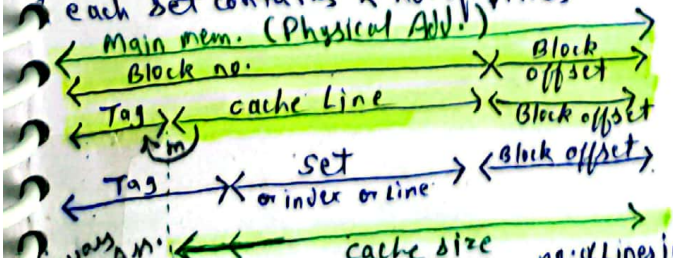
*) Fully Associative mapping:



Tag bits = comparator bits
No. of CL = no. of comparators
Hit Latency = Time taken by one comparator + Time taken by or gate
Each CL has comparators

*) Set Associative mapping:

2 way set ass. ⇒ C.Lines grouped into sets
each set contains 'K' no. of lines

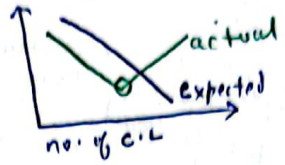


no. of sets = no. of Lines in cache / no. of CL in a set (K)

#) Cache replacement policies:

(only for full ass. & set-Ass.)

1. Fifo: Belady Anomaly



2. LRU
3. MRU
4. Optimal algo (Future ref. Unimplementable)

ex: 4 way set ass.
16 cache blocks
LRU

Request order: 0, 255, 1, 4

Procedure: Request order % 4 → Ber. wise 4 sets

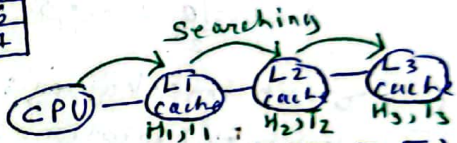
S-0	0-0, 1-4, 2-, 3-
S-1	4-1, 5-, 6-, 7-
S-2	8-, 9-, 10-, 11-
S-3	12-255, 13-, 14-, 15-

Ex: Direct mapped
8 cache blocks
Req. order: 3, 5, 2, 8, 0
Procedure: order % 8

0	8, 0
1	
2	2
3	3
4	
5	5
6	
7	

*) Memory Org.n:

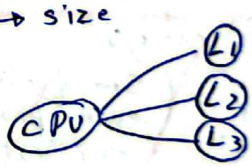
I) Hierarchical Access:



$EMAT = H_1T_1 + (1-H_1)H_2(T_1+T_2) + (1-H_1)(1-H_2)H_3(T_1+T_2+T_3)$

$\text{cost (Avg.)} = \frac{C_1S_1 + C_2S_2 + C_3S_3}{S_1+S_2+S_3}$ C → cost per byte, S → size

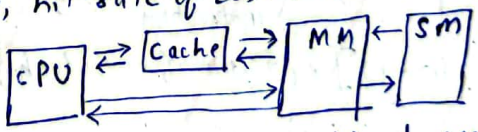
II) Simultaneous Access:



$EMAT = H_1T_1 + (1-H_1)H_2T_2 + (1-H_1)(1-H_2)H_3T_3$

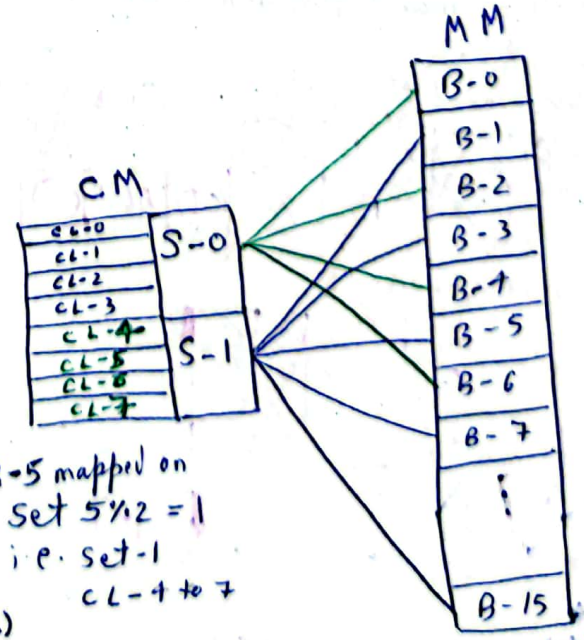
Here, hit rate of Last level is always 1 in I) & II) both

Note:)



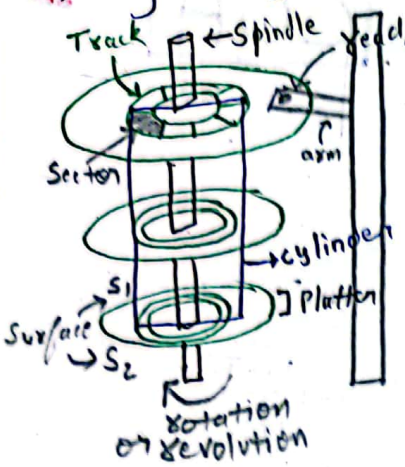
Note:) By default = Hierarchical access

Note:)



B-5 mapped on
Set 5%2 = 1
i.e. set-1
CL-4 to 7

Magnetic Disk:



Total transfer time = Seek time + Rotational Latency + Transfer Time

1. Surface no.
2. Track no.
3. Sector no.
4. Byte no.

Seek time = Time taken by R/W head to reach correct track.

Rotational latency = Time for 1 rotation/revolution / 2

Bits Transferred in 1 s = Transfer time = (File size / Track size) × Time for 1 rotation/revolution

1 Rotation = 1 track read/write

Ex:) Tracks = 500
Sectors = 100 (per track)
Bytes = 500 (per sector)
Seek time = 1 ms
RPM = 600 rpm
File size = 250 B

① TT = ST + RL + TT

RL: 600 r → 60 s
600/60 r → 1 s
1 r → 60/600 s
1 r = 100 ms
⇒ RL = 100/2 = 50 ms

② TT = (250 B / 100 × 500 B) × 100 ms
= 0.5 ms

Note: c, h, s (c: cylinder no., h: surface no., s: sector no.)

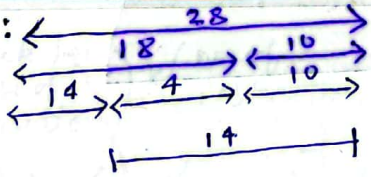
③ TT = 1 + 50 + 0.5 = 51.5 ms

② 1 r → 100 × 500 B → 100 ms
250 B → 100/(100×500) × 250
= 0.5 ms

Note:) Disk interleaving:
No interleaving ⇒ 1 rotation = 1 track read
Single interleaving ⇒ 2 " = " " "
Double " ⇒ 2.75 " = " " "

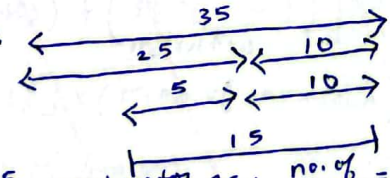
Direct mapping:

MM = 256 MB
cache = 16 KB
Block size = 1 KB
Tag = 14; Tag dir. = $2^4 \times 14$



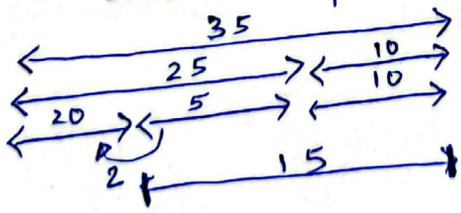
Fully Associative:

MM = 32 GB
Cache = 32 KB
Block size = 1 KB
Tag = 25; Tag dir. = 25×2^5; comparator bits = 25; no. of comparator = 2^5



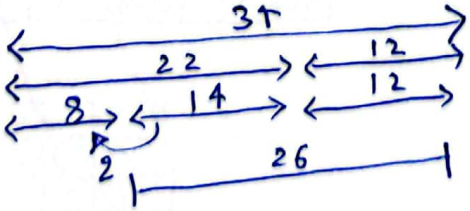
Set Associative

MM = 32 GB
Cache = 32 KB
Block size = 1 KB
4-way



Tag = 22
Tag dir. = 22×2^5

MM = 16 GB
Block size = 4 KB
Tag = 10
4-way
Tag dir = 10×2^{14}
cache = 64 MB



Q:) add = ⟨ 400, 16, 29 ⟩ ; Sec. no. = ?

Sec. no. = 400 × (10 × 2) × 63
+ 16 × 63
+ 29 = 505037

1 Cylinder → Platters = 10, Surfaces = 20

1 ~~track~~ Surface → 63 sectors

1000 cylinders

3

Q:) In above, if Sec. no. = 1,24,875 ; add = ?

Cyl. no. = 1,24,875 / 1 × 20 × 63 = 99.107 = 99

⇒ 124875 − 99 × 20 × 63 = 135 sectors बचे

Sur. no. = 135 / 1 × 63 = 2.14 = 2

⇒ 135 − 2 × 63 = 9 sectors बचे

Sec. no. = 9

add ⟨ 99, 2, 9 ⟩

File size = 42797 KB

⟨starting add⟩ = ⟨1200, 9, 10⟩

⟨ending add⟩ = ?

Sectors req. for file = File size / Sector size = 42797 × 1024 / 512 = 85594

1 cylinder → 8 Platters, 16 surfaces

1 ~~cylinder~~ Surface → 64 sectors

1 sector = 512 B

1638 4 cylinders

⟨1200, 9, 10⟩ ⇒ no. of sectors = 1200 × (16) × 64 + 9 × 64 + 10 = 1229416

1229416
+ 0085594
1,315,010

Cyl. no. 1315010 / 1 × (16) × 64 = 1284.18 = 1284

⇒ 1315010 − 1284 × (16) × 64 = 194 sectors बचे

Sur. no. 194 / 64 = 3.03 = 3

⇒ 194 − 3 × (64) = 2 sectors बचे

Sec. no. = 2

⟨ending add⟩ = ⟨1284, 3, 2⟩

#) Pipelining :-

*) Formulas —

$$T_{wp} = \left(\begin{matrix}\text{Sum of clock for}\\ \text{each phase of}\\ \text{one instruction}\end{matrix}\right) \times \left(\begin{matrix}\text{no. of}\\ \text{instructions}\end{matrix}\right) \times \left(\begin{matrix}\text{Time of}\\ \text{one clock}\end{matrix}\right)$$

$$T_p = \left[\left(\begin{matrix}\text{no. of}\\ \text{Phases}\end{matrix}\right) + \left(\begin{matrix}\text{no. of}\\ \text{Instr.}^{ns}\end{matrix} - 1\right)\right] \times \left(\begin{matrix}\text{Time of}\\ \text{one clock}\end{matrix}\right) = (m+n-1) \times t_c$$

Note:) For single instruction $T_p \geq T_{wp}$ (∵ pipelined वाले में Buffer delays होता)

$$\text{Speedup} = \frac{T_{wp}}{T_p}$$

$$\text{Max.}^m \text{ Speedup} = \text{No. of stages}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Max.}^m \text{ Speedup}} \times 100$$

Note:) when CPI = 1 ; Speedup = no. of Stages

*) Example :-

1 clk = 1 s

Instructions (phases →):

	F	D	E	WB	clock requirement
I_1	1	2	1	1	→ 5
I_2	1	2	2	1	→ 6
I_3	2	1	3	2	→ 8
I_4	1	3	2	1	→ 7
I_5	1	2	1	2	→ 6

$T_{wp} = (5+6+8+7+6) \times 1$
$= 32$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
I_1	F	D	D	E	WB														
I_2		F	X	D	D	E	E	WB											
I_3			F	F	X	D	X	E	E	E	WB	WB							
I_4				X	F	X	D	D	D	X	E	E	WB						
I_5					X	F	X	X	X	D	D	X	E	WB	WB				

$T_p = 15$

$\text{speedup} = \frac{32}{15} = 2.13$

$\text{max.}^m \text{ speedup} = 4$

$\text{efficiency} = \frac{2.13}{4} \times 100 = 53.25\%$

*) Example (Stage latencies) :-

Stages = 4 ; Instructions = 1000

5 ns	S_1 150 ns	5 ns	S_2 120 ns	5 ns	S_3 160 ns	5 ns	S_4 140 ns
155 ns		125		165		145	

: constant clocking rate
(or) Steady state under ideal cond.^{n}
(or) Large no. of instr.^{ns}
(or) synchronous pipeline
⇒ CPI = 1

i) Freq. (const.) = ?

at 1 clk = 165 ns , CPI = 1

$$\text{freq.} = \frac{1}{T} = \frac{1}{165} \text{ GHz} = 6.06 \times 10^{-3} \text{ GHz} = 6.06 \text{ MHz}$$

~~ii) $T_{wp} = 150 + 120 + 160 + 140 = 570$ ns~~
~~$T_p = [m + (n-1)] \times 165$~~

ii) $T_{wp} = (150 + 120 + 160 + 140) \times 1000 \times 1 = 570 \times 1000 = 570{,}000$ ns

iii) $T_p = [m + (n-1)] \times 165 = [4 + (1000-1)] \times 165 = 1003 \times 165 = 165{,}495$ ns

Note:) $\text{Throughput increase} = \frac{\text{final freq.} - \text{initial freq.}}{\text{initial freq.}}$

$= \quad " \quad \times 100\ \%$

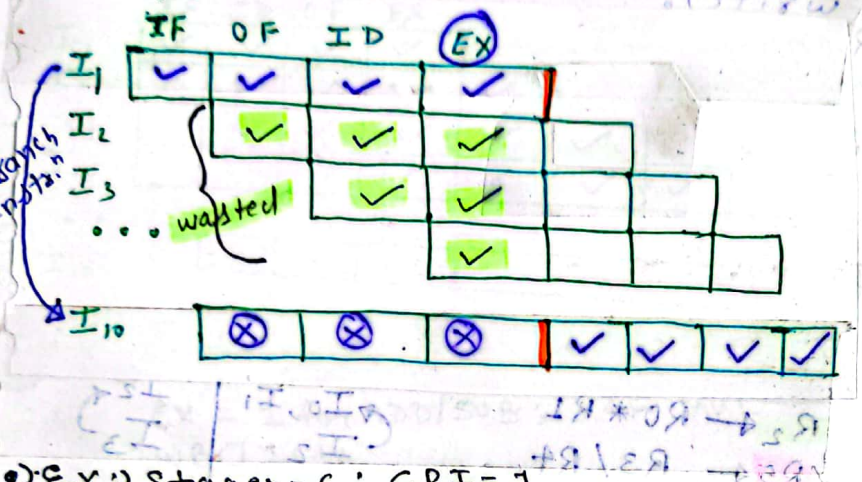
(*) Hazards
1. Structural hazard
2. Control ,,
3. Data ,,

•) Structural hazard :-

	1	2	3	4	5	6	7
	IF	OL	ALU	OS			
	⊗	IF	⊗	OL	ALU	OS	

IF = instr.n Fetch
OL = operand load
OS = ,, store
→ Those who works on memory but we only have single memory
So, 2 stalls ⊗ occured

•) Control hazard :-



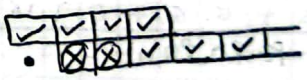
I_1 के Execute phase के बाद पता चला के I_1 के बाद I_{10} आएगा
So, 3 stalls ⊗ occured

•) Ex :) Stages = 6 ; CPI = 1
Non-Branch instr.ns = 75 %
Branch instr.ns = 25 %
Branch instr.n incur 2 stalls

$T_{wp} = 6 \times 1 = 6$

$T_p = 0.75 \times 1 + 0.25(1+2)$
$= 1.5$

1 तो generally आता ही है
2 Stalls Branch instr.n की वजह से लगते



•) Ex :)

IF	ID	EX	MEM	WB
1 ns	2.2 ns (Max)	2 ns	1 ns	0.75 ns

CPI = 1

Non-Branch instr.n = 80 %
Branch instr.n = 20 %
Total instr.n executed = 100

Calc.n of Stalls

IF	ID	EX	MEM	WB
	⊗	⊗	IF	

⇒ 2 stalls

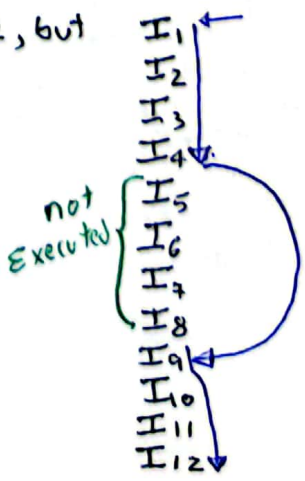
Shortcut : Ex से पहले 2 phases है
⇒ 2 Stalls

Execution time $= [0.8 \times 1 + 0.2 \times (1+2)] \times 2.2 \times 100 = 308$ ns

•) Ex :)

IF		ID		OF		EI		WO	
5 ns	1 ns	7 ns	1 ns	10 ns	1 ns	8 ns	1 ns	6 ns	1 ns
6 ns		8		11 (max)		9		7	

Total instr.ns = 12, but



Stalls = 3 ; Total instr.ns executed = 8

Execution time, without stalls $= [m + (n-1) + \text{Stalls}] \times 11$
$= [5 + (8-1) + 3] \times 11$
$= 165$ ns

• **Data Hazard** :- Instr.ⁿs exhibiting data dependency modifies data in different stages of pipeline. (3 types)

1. RAW (Read after write): Let, J follows instr.ⁿ I
   - I: $R_2 \leftarrow R_1 + R_3$
   - J: $R_4 \leftarrow R_2 + R_3$ — J trying to read R_2 before I written it

2. WAR (write after read): anti-dependence
   - I: $R_2 \leftarrow R_1 + R_3$
   - J: $R_3 \leftarrow R_4 + R_5$ — J trying to write R_3 before I read it.

3. WAW (write after write):
   - I: $R_2 \leftarrow R_1 + R_3$
   - J: $R_2 \leftarrow R_4 + R_5$ — J trying to write R_2 before I write it

Sol.ⁿ is operand forwarding

occurs during out of order execution instructions

• **Ex-1:)** Stages = 5 | operand forwarding used

IF, ID, OF] 1 clk

Perform op.ⁿ → PO = 1 for ADD & SUB ; 3 for MUL ; 6 for DIV

WO] 1 clk

Instructions:

I_0: MUL R2, R0, R1 — $R_2 \leftarrow R0 * R1$

I_1: DIV R5, R3, R4 — $R5 \leftarrow R3 / R4$

I_2: ADD R2, R5, R2 — $R2 \leftarrow R5 + R2$

I_3: SUB R5, R2, R6 — $R5 \leftarrow R2 + R6$

Dependencies:

I_0, I_1 | I_2

I_2 | I_3

[I_2 का OF नहीं होगा जबतक I_1, I_0 पूरा execute हो चुके

[I_3

[I_2

when done without operand forwarding:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
I_0	IF	ID	OF	PO	PO	PO	WO												
I_1		IF	ID	OF	X	X	PO	PO	PO	PO	PO	PO	WO						
I_2			IF	ID	X	X	X	X	X	X	X	X	X	OF	PO	WO			
I_3				IF	ID	X	X	X	X	X	X	X	X	X	X	X	OF	PO	WO

⇒ 19 clocks

when operand forwarding used:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I_0	IF	ID	OF	PO	PO	PO	WO										
I_1		IF	ID	OF	X	X	PO	PO	PO	PO	PO	PO	WO				
I_2			IF	ID	X	X	X	X	X	X	X	OF	PO	WO			
I_3				IF	ID	X	X	X	X	X	X	X	OF	PO	WO		

⇒ 15 clock

Ex-2:) Stages = 4 ; operand forwarding used from PO to OF

IF] 1 clk
OF]
PO = 1 for ADD/SUB ; 3 for MUL ; 5 for DIV
WB] 1 clk

I_0 : MUL R5, R0, R1 | R5 ← R0 * R1
I_1 : DIV R6, R2, R3 | R6 ← R2 / R3
I_2 : ADD R7, R5, R6 | R7 ← R5 + R6
I_3 : SUB R8, R7, R4 | R8 ← R7 - R4

I_0, I_1 → I_2 ; I_2 → I_3

	1	2	3	4	5	6	7	8	9	10	11	12	13
I_0	IF	OF	PO	PO	PO	WB							
I_1		IF	OF	X	X	PO	PO	PO	PO	PO	WB		
I_2			IF	X	X	X	X	X	X	OF	PO	WB	
I_3				IF	X	X	X	X	X	X	OF	PO	WB

⇒ 13 clocks

Ex-3:) Stages = 4 ; Operand forwarding is used

IF]
DF = (ID + OF)] 1 clk
EX = 1 for ADD/SUB ; 3 for MUL
WB] 1 clk

I_0 : ADD R2, R1, R0
I_1 : MUL R4, R3, R2
I_2 : SUB R6, R5, R4

I_0 → I_1 ; I_1 → I_2

	1	2	3	4	5	6	7	8
I_0	IF	DF	EX	WB				
I_1		IF	DF	EX	EX	EX	WB	
I_2			IF	X	X	DF	EX	WB

⇒ 8 clocks

#) Instruction format :-

•) 4 address instruction

Mode/opcode	Operand 1	Operand 2	Result	NextInstr.ⁿ
ADD	A1	A2	A3	0x233989

ex:

•) 3 address instruction : ProgramCounter stores nextInstr.ⁿ address

opcode	operand 1	Operand 2	Result

ex: X = (A+B) * (C+D)

ADD R1, A, B | R1 ← M[A] + M[B]
ADD R2, C, D | R2 ← M[c] + M[D]
MUL X, R1, R2 | M[X] ← R1 * R2

Note :)
Immediate operand = operand
$\lg_2$(Instr.ⁿ set length) = opcode
Instr.ⁿ format length = Length of instr.ⁿ
opcode = $\lg_2$ No. of instructions
operand = $\lg_2$ words

•) 2 address instruction : Operand 1 used to store result

opcode	operand 1	Operand 2

Note :)
Result is actually processor register

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

MOV R1, A	R1 ← M[A]
ADD R1, B	R1 ← R1 + M[B]
MOV R2, C	R2 ← M[C]
ADD R2, D	R2 ← R2 + M[D]
MUL R1, R2	R1 ← R1 * R2
MOV X, R1	M[X] ← R1

•) 1 address instruction : ACC contains result of all operations

opcode	operand 1

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

LOAD A	AC ← M[A]
ADD B	AC ← AC + M[B]
STORE T	M[T] ← AC
LOAD C	AC ← M[C]
ADD D	AC ← AC + M[D]
MUL T	AC ← AC * M[T]
STORE X	M[X] ← AC

•) 0 address instruction: STACK used

Ex: X = (A+B)*(C+D) — Infix

= AB+CD+* — Postfix

opcode

8

Ex: ADD i.e. pop 2 people from stack & add them then push

Push A	TOS ← A
Push B	TOS ← B
ADD	TOS ← (A+B)
Push C	TOS ← C
Push D	TOS ← D
ADD	TOS ← (C+D)
MUL	TOS ← (A+B)*(C+D)
Pop X	M[X] ← TOS

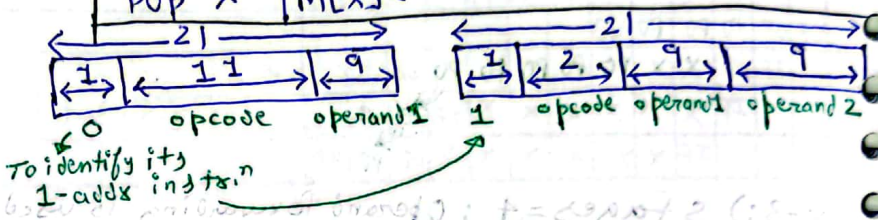
•) Examples :—

Q:) words = 512 = 2^9

no. of 1-addr instr.^{n} = 2048 = 2^{11}

—— 2 —— = 4 = 2^2

Length of instr.^{n} = ?



Q:) Registers = 64 = 2^6

Instr.^{n} set length = opcode = 12 ≃ 2^4

Immediate = operand = 12 b

opcode	R_1 Source	R_2 Source	R_3 Result	operand 1
4	6	6	6	12

100 such instr.^{n}

⇒ 34 b for 1 instr.^{n}

⇒ 34 b / 8 = 5 B per instr.^{n}

⇒ 500 B total

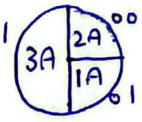
Q:) Instr.^{n} Length = 24 b

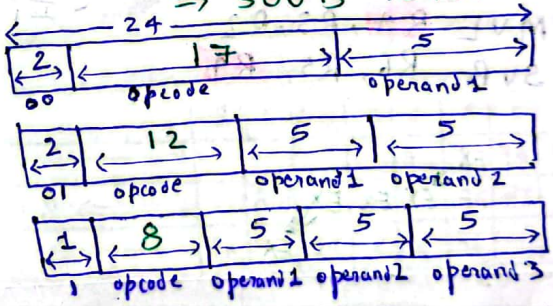
words = 32 = 2^5

50 % instr.^{n} = 3-Addr

25 % —— 2 ——

25 % —— 1 ——





Q:) Int Registers = 16 = 2^4

Float " = 64 = 2^6

Instr.^{n} length = 28 = 16 b

categories of instr.^{n} :—

Type-1: 4 instr.^{n}s = 2^2

each instr.^{n} = 3 INT Reg. operand

Type-2: 8 instr.^{n} = 2^3

" 2 float "

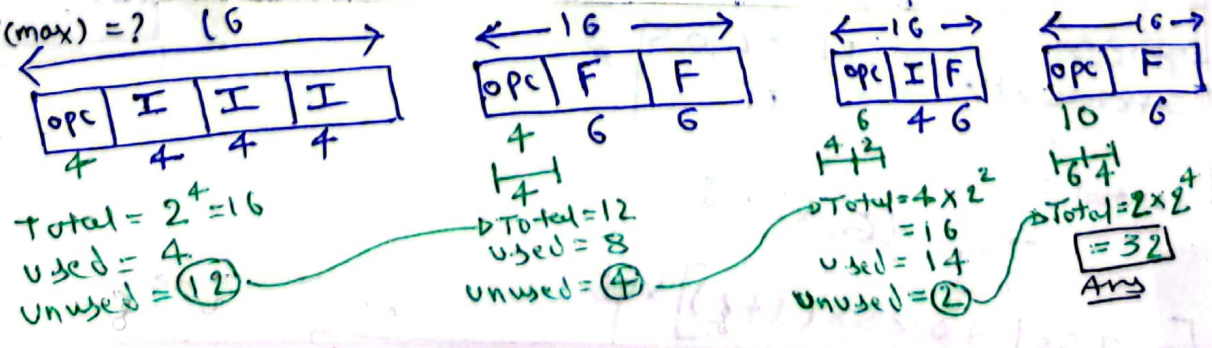
Type-3: 14 instr.^{n} ≃ 2^4

" 1 INT, 1 float "

Type-4: N - instr.^{n} = $\lg_2 N$

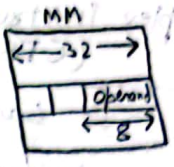
" 1 float

N(max) = ?

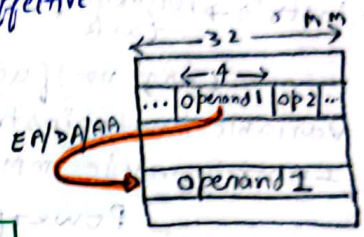


Addressing modes: - ways for reference to operand

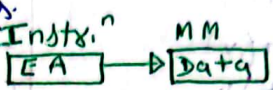
I: Immediate: - Instr.ⁿ itself have operand
- Used for constants, where value known.
- Large constants & constants whose value are unknown can't be used.



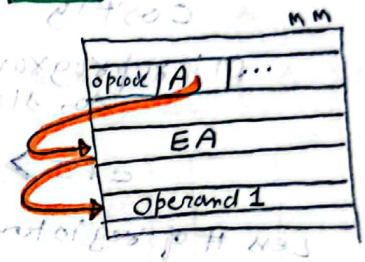
II: Direct (or Absolute): - Instr.ⁿ have address (Effective address/Direct address/Absolute address) of operand.
- Used for variables with unknown value, Large value variables, global variables.
- No. of variables used are limited
- Fails where large no. of variables.



III: Indirect: - Instr.ⁿ have address of EA of operand.

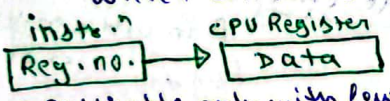


- NO Limit on no. of vars or size of vars
- Pointer is used

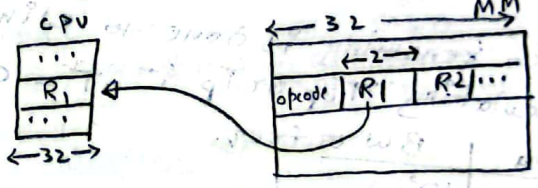


IV: Implied: - Address already provided, no need to provide explicitly.
Ex: 0-addr. ins.ⁿ (Top of stack)
Accumulator (ACC)

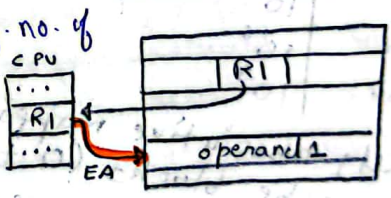
V: Register mode: - Instr.ⁿ have register no. which contains operand.



- Applicable only with few variables, bec. Reg.s are limited.

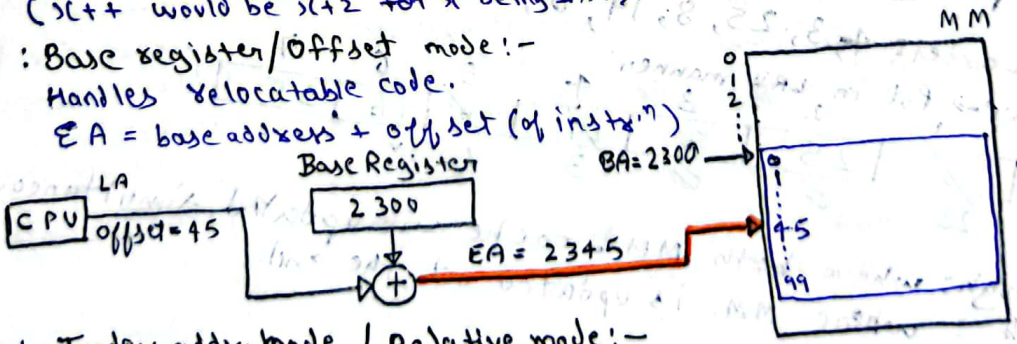


VI: Register indirect mode: - Instr.ⁿ have reg. no. of register which have EA of operand.
- Array implementation is done.

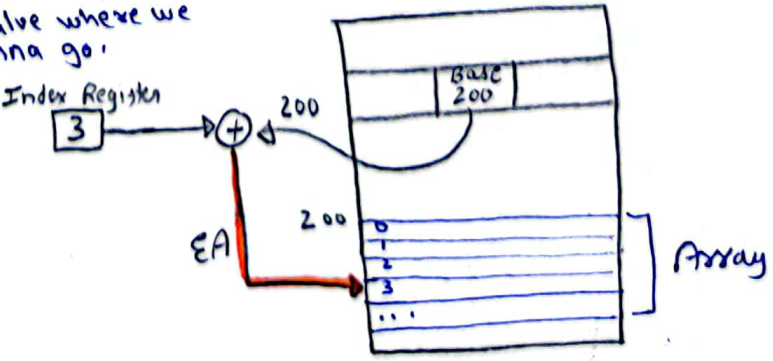


Auto inc./dec. register indirect: -
EA++ gives next element address
(x++ would be x+2 for x being INT)

VII: Base register/Offset mode: -
Handles relocatable code.
EA = base address + offset (of instr.ⁿ)



VIII: Index addr. mode / Relative mode: -
- Array is accessed.
- Instr.ⁿ have Base addr. of arrays, Index register have index value where we wanna go.
- Position independent code is implemented where no change in code needed.



*) Complex Instruction Set format (CISC)	Reduced instruction set format (RISC)
Large no. of instr.n	Few no. of instr.n
Few instr.ns → Perform specific task → used rarely	Few instr.ns → used very frequently
Large no. of addressing modes	Few addressing modes (3-4)
variable length instr.n format	Fixed length instr.n format
Instr.n inside MM	Loads instr.n in Register, then does work. (All operations done within register)
Powerful	Less powerful
costly	cheap
Microprogrammed so, slow	Hardwired. so, fast
CPI > 1	CPI = 1
Less # of registers	more # registers

Notes:

*) Register renaming is done to eliminate WAR & WAW data hazards

*) for calculating T_{wp} or T_p :- $T = CPI \times$ (cycle time $\rightarrow \frac{1}{f}$)

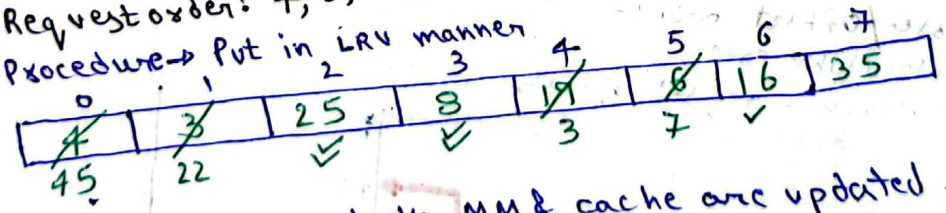
f in GHz or MHz

*)

Data		BW or freq.
K	2^{10}	10^3
M	2^{20}	10^6
G	2^{30}	10^9

*) Q:) • Fully associative mapping
• 8 cache blocks (0 to 7)
• LRU cache replacement policy used.
• Request order: 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

A:) Procedure → Put in LRU manner



*) write-through: where both MM & cache are updated simultaneously
write-back : where MM is updated at the end.

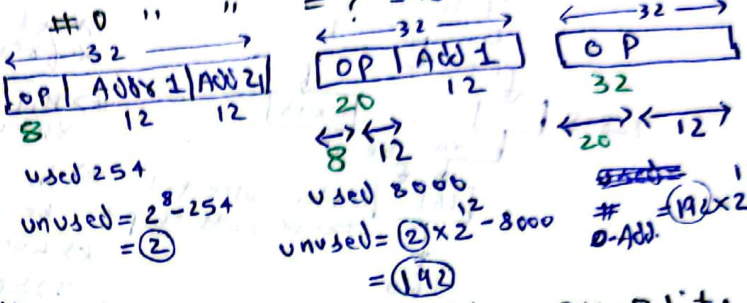
*) Special purpose registers:
1) AC : stores result of ALU
2) PC : address of next instr.n to be executed.
3) IR : • instruction register
• address of current instr.n being executed.
4) SP : • stack pointer
• address of TOS
5) MAR : • memory address register
• used to send address to memory
6) DR : • Data buffer register
• acts as buffer while accessing memory (R/W)

*) Microoperation: break one operation into small-small operations which CPU can execute.

*) Memory cycle time: Time to read/write 1 unit cell of memory. (i.e. 1 B memory) (or 1 word memory)

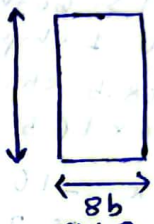
*) Throughput: work or task done in 1 unit of time. i.e. tasks/sec.
Ex: 1 task is completed in 5 ns
5 ns → 1 task
1 s → $\frac{1}{5n}$ task
⇒ Throughput = $\frac{1}{5n}$

*) CPU supports 2-Addr. & 1-Addr. & 0-Addr. instr.ns
Instruction size = 32 b
Address size = 12 b
2-Addr. instr.ns = 254
1 " " = 8000
0 " " = ? = x



used 254
unused = $2^8 - 254$ = (2)
used 8000
unused = (2) × $2^{12} - 8000$ = (192)
# 0-Add. = (192) × 2 12

*) Consider a memory of size 2K × 8 bits
Decoder size = ? to access memory.

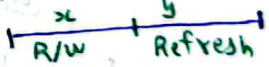


⇒ Decoder is 11 : 2^{11}

*) MM < RAM (volatile) < Static RAM, Dynamic RAM
ROM (non-volatile)

*) SRAM
made of Flip flops
No periodic refresh
costlier, faster
used for cache
R/W
efficiency ≈ 1
power cons.n ↑

DRAM
capacitors
requires periodic refresh
cheap, slower
used for MM

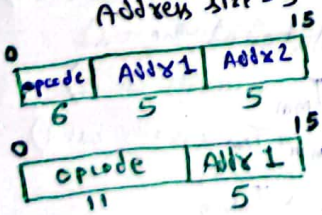


Eff.y < 1 ↓

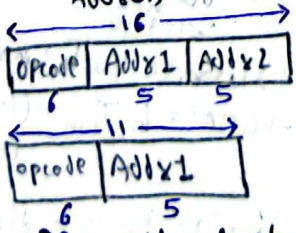
*)

A	B	A+B result	A×B result	Addn.n table size	Multn.n table size
4b	4b	5b	8b	$2^8 \times 5$ b	$2^8 \times 8$ b
nb	nb	n+1 b	2n b	$2^{2n} \times (n+1)$ b	$2^{2n} \times (2n)$ b

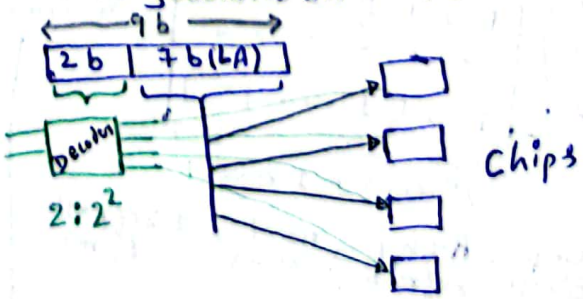
*) Instructions
Fixed length (opcode variable)
Ex: instr.n size = 16
Address size = 5



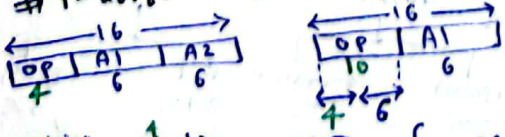
Variable length (opcode fixed)
Ex: opcode size = 6
Address size = 5



*) 32K × 1 RAM chip ⇒ 32K b capacity
*) Multiple chips in single memory:-
1) vertical arrngmnt:- used when no. of addresses required are more than no. of addresses in a single chip.
Ex: 4 chips of size 128 × 8 b (each)
⇒ Total capacity = 512 × 8 b
CPU generates LA = 9 b



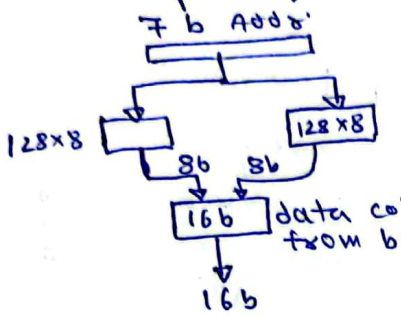
*) CPU supports 2-Addr. & 1-Addr. instr.ns
Instruction size = 16 b
Address size = 6 b
2-addr. instr.ns = 10
1-addr. " = ? = x



Remaining from 2-Addr. = $2^4 - 10$ = (6)
(6) × 2^6 = x
⇒ x = 384

2) Horizontal arrangement: Used when data size on each address required more than a chip capacity.

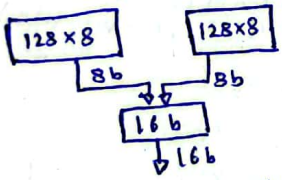
Ex: 2 chips of 128×8 b ⇒ 7b LA
we need capacity of 128×16 b



3) Hybrid: Used when no-of addr. & no-of data bits both are required more than 1 chip

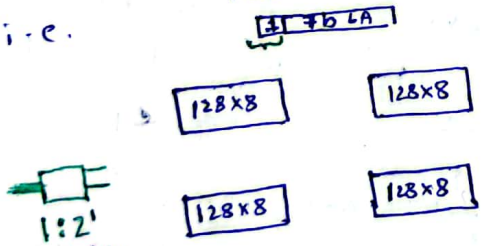
Ex: From 128×8 b chips ⇒ 7b LA
create 256×16 b chip.

#cols = x
8+8+8+... x = 16
⇒ x = 2



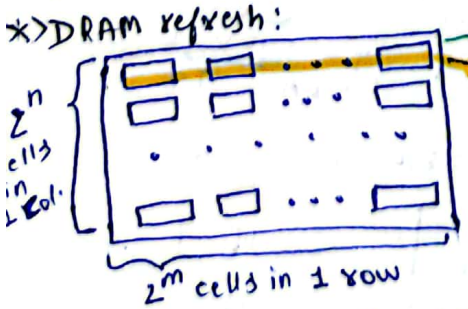
#rows = 256/128 = 2 ⇒ 1:2' decoder

i.e.

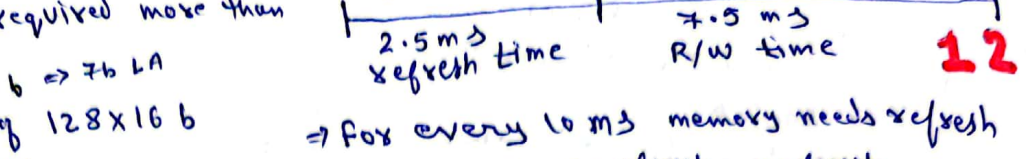


Q:) Required capacity	1 chip capacity	Arrangement #chips | rows | cols
128K×8 b	128K×8b	1 :
256M×16b	64M×8b	8 : 4×2
128M×64b	128M×8b	8 : 1×8
4G×16b	256M×8b	32 : 16×2

*) DRAM refresh:



1 chip refresh time = K-chip refresh time = no. of rows × 1 refresh time



⇒ For every 10 ms memory needs refresh

Next refresh = previous refresh Start time + refresh period

*) Hit ratio = #hits / Total references

*) In single access of cache, whole block can be accessed from cache.

*) Locality of reference :-
1) Spatial: nearby addresses will be accessed soon
2) Temporal: same address will be accessed after some time.

*) when locality of reference used, EMAT में MM से replace करने में T_{block} लगता है. -

i.e. T_{block} = Block access time in MM = block size × T_{mm}

Simultaneous
⇒ EMAT = $h \times T_{cm} + (1-h)(T_{block})$ // w/o copying into cache
Or EMAT = $h \times T_{cm} + (1-h)(T_{cm} + T_{block})$ // with copying into cache when miss happened

Hierarchical
EMAT = $h \times T_{cm} + (1-h)(T_{cm} + T_{block})$ // w/o copy
Or EMAT = $h \times T_{cm} + (1-h)(T_{cm} + T_{block} + T_{cm})$ // with copy

*) EMAT in write-through & write-back :-
1) write through: immediately update MM.
→ EMAT = % of Read operations × T_R + % of write operations × T_w

$T_R = h \cdot T_{cm} + (1-h) \times (T_{mm})$ (or) $(T_{mm} + T_{cm})$

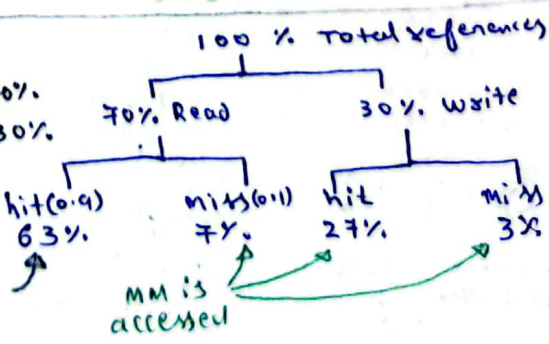
$T_w = \max(T_{cm}, T_{mm})$

→ Effective hit ratio = h × read ratio = % of time memory not accessed (MM)

2) write back: MM updated at the end
→ EMAT = $h \cdot T_{cm} + (1-h)(T_{mm} + \text{write back})$
(or) $(T_{mm} + T_{cm} + \text{write back})$

write back time = dirty % (i.e. fraction of dirty blocks) × T_{block} (i.e. MM block access time)

Note:
Ex: h = 0.9
% read op's = 70%.
% write op's = 30%.



1) Direct mapped cache:

a) MUX:
# of MUX = Tag bits
Size of MUX = no. of blocks in cache : 1 = 2^{BO} : 1

b) comparator:
# of comparator = 1
size = Tag bits

c) Hit latency = MUX Delay + comparator delay

2) Set associative: (k-ways)

a) MUX:
# = K x Tag bits
Size = 2^{setno} : 1 ; Set no. = bits for set no.

b) comparators:
# = K
size = Tag bits

c) OR gate
= 1
i/ps = K
} can be implemented using K:1 MUX

d) Hit latency = MUX delay + comparator delay + OR gate delay

3) Fully associative:

a) MUX:
NO mux needed

b) comparator:
# = 2^{BO}
Size = Tag bits

c) OR gate
= 1
i/ps = 2^{BO}

d) Hit latency = comparator delay + OR gate delay

NOTE) Control unit design:-
1) Hardwired (fastest)
microprogr 2) Horizontal
3) Vertical (slowest)

#) Control Unit Design ▷ Hardwired C U ▷ Microprogrammed C U

1) Hardwired Control unit:-

Question GATE-2005

A hardwired CPU uses 10 control signals S1 to S10, in various time steps T1 to T5, to implement 4 instructions I1 to I4 as shown below:

	T1	T2	T3	T4	T5
I1	S1, S3, S5	S2, S4, S6	S1, S7	S10	S3, S8
I2	S1, S3, S5	S8, S9, S10	S5, S6, S7	S6	S10
I3	S1, S3, S5	S7, S8, S10	S2, S6, S9	S10	S1, S3
I4	S1, S3, S5	S2, S6, S7	S5, S10	S6, S9	S10

Which of the following pairs of expressions represent the circuit for generating control signals S5 and S10 respectively?

(A) S5=T1+I2·T3 and S10=(I1+I3)·T4+(I2+I4)·T5

(B) S5=T1+(I2+I4)·T3 and S10=(I1+I3)·T4+(I2+I4)·T5

(C) S5=T1+(I2+I4)·T3 and S10=(I2+I3+I4)·T2+(I1+I3)·T4+(I2+I4)·T5

(D) S5=T1+(I2+I4)·T3 and S10=(I2+I3)·T2+I4·T3+(I1+I3)·T4+(I2+I4)·T5

For S5

$$= T_1(I_1 + I_2 + I_3 + I_4) + T_3(I_2 + I_4)$$
$$T_1 \cdot 1 + I_2 T_3 + I_4 T_3$$
$$= T_1 + I_2 T_3 + I_4 T_3$$

For S10

$$= I_2 T_2 + I_3 T_2 + I_1 T_4 + I_3 T_4 + I_2 T_5 + I_4 T_5 + I_4 T_3$$
$$= (I_2 + I_3) T_2 + (I_1 + I_5) T_4 + (I_2 + I_4) T_5 + I_4 T_3$$

Question GATE-2004

A CPU has only three instructions *I1*, *I2* and *I3*, which use the following signals in time steps *T1* – *T5*:

I1 : *T1* : Ain, Bout, Cin
T2 : PCout, Bin
T3 : Zout, Ain
T4 : Bin, Cout
T5 : End

I2 : *T1* : Cin, Bout, Din
T2 : Aout, Bin
T3 : Zout, Ain
T4 : Bin, Cout
T5 : End

I3 : *T1* : Din, Aout
T2 : Ain, Bout
T3 : Zout, Ain
T4 : Dout, Ain
T5 : End

Which of the following logic functions will generate the hardwired control for the signal Ain ?

A. *T1.I1 + T2.I3 + T4.I3 + T3*
B. *(T1 + T2 + T3).I3 + T1.I1*
C. *(T1 + T2).I1 + (T2 + T4).I3 + T3*
D. *(T1 + T2).I2 + (T1 + T3).I1 + T3*

$I_1(T_1 + T_3) + I_2(T_2) + I_3(T_2 + T_3 + T_4)$

$= I_1 T_1 + I_1 T_3 + I_2 T_3 + I_3 T_2 + I_3 T_3 + I_3 T_1$

$= I_1 T_1 + (I_1 + I_2 + I_3) T_3 + I_3 T_2 + I_3 T_4$

$= I_1 T_1 + I_3 T_2 + I_3 T_4 + T_3$

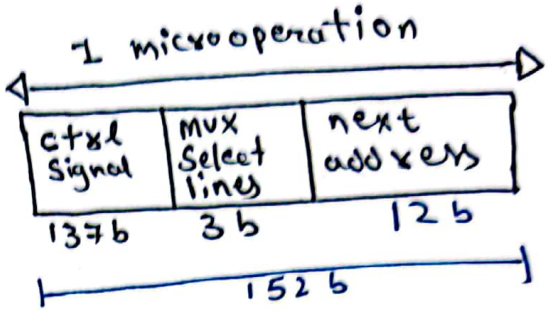
$= I_1 T_1 + I_3 (T_2 + T_4) + I_3$

Microprogrammed CU :-

1) Horizontal MCU :-

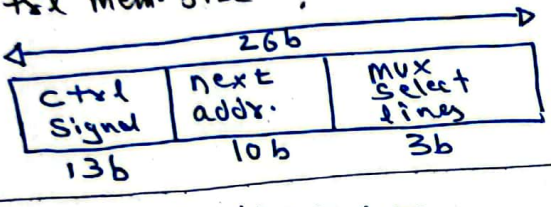
Q:) #Instr^{n}s = 256
Each instr^{n} ⇒ 16 micro oper^{n}s
Ctrl mem. size = ?

#microoperations = 256 × 16 = 2^{12}
⇒ Address = 12 b



Therefore,
ctrl mem. size = # micro operations × sized each
= 2^{12} × 152 = 608 kb

Q:) 8 status bits (in MUX input) ⇒ MUX Select lines = 3b
one microoperation is 26b; ctrl signal = 13b
Ctrl mem. size = ?



Therefore,
ctrl mem. size = 2^{10} × 26b = 3.25 KB

2) Vertical MCU :-

Q:) #control signals = 120
Control signals Divided into groups :-

G1	G2	G3	G4	G5	G6	
30	13	12	3	27	35	Σ = 120 b
5b	4b	4b	2b	5b	6b	Σ = 26b

#Bits saved on using vertical MCU as compared to Horizontal MCU = ?

Therefore, (26b) for control signal in vertical MCU
(120b) " " " " Horizontal "

⇒#Bits saved (per control signal) = 120 − 26 = 94

Q:) ... same as previous.
Mux select lines = 4b
#microoperations = 2^{11} ⇒ address = 11b
Size of memory = ?

Horizontal MCU

ctrl signal	mux select	next addr.
120b	4b	11b

135b

mem. size = 2^{11} × 135 b

Vertical MCU

ctrl signal	mux select	next addr.
26b	4b	11b

41b

mem. size = 2^{11} × 41 b

Q:) #ctrl signals = 100
94 ctrl signals are grouped as :-

G1	G2	G3	G4	G5	G6	
30	25	2	5	10	22	Σ = 94 b
6b	5b	1b	3b	4b	5b	Σ = 23 b

Remaining 6 ctrl signals can be kept in any group
These 6 ctrl signals have to be kept in Horizontal manner

Therefore,
For ctrl signal in Vertical MCU = 23+6 = 29b
" " " " Horizontal " = 100b