Computer Architecture :-

(7-8 marks)
(5-6 questions)

Syllabus :-

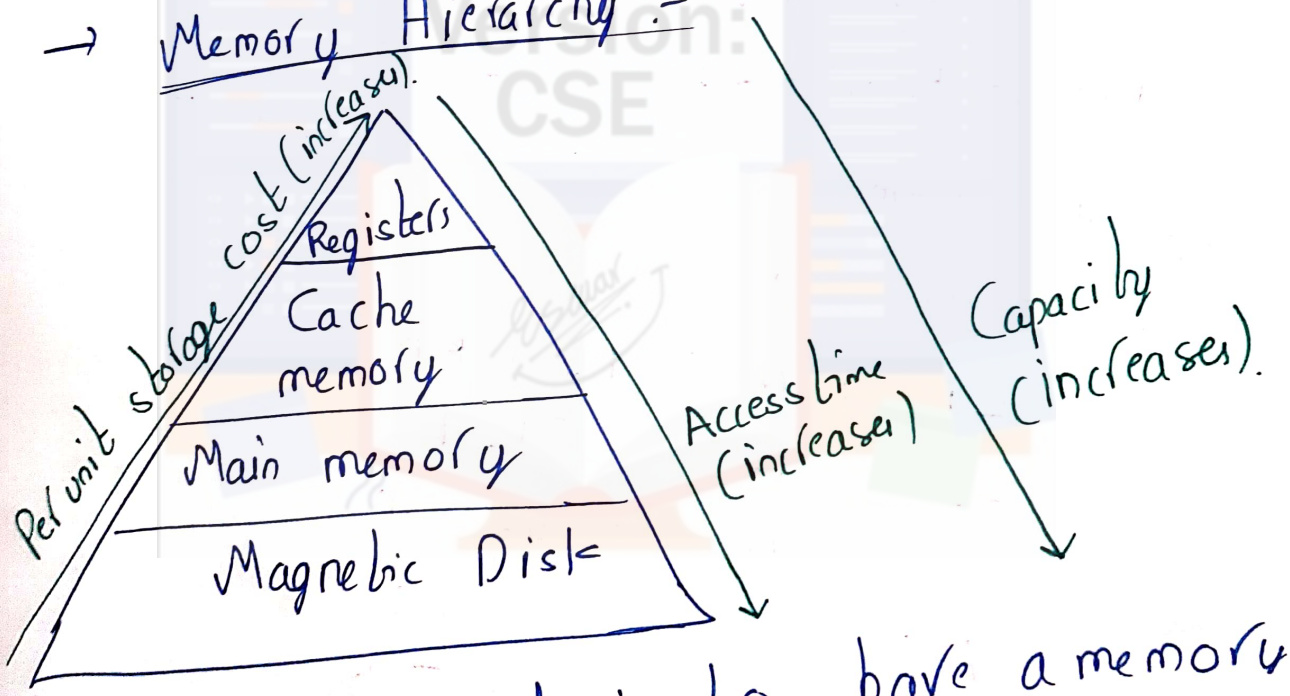
→ Machine instructions & addressing modes ④
→ ALU, data path & control unit ⑤
→ Instruction pipelining ③
→ Memory hierarchy ①
→ I/O interface. ②

(Morris Mano)

→ Computer Organization :-

Computer organization refers to the way in which the haldware components operate & the way they are connected together that realize the architectural specifications.

→ Memory Hierarchy :-



Our goal is to have a memory which has → large capacity
→ less per unit cost
→ less access time (Fast access).

but capacity $\propto \frac{1}{\text{access time}}$

hence both can't be low

that is the main reason why we use multiple types of storage devices.

Access time/speed :- Register > Cache > MM > SM

Cost :- Register > Cache > MM > SM

Size :- Register < Cache < MM < SM

ex: my system

3 levels of cache.

L_1 cache :- 128 KB

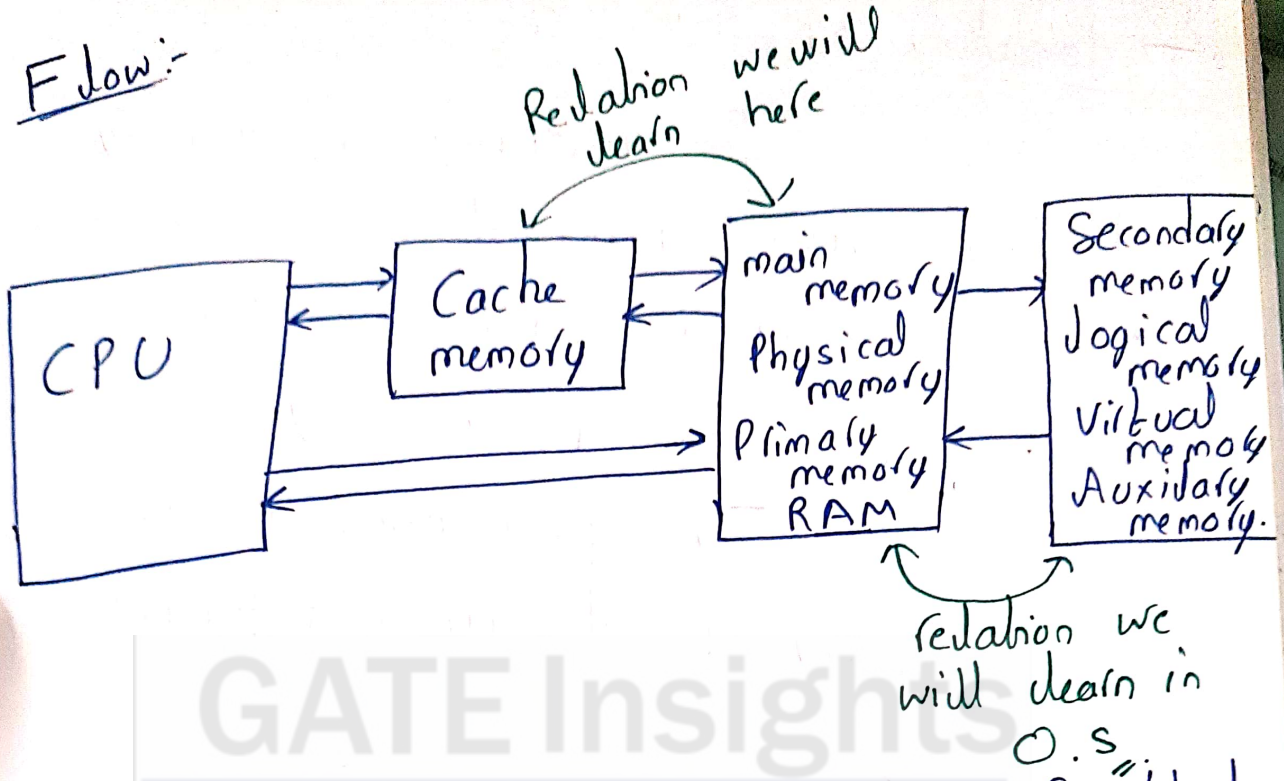
L_2 cache :- 512 KB

L_3 cache :- 3.0 MB

MM :- 8 GB

SM :- 2 TB

Flow:-



Logic:- if we are searching for a file/data

First we check in cache

if found its a (Cache hit)

else its a (Cache miss)

if not found then we search in Main memory.

if found copy it to cache

else

With the help of main memory search in secondary memory if found bring to main memory & use it

else not found.

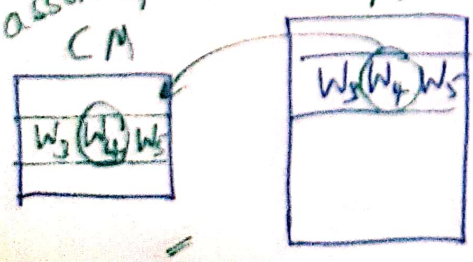
→ Locality of Reference:

The references to memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as property of locality reference.

Spatial locality (w.r.t space)

→ If a word is accessed now then the word adjacent to it will be accessed next.

whole block bought to cache on assuming W_3 or W_5 will be accessed next

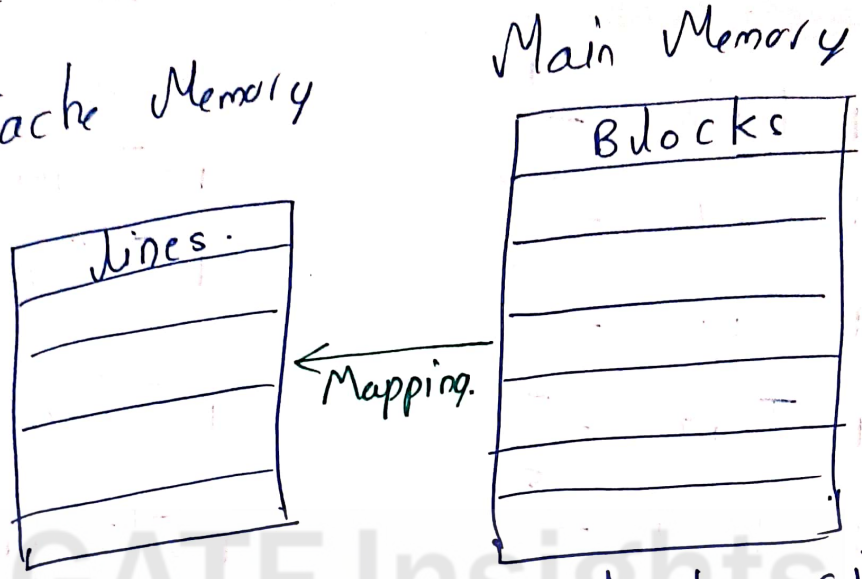


Temporal locality (w.r.t time).

→ It refers to the reuse of specific data & / or resources within a relative small-time during. Or the most frequently used will be need soon.

→ LRU (example). (least recently used).

Note:-



Note size of line - Blocks size.

Hence as we know MM size is greater than Cache memory.

Hence at any movement of Line. all Blocks can't be in cache

Hence we using mapping to give slots as per requirement.

3 types.

i) Direct Mapping
ii) Associative Mapping.
iii) Set Associative Mapping.

(very imp)

Metrics ** space

1 Kilo	2^{10}
1 Mega	2^{20}
1 Giga	2^{30}
1 Tera	2^{40}
1 Peta	2^{50}

Time

1 sec	m sec
10^{-3}	milli second
10^{-6}	micro second µ sec
10^{-9}	nano second n sec.
10^{-12}	pico second

//.

Basics: (smallest unit we call a word)
(collection of words form block)
(collection of blocks form main memory)

Main memory

	0	1	2	3
B0	W_0	W_1	W_2	W_3
B1	W_4	W_5	W_6	W_7
B2	W_8	W_9	W_{10}	W_{11}
B3	W_{12}	W_{13}	W_{14}	W_{15}
B4	W_{16}	W_{17}	W_{18}	W_{19}
B5	W_{20}	W_{21}	W_{22}	W_{23}
B6	W_{24}	W_{25}	W_{26}	W_{27}
B7	W_{28}	W_{29}	W_{30}	W_{31}

hence to find

W_{18} ⇒ write binary of 18

10010

⇓

we have 8 blocks means 3 bits to differentiate

we have 4 word, 2 bits to differentiate

i.e 100 10

100 → Block no.

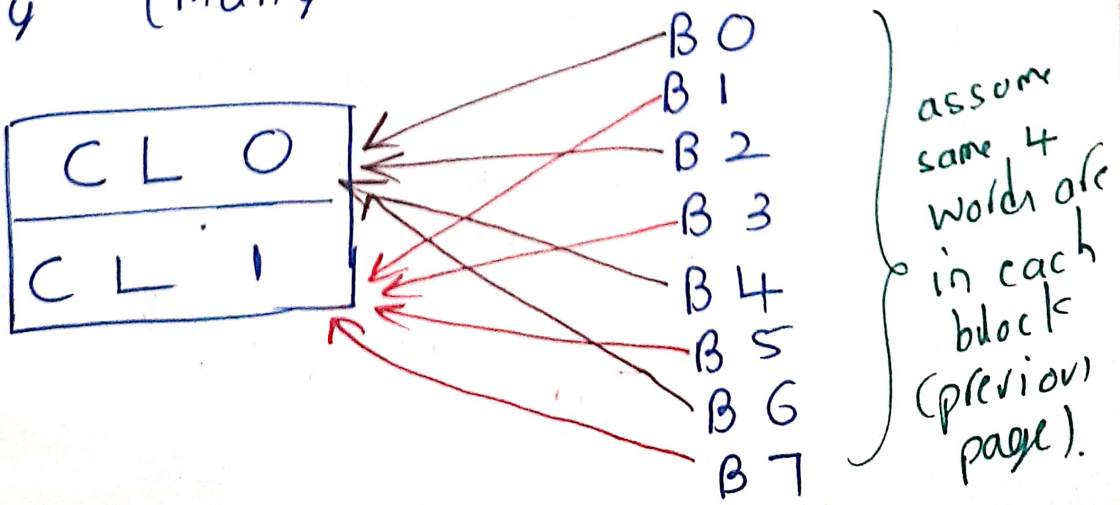
10 → Block offset

4 – 2

Hence in this we can get location.

I) Direct Mapping :-

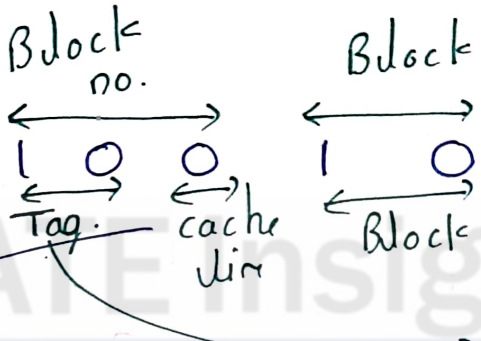
In direct mapping scheme the main memory blocks are directly mapped onto a particular cache memory (many to one mapping).



to find exactly to which line

W_{18} goes

$\Longrightarrow$ 10010



only 1 bit coz we have only 2 lines

1 bit is enough

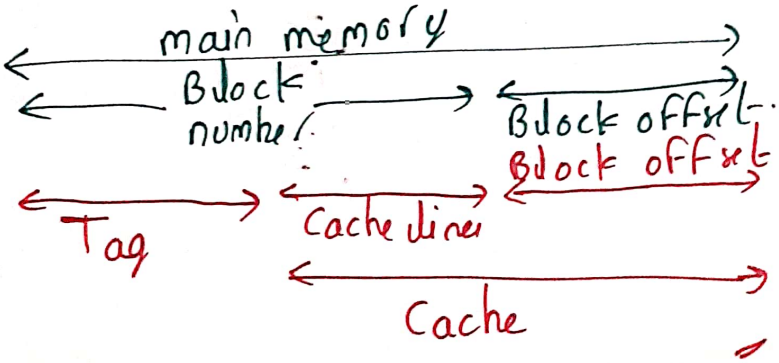
0 means CL-0

say it, position like

CL0 ⇒ B0
B2
B4
B6

2nd place (10)

1st time a bit confusing but only remember the diagram it will be easy.



For solving problems if you remember previous diagram is enough.

ex: given. (direct mapping)

main memory size = 2^{32} bytes

cache size 32 KB = 2^{15} bytes.

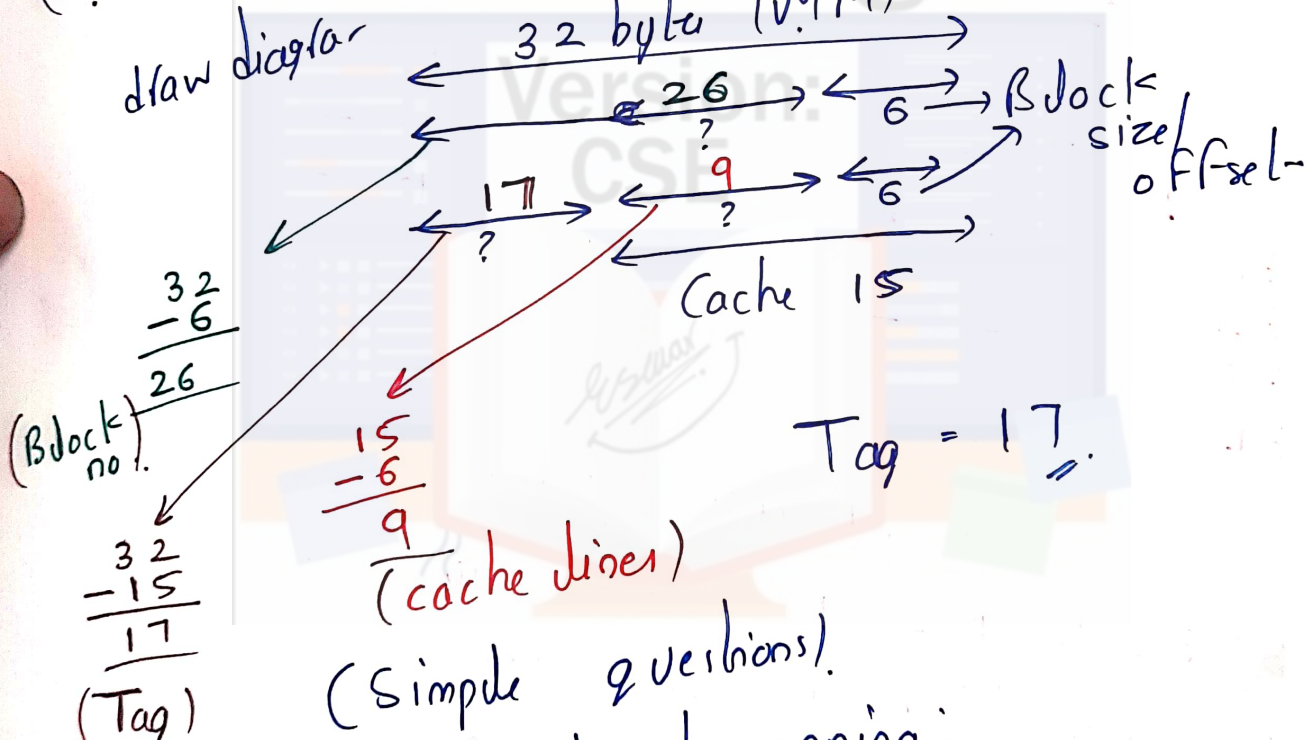
cache block size = 2^{6} bytes.

(Gate 2021)

draw diagram

(or offset)

Tag = ?



32 − 6 = 26 (Block no.)

15 − 6 = 9 (cache lines)

32 − 15 = 17 (Tag)

Tag = 17.

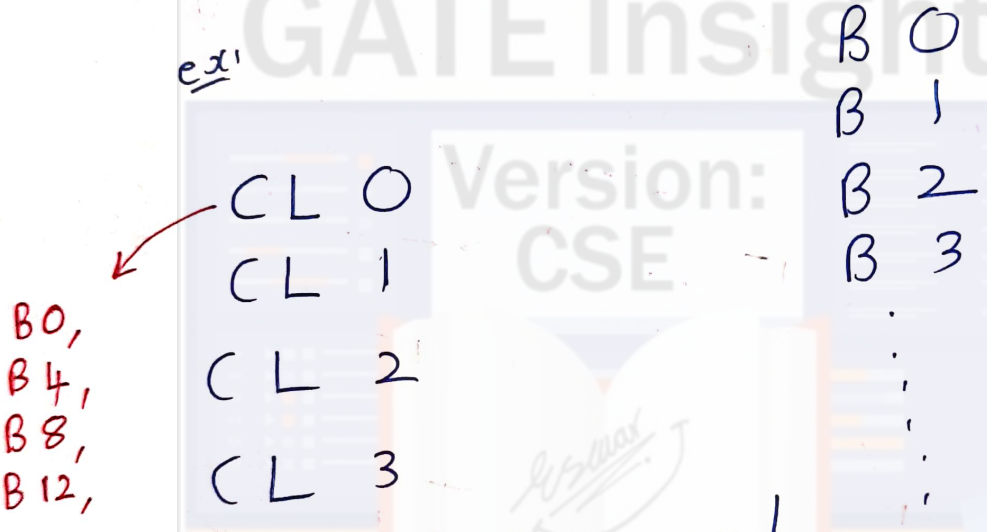
(Simple questions)

Problem: with direct mapping:

Here we are assign a particular cache lines to a set of blocks.

Hence if that particular blocks are called again & again Even we have space in cache memory as we are forced that slot only for them hence this is any issue.

ex:



Assume if a program used B0, B4, B8, B12 in exact order multiple times as they all map to CL0 even CL1, 2, 3 are empty as this is direct mapping we push them only to CL0

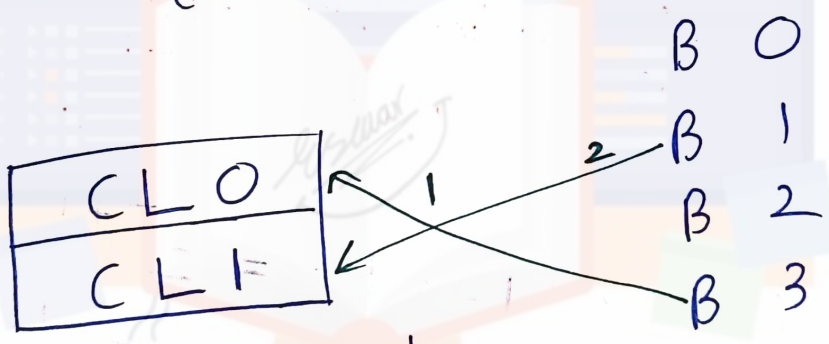
II)

Hence due to this issue only we designed Associative mapping.

II) Associative Mapping :-

here A block of main memory can be mapped freely to any available cache line.

This makes fulling associative mapping more feasible than direct mapping (many to many mapping).



B_3, B_1 hence based on their respective order they get there slots. (no pattern).

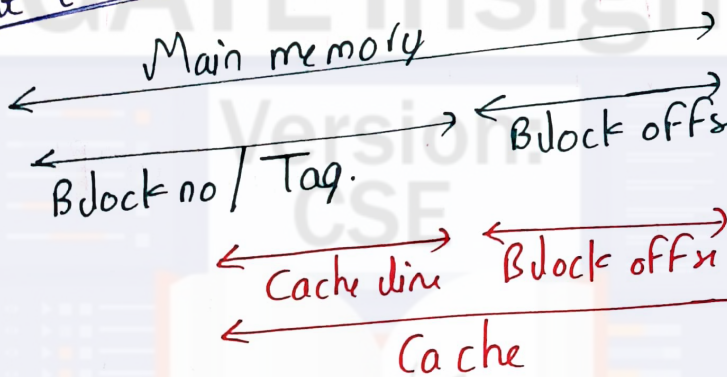
If Cache full we Cache replacement algo's ← will learn soon.

hence as there is no pattern here tag = Block no //.

CL 0 Tag = block no

CL 1 Tag = block no //.

Update the diagram:

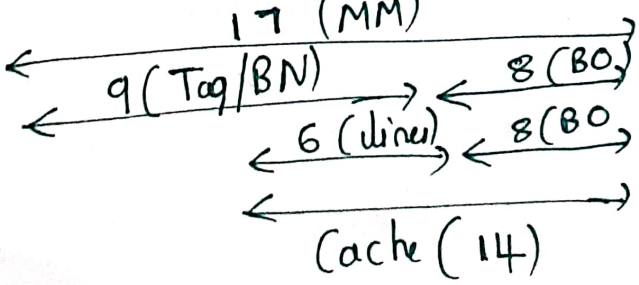


//.

ex! MM size = 128 KB = $2^{7+10} = 2^{17}$

Cache size = 16 KB = $2^{4+10} = 2^{14}$

Block size = 256 B = 2^{8}



∴ No. of tag bits = 9

Tag directory size = 9×2^6

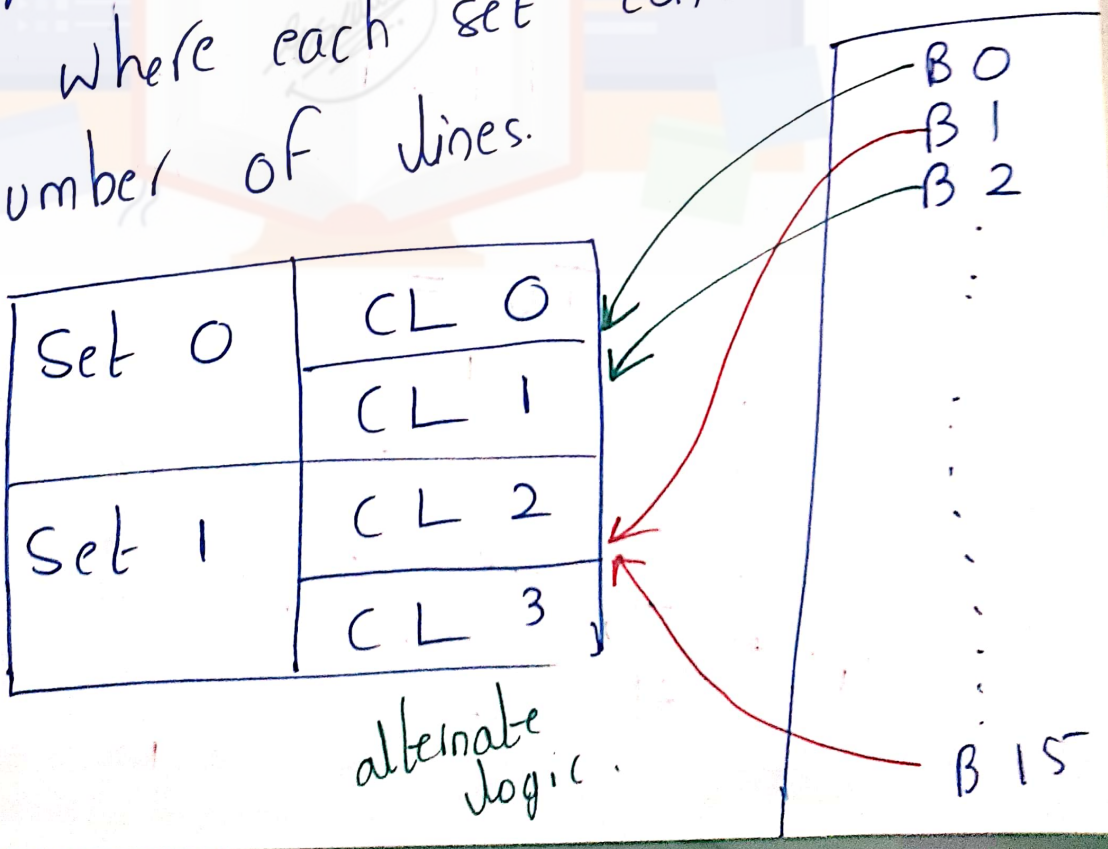
$= 9 * 64$ ||.

As here in associative we gave full freedom which made tag bit = Block no.

hence which is not ideal ||.

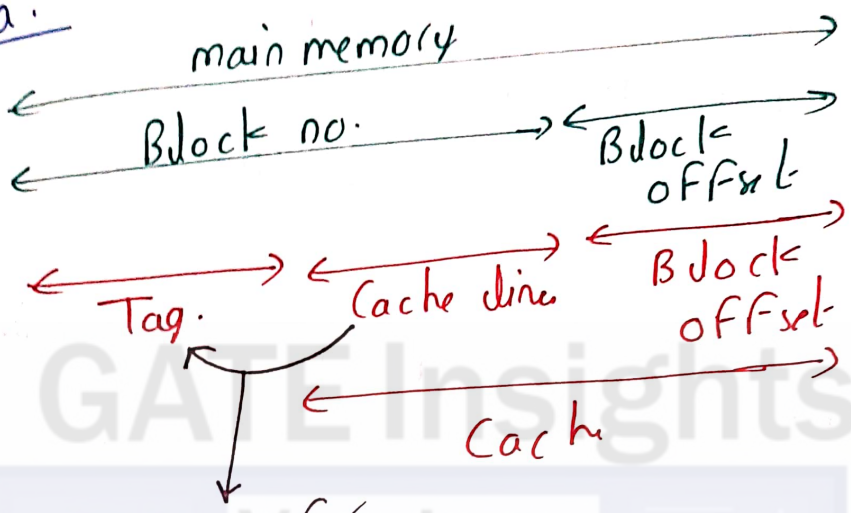
III) Set Associative Mapping :

In k-way set associative mapping, cache lines are grouped into sets where each set contains k number of lines.



Hence set associate is a combination of associate & direct.

Formula:



ex:

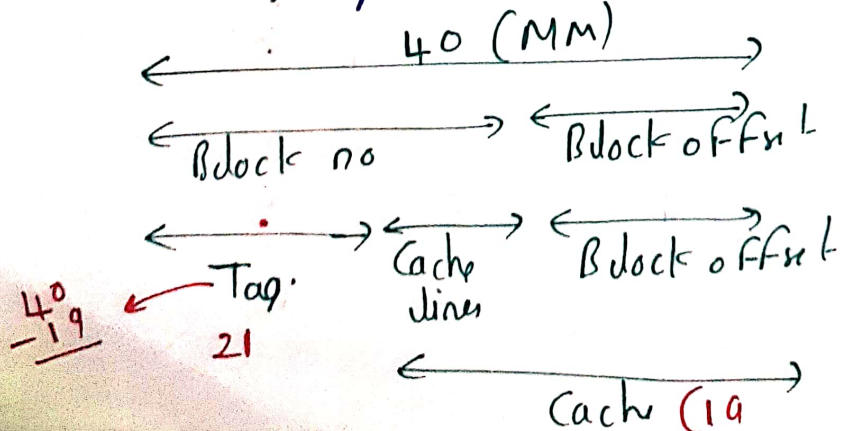
Gate 2016

Main memory = 2^{40} bytes

Cache = 512 KB = 2^{19} bytes

8 way set associative.

Tag size = ?



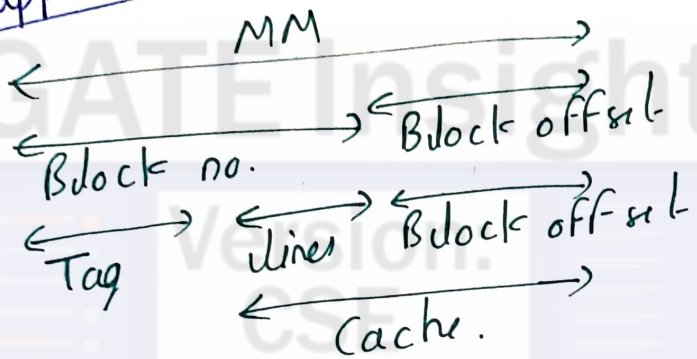
8 way

3

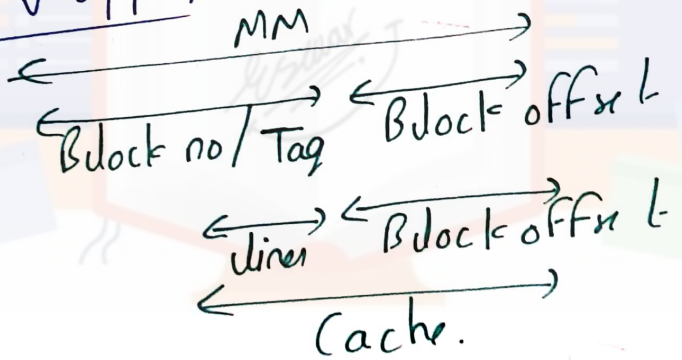
21 + 3 ⇒ 24 bits of Tag

Note:

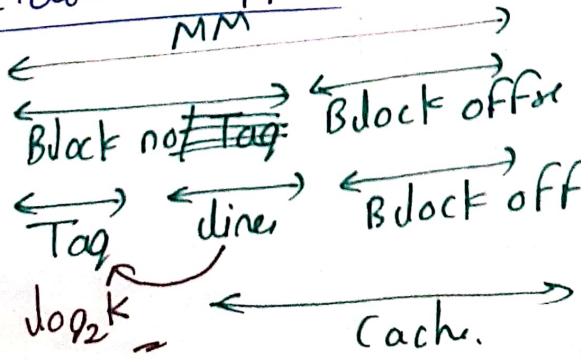
Direct Mapping:



Associate Mapping:



Set Associative mapping:



→ Cache Replacement Policies:

As we know when cache is full we need to replace some to give place for a block
Hence to decide whom to remove/replace we use Cache Replacement Policies.

In direct mapped cache, the position of each block is predetermined hence no replacement policy needed!!.

In fully associative & set associative caches there exist policies

Types:
i) FIFO
ii) LRU
iii) Optimal.

will repeat in OS.

I) FIFO (First in First out)

The block which have entered first in the memory will be replaced first.

ex: Block Sequence.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

0, 1 & cache memory has 4 lines.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	7		3		3		3			3	2		2				
	0	0	0		0		4		4			4	4		7				
		1	1		1		1		0			0	0		0				
			2		2		2		2			2	1		1				
M	M	M	M		M		M		M			M	M		M				

10 - ~~mill~~ miss Total // Faults.

Belady's Anamaly: (We assume if no. of cache lines increase Faults will decrease But in FIFO we might have some case in which Faults increase if we increase cache line i.e Belady's Anamaly.

ex: (3 cache lines)

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5			5	5	
	2	2	2	1	1	1			3	3	
		3	3	3	2	2			2	4	
M	M	M	M	M	M	M			M	M	

9 - Faults.

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1			5	5	5	5	4	4
	2	2	2			2	1	1	1	1	5
		3	3			3	3	2	2	2	2
			4			4	4	4	3	3	3
M	M	M	M			M	M	M	M	M	M

10 Faults

3 lines	4 lines	
9 Faults	10 Faults	} Belady's anomaly.

II) LRU (least recently used). (Past)

The page which was not used for the longest period of time in the past will get replaced first.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	7		3		3				3						7		
	0	0	0		0		0				0						0		
		1	1		1		4				1						1		
			2		2		2				2						2		
M	M	M	M		M		M				M						M		

8 Faults.

III) Optimal algorithm: (Future work)

The page which will not be used for the longest period of time in future references will be replaced first.

Best performace but can't be used as we have no idea about future.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	7	3		3					3						7		
	0	0	0	0		0					0						0		
		1	1	1		4					1						1		
			2	2		2					2						2		
M	M	M	M	M		M					M						M		

8 - Faults.

Type of miss:

Compulsory Miss:- When CPU demands for any block for the first time then definitely a miss is going to occur as the block needs to be brought into cache. (brown underline).

Capacity Miss:- Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (red underline).

Conflict Miss :-

In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set / block frame.

ex: Consider a 4 way set associative cache

16 cache blocks.

MM consist of 256 blocks.

request order.

0, 255, 1, 4, 3, 8, 133, 159, 216, 129, 63, 8, 48, 32, 73, 92, 155.

which of the following will not be in cache.

A) 3 B) 8 C) 129 D) 216

4 way set associative

64 line in 1 set

16 cache block

256 block in memory

0, 255, 1, 4, 3, 8, 133, 159, 216, 129, 63, 8, 48, 32, 73, 92, 155.

same like hashing

0 mod 4 = 0

255 mod 4 = 3

1 mod 4 = 1

4 mod 4 = 0

3 mod 4 = 3

8 mod 4 = 0

133 mod 4 = 1

159 mod 4 = 3

216 mod 4 = 0

129 mod 4 = 1

63 mod 4 = 3

8 mod 4 = 0 ✓

48 mod 4 = 0 ↑

32 mod 4 = 0 ↑

73 mod 4 = 1

92 mod 4 = 0 ↑

155 mod 4 = 3

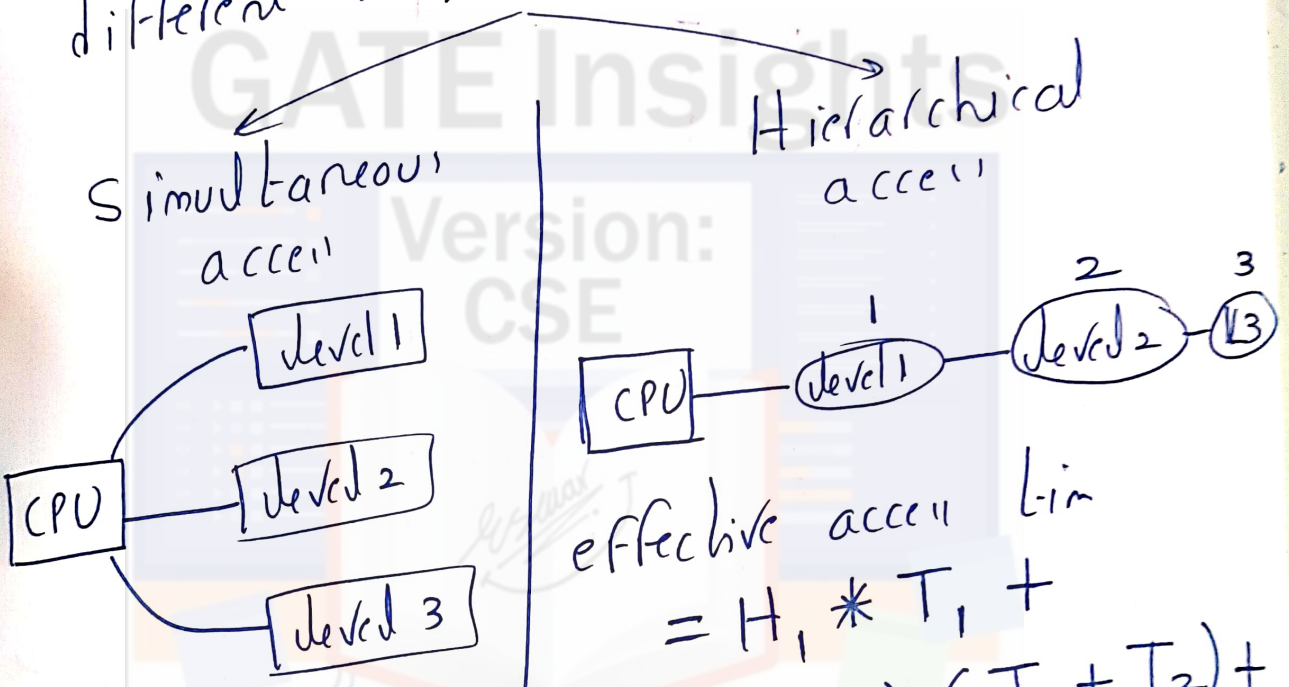
(Use LRU on swapping)

(D)

→ Memory Organization:

→ Memory is organized at different levels.

→ CPU may try to access different levels of memory in different ways



effective access time

$= H_1 * T_1 +$
$(1-H_1)(H_2)(T_2) +$
$(1-H_1)(1-H_2)(T_3)$ //

effective access time

$= H_1 * T_1 +$
$(1-H_1)(H_2)(T_1+T_2) +$
$(1-H_1)(1-H_2)(T_1+T_2+T_3)$.

H → hit rate
T → time / access time //

ex:

(Gate 2015) on cache. { read request miss ⇒ 50 nsec → $T_1 + T_2$
read request hit ⇒ 5 nsec → T_1

80% request hit

$\Rightarrow H_1 * (T_1) + (H_1 - 1) H_2 (T_1 + T_2)$

$\Rightarrow 0.8 * 5 + (0.2)(1)(50)$

$\Rightarrow 4 + 10$ nsec

$\Rightarrow 14$ nsec.

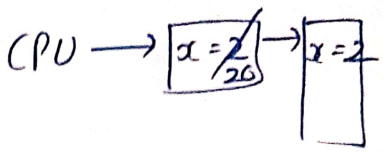
→ Cache Coherence Problem :-

As we know the data which we store in cache is a copy of MM.

Hence as multiple copies at different level of memories the inconsistency may occur, this problem is known as Cache Coherence Problem.

$+T_2$

It can be solved using 2 techniques.

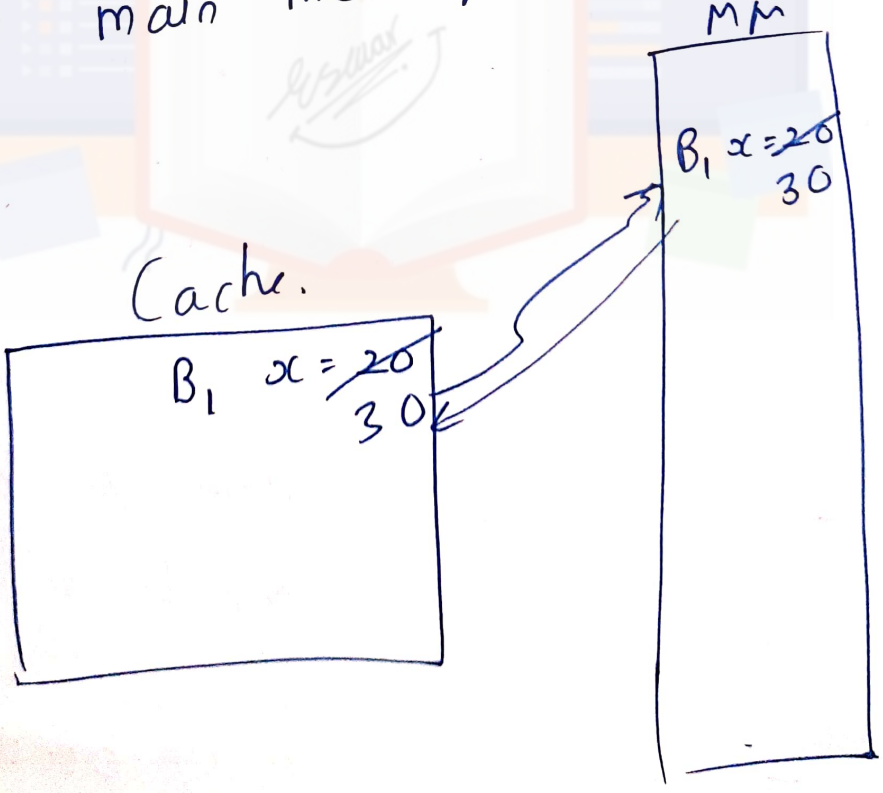


→ Write Trough
→ Write Back

I) Write Trough:

Write through is used to maintain the consistency between the cache & main memory.

According to it if the cache copy is updated at the same time main memory is also updated.



Adv: It provides the highest level of consistency.

Disadv: It requires more number of memory access.

II) Write Back:

Write back is also used to maintain the consistency between the cache and main memory.

According to it all the changes performed on cache are reflected back to the main memory in the end. (We maintain a bit modified bit ← 1 (modified), 0 (not modified))

Adv:

→ less number of memory access & less write operations

Disadv:

→ Inconsistency may occur.

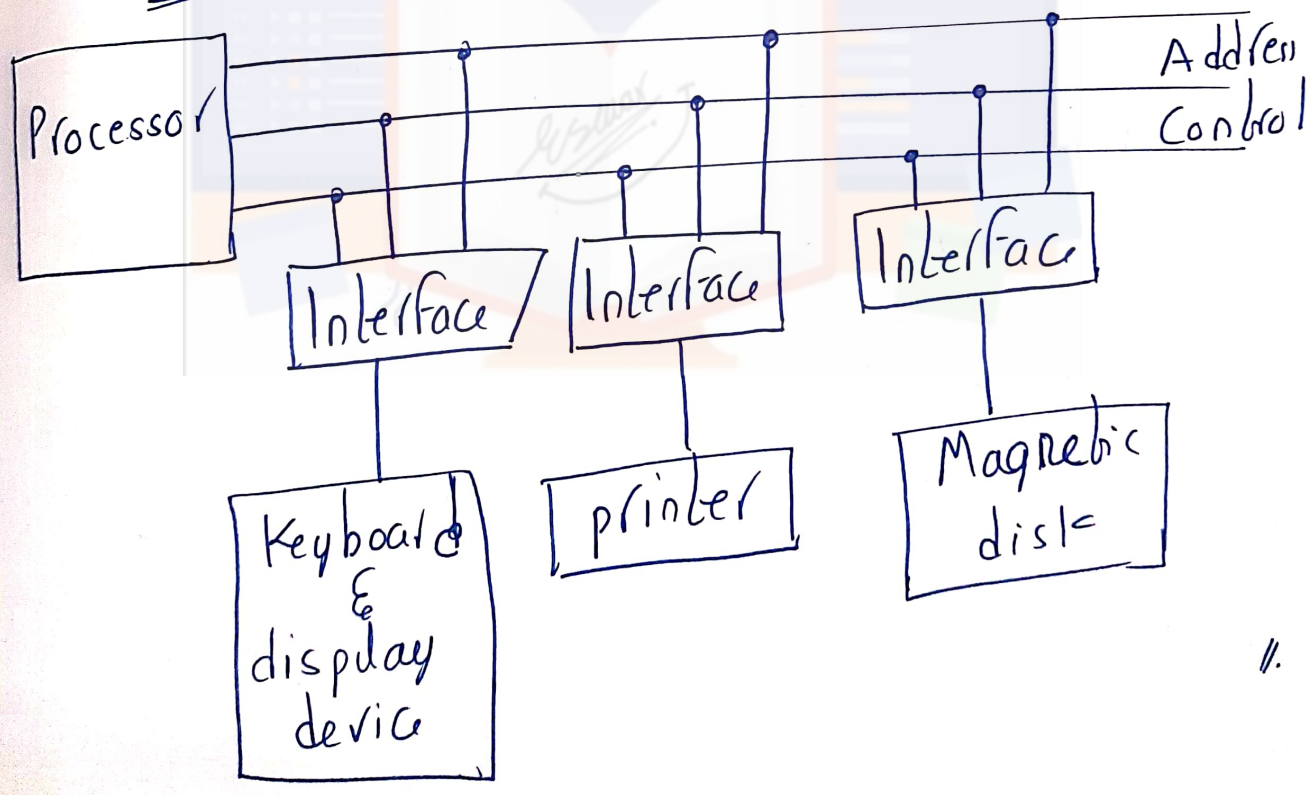
→ I/O management:

As we know the next most important component after CPU & memory. is input & output (as without input & output the computer cannot serve any).

We cannot directly connect a i/o device to computer.

We use interface because

Interface:



Reason for having interface is:

Speed:- The speed of CPU & i/o device will usually be different.

Format: The data code & format of CPU & peripherals may be different

eg: ASCII, Unicode etc..

Physical orientation: Different device have organizations like optical, magnetic, etc...

Signal conversion: peripherals are electromagnetic & their manner of operation is different from the operation of CPU.

(Analog to digital etc...)

Address bus: Is Used to identify the correct i/o device among the number of i/o devices.

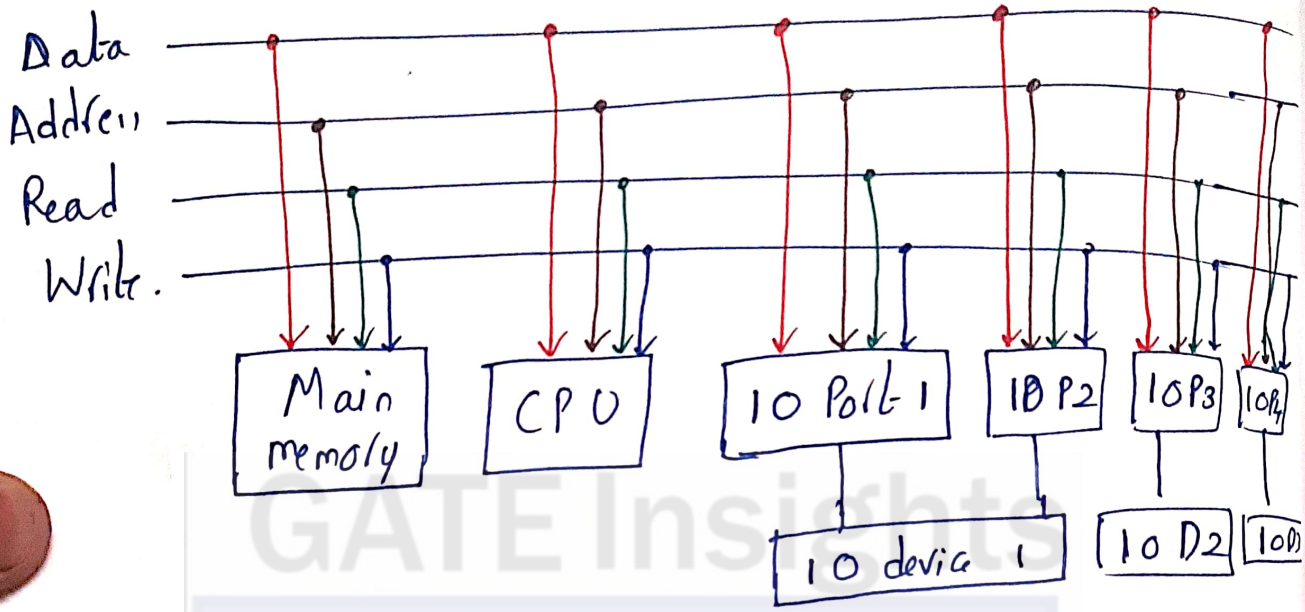
Control bus: After selecting a specific i/o device CPU sends a Functional code on the control line. (read, write).

Data bus: In this final step depends on the operation either CPU will put data on the data line & device will store/display it & wise versa.

Based on the connection b/n CPU & i/o devices

3 ways.

I) Memory Mapped I/O :-



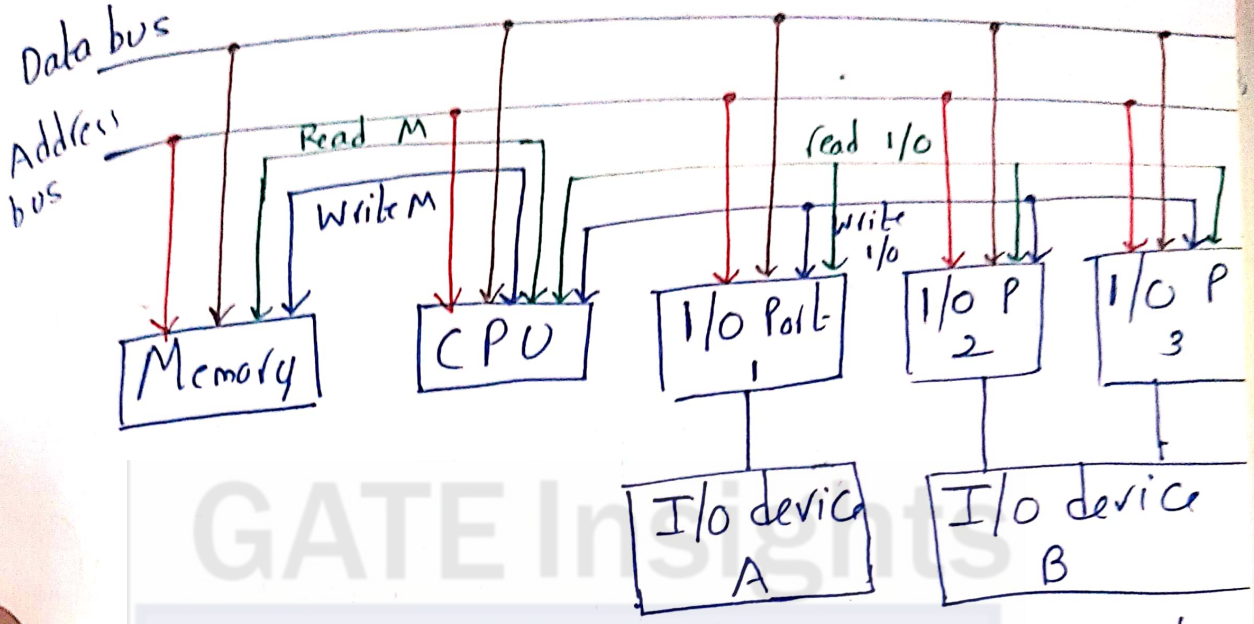
It uses the same address space to address both memory & I/O

ex: 8085

Adv: In typical computer there are more memory reference instruction than i/o instruction, but in memory mapped i/o all instruction that refer to memory are also available for i/o

Disadv: Total address get divided. some range is occopied by i/o while some by memory.

II) Isolated I/O :



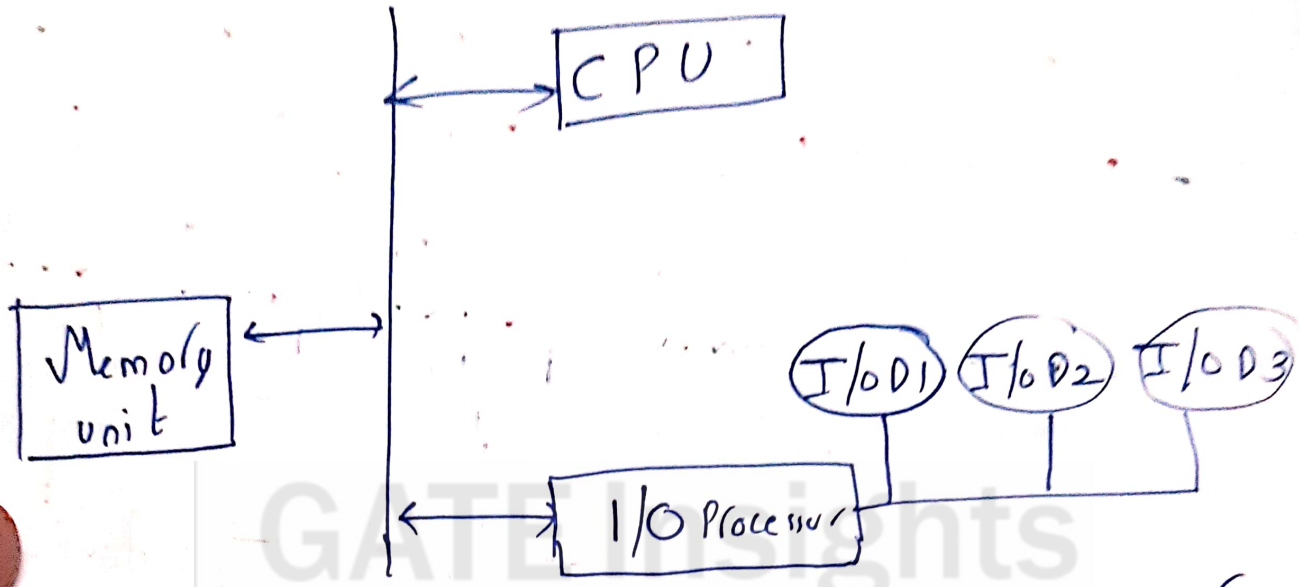
Here common bus to transfer data b/n the memory or i/o & CPU. The distinction between a memory & i/o transfer is made through separate read & write line.

Adv: Here memory is used efficiently as the address can be used two times

ex: 8086.

DisAdv: Need different control lines one for memory & other for i/o devices.

III) I/O Processor :-



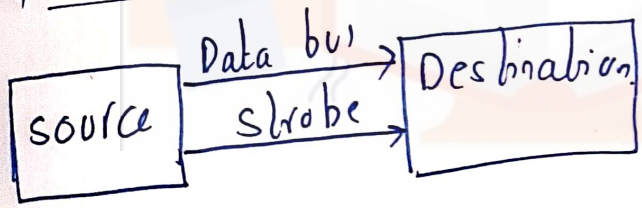
Computer has independent set of data, address & control buses, one for accessing memory & other for I/O. This is done in computers that provide a separate I/O processor other than CPU.

→ Synchronous v/s Asynchronous data transfer

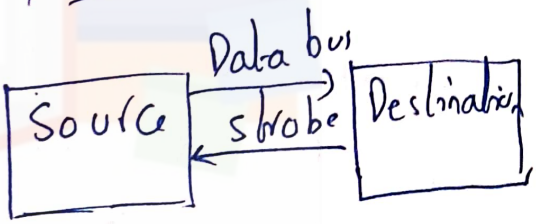
Synchronization is achieved by a device called master generator. which generate a periodic train of clock pulse
(The clock by CPU).

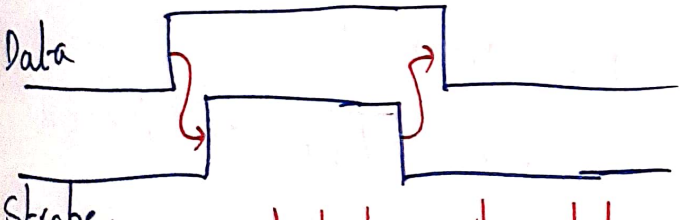
In Asynchronous data the timing units of two devices are independent that is they are under different control

i) Source initiated

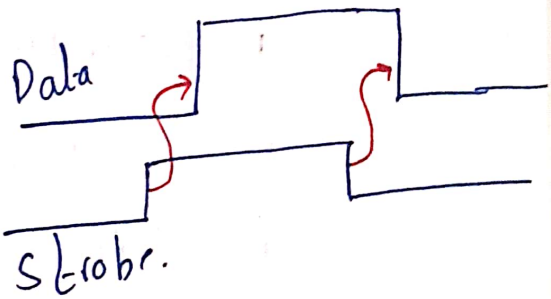


ii) Destination initiated

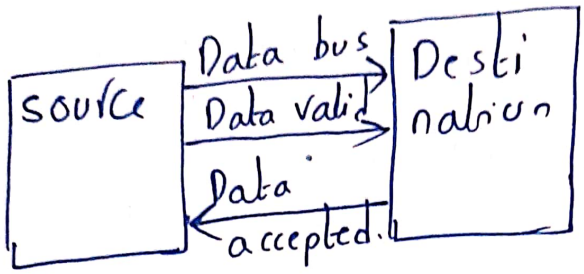




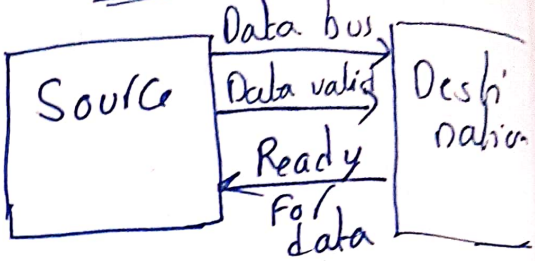
↳ is a signal high when data is ready to copy.

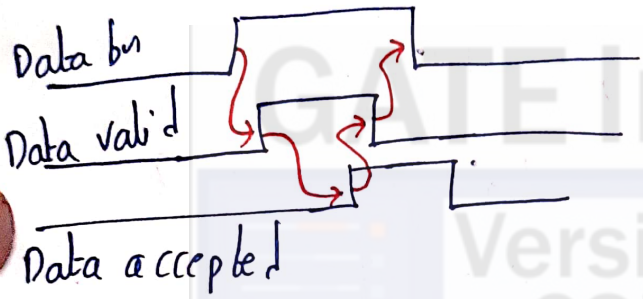


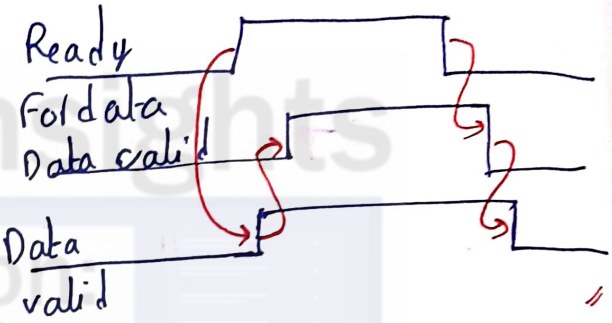
iii) Source initiated with ACK



iv) Destination initiated with ACK







→ Modes of Data Transfer :-

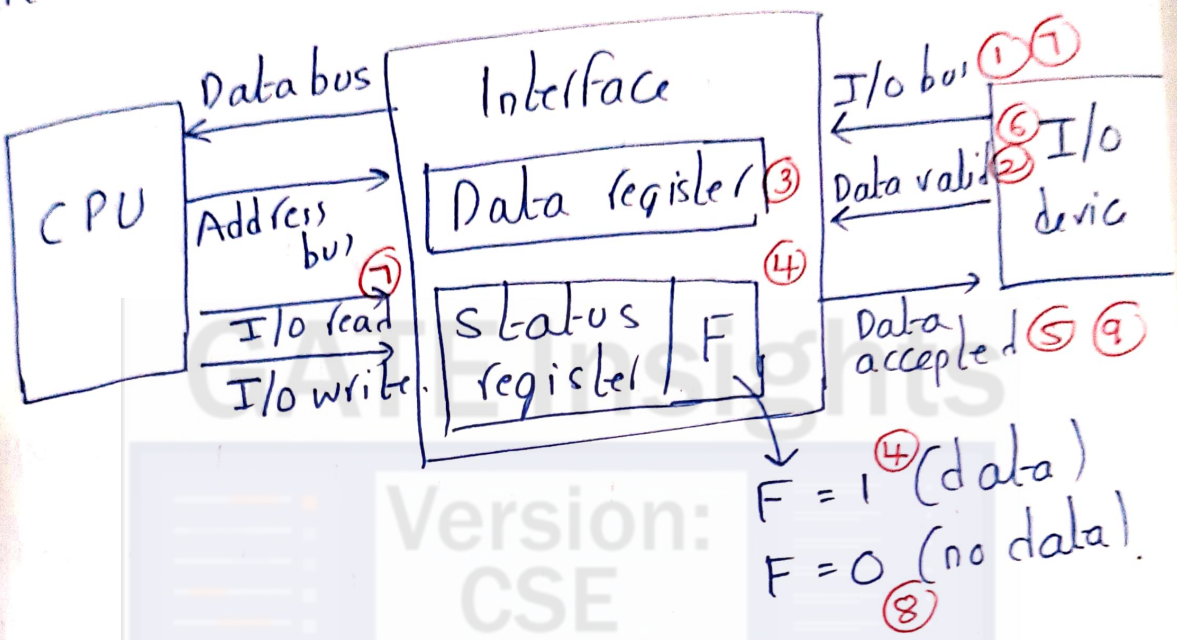
here we discuss how data communication will take place b/n CPU & I/O device.

3 methods

i) Programmed i/o
ii) Interrupt initiated i/o
iii) Direct Memory Trasfer/Access

I) Programmed I/O :-

In this I/O device cannot access the memory directly.



The flow is quite simple

Note: Here CPU alway check flag bit wheather it's 1 or 0

Hence many cycles of CPU are wasted here.

Hence indirectly CPU is like slave.

Hence this method is not an ideal way.

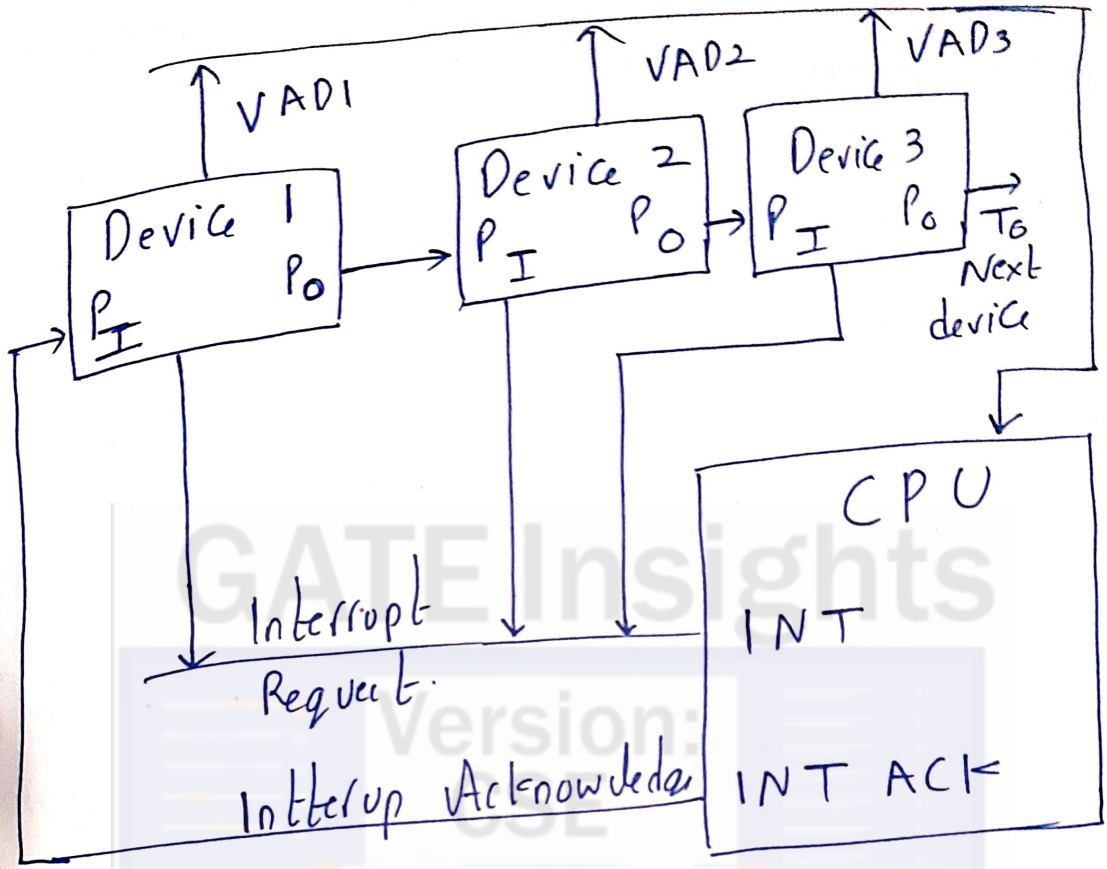
II) Interrupt initiated I/O :-

In this method I/O device interrupt CPU when it is ready for data transfer.

CPU keep executing instructions & after executing one instruction & before starting another instruction CPU wait & see if there is interrupt or not & if there is interrupt then take a decision whether CPU should entertain this interrup or continue with the execution.

Note: The one which we discu.. is daisy chain

It has priority.



If an I/O device wants to communicate it raises interrupt (0 11) (1, 2, 3).

Once CPU completes its current instruction it will check & decide whether to continue or to work on I/O.

It sends INT ACK to D_1, if D_1 has request it will complete it else will move to D_2 ... (Priority $D_1 > D_2 > D_3$)

In this way it works.

Here interrupts are of 2 types.

i) Non vectored interrupt: Here there is a mutual understanding between CPU & device that where this routine is stored in memory (high priority device) (Well know friends/best friends?

ii) Vectored interrupt: Some interrupts may be vectored where the interrupting device will also tell the address that where this routine is stored in the memory. (Address should be informed)

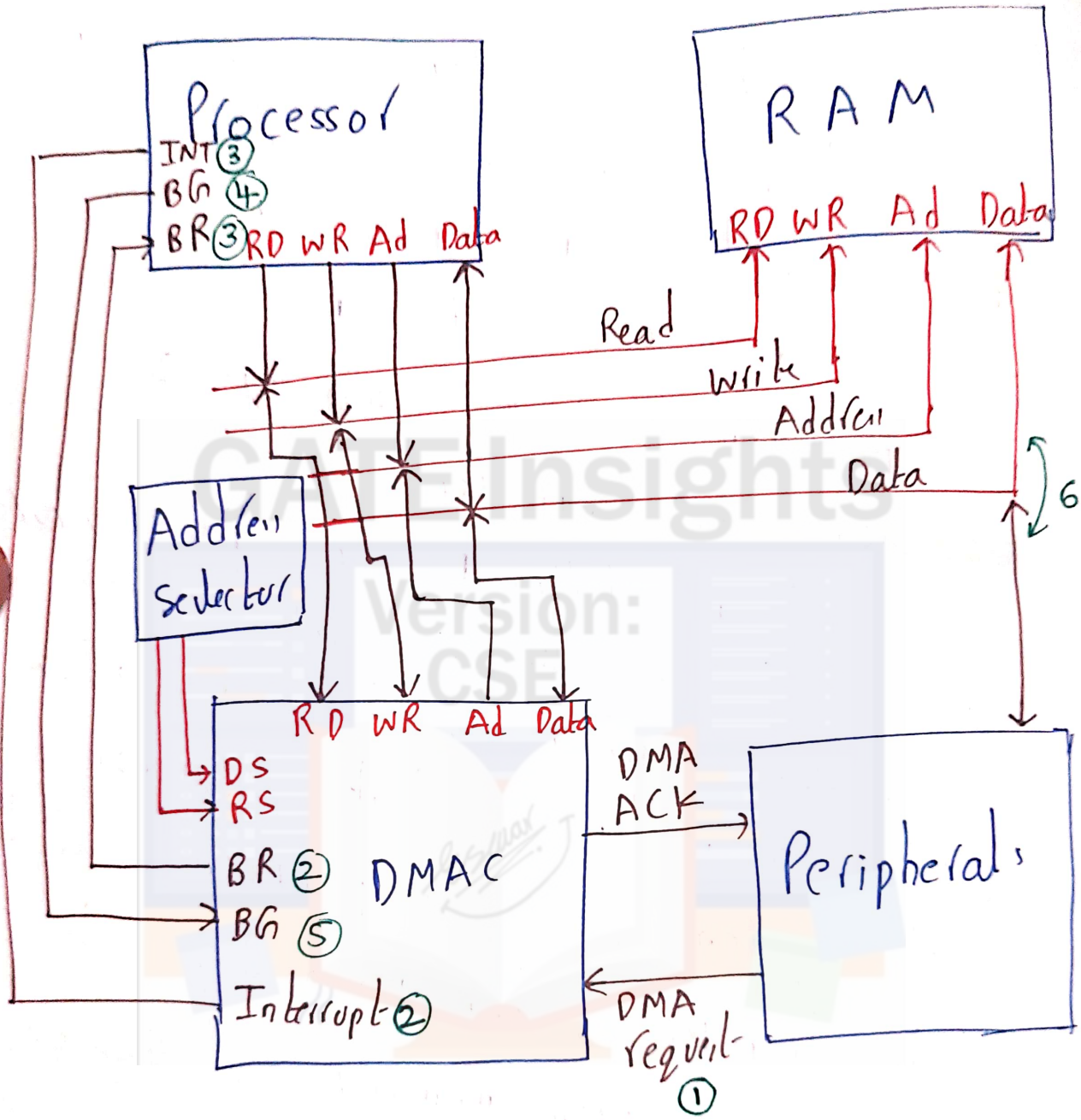
Adv: very simple, easy to use, easy to understand, relatively fast.

Disadv: here priority fixed & even in case of requirement we cannot change it.

III) Direct Memory Access (DMA)

When we want to perform i/o operations then the actual source or destination is either i/o device or memory but CPU is placed in between just to manage & control the transfer.

Hence DMA is the idea where we use a new device called DMA Controller using which CPU allow DMA controller to take control of System buses & perform direct data transfer either from device to memory or from memory to device.



BG - Bus Grant

BR - Bus Request

Modes of transfer:

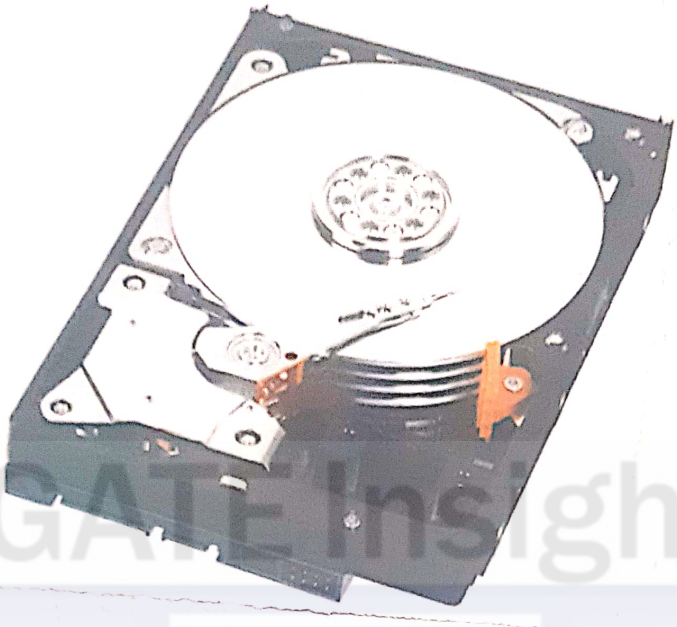
Burst mode: When entire I/O transfer is completed & the control comes back to CPU then it is called Burst mode.

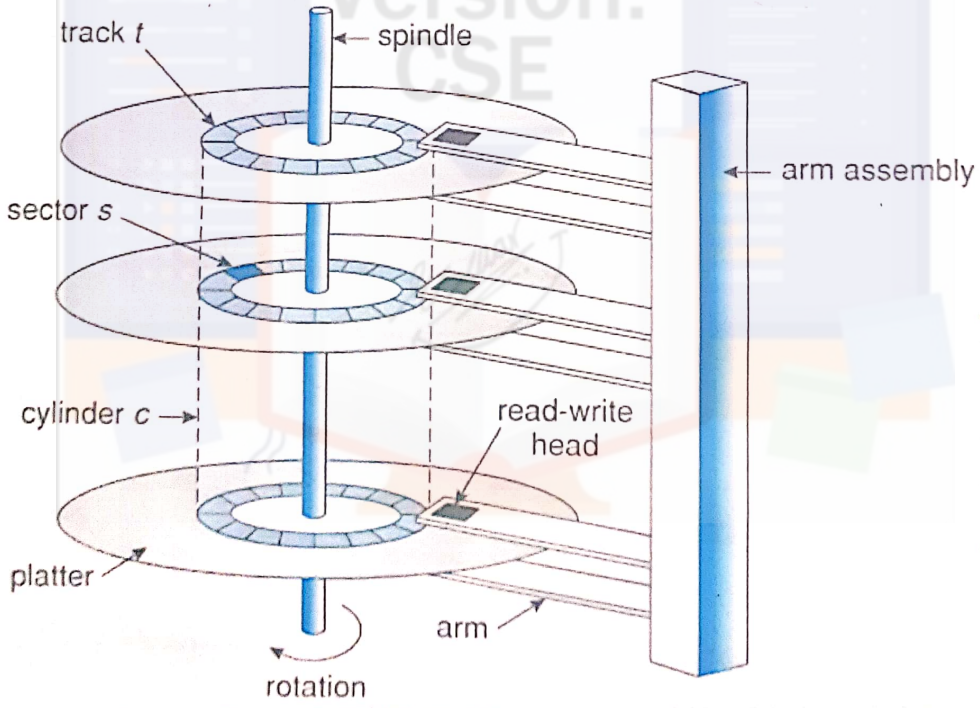
Cycle stealing mode: - When CPU executes an instruction then normally their could be following phase

IF: Instruction fetch
ID: Instruction Decode
OF: Operand fetch
IX: Instruction Execute
WB: Write back.

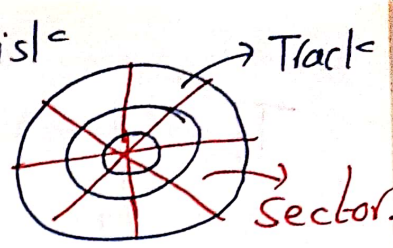
Here ID & IX phase CPU don't require system buses & if only in that time control is giving to DMA then it is called cycle stealing.

→ Mass storage structure:





→ Spindle holds the disk (platters) which are stacked one on other



→ Spindle rotates at high speed. (counter clock wise).

→ Surface of platter are covered by magnetic material.

→ Both surfaces are coated

→ Data is written/red with the help of read write head controlled by hand/arm assembly.

→ moving we can access different tracks.

→ These tracks are further divided into sectors.

surface no → Track no → sector no //

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

Seek Time: It is ~~a~~ the time taken by Read/Write header ~~during~~ to reach the correct track (Given in question).

Rotational Latency: It is the time taken by Read/Write header during the wait for the correct second.

In general it's a random value 0 to 360° rotation time.

So we take avg.

i.e half of complete time.

Transfer time: It is the time taken by read/write header either to read/write on a disk.

Total time = $\left(\frac{\text{File size}}{\text{track size}}\right)$ * time taken to complete one revolution

ex! disk speed = 360 rpm
track size = 512 bytes.
speed = ?
of data transfer.

(Gate 1995)

360 r → 1 min
360 r → 1 sec
60
6 r → 1 sec
1 r → $\frac{1}{6}$ sec.

$\frac{1}{6}$ s → 512 B
1 s → 3072 B
∴ 3072 B/sec.

ii) 15000 rpm

data transfer rate: 50×10^6 byte/sec

seek time = 2 * Rotational delay.

control transfer = 10 * disk transfer time.

read/write 512 byte sector

15000 r → 1 min

$\frac{\cancel{1500}^{250}}{\cancel{6}}$ r → 1 sec.

250 r → 1 sec.

1 r → $\frac{1}{250}$ sec

1 r → 4 msec

$RL = \frac{4}{2} = 2$

$ST = 2 * 2 = 4$

disk Transfer time = ~~512~~

50×10^6 B — 1 s

1 B — $\frac{1}{50 \times 10^6}$ sec.

$512B \longrightarrow \frac{512}{50 \times 10^6}$ sec.

$\longrightarrow \frac{512 \times 10^{-3}}{5 \times 10^6}$ msec

$\Longrightarrow 0.01024.$

$T.T.T = ST + RL + TT + C.T$

$= 4 + 2 + 0.01024 + 10 * 0.01024$

$= 6.11148$

6 ms

→ Pipelining :-

If the system has only one processor then at most one instruction can be executed at a time.

If we really want to execute multiple instructions together or concurrently then we must have multiple processors

→ Pipelining is a phenomena or method using which we will be able to run more than one instruction at the same time, on a single processor

ex: Asume we 2 instructions & each instruction has 5 phases

(5 phases) each phase takes 1 cycle.

- Instruction Fetch
- Instruction Decode
- Operand Fetch
- Instruction executes
- Instruction Store / Write back.

Instruction	1					2				
Fetch	✓					✓				
Decode		✓					✓			
Fetch			✓					✓		
execute				✓					✓	
Write					✓					✓
clock	1	2	3	4	5	6	7	8	9	10

For 2 instructions without pipelining it took 10 clocks

⇒ no. of instructions * no. of phase

⇒ (2 * 5) = (10) //.

with pipeline:

Instruction	1	2				
fetch	✓	✓				
Decode		✓	✓			
fetch			✓	✓		
execute				✓	✓	
write					✓	✓
clock	1	2	3	4	5	6

For 2 instructions with pipelining it took 6 clocks

⇒ no. of phase * + (no. of instruction − 1) * time of one clock //.

ex: 10 instructions 4 phase

without pipelining

$10 * 4 \rightarrow 40$

with pipelining

$4 + (9 * 1)$

$\rightarrow 13$

$$\text{Speed up} = \frac{\text{Time without pipelining (w.Twp)}}{\text{Time with pipelining (Tp)}}$$

$$= \frac{40}{13} = 3.076$$

Max Speed up = no. of stage = 4 //.

$$\text{Efficiency} = \frac{\text{speed up}}{\text{max speed}} * 100 = 76.92\%$$

1000 instruction 4 phase

$Twp \Rightarrow 1000 * 4$

$\rightarrow 4000$

$Tp \Rightarrow 4 + 999$

$\Rightarrow 1003$

$$\text{Speed up} = \frac{4000}{1003} = 3.988$$

Max speed up = 4

$$\text{efficiency} = \frac{3.988}{4} \times 100 = 99.7\%$$

∴ As no. of instruction increase efficiency also increase

Ideal state which we want is 1 clock per instruction.

if we assume

m - no. of stages

n - no. of instruct

$$\text{speed up} = \frac{m * n}{m + (n-1)}$$

if m value is tooo big m can be neglected

$$= \frac{m * \cancel{n}}{\cancel{n}} = m$$

Note that here we assumed each phase take 1 cycle but it reality it's not.

ex: i) 5 instructions 4 phase

	F	D	E	WB	Total
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

$T_{wp} = 32$

D ✓ D ✗ → stall (can't execute) (wait)

T_p =

	1	2	3	4	5	6	7	8	9	10	11	12
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
I_5					X	F	X	X	X	D	D	X

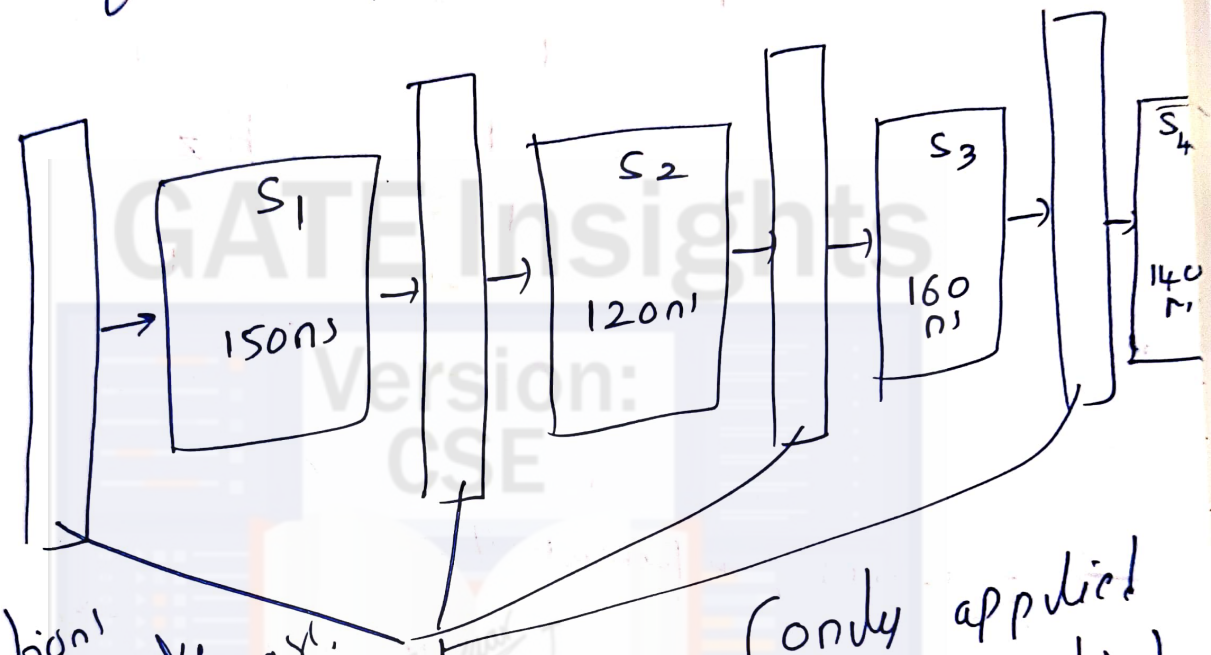
	13	14	15	16	17
I_1					
I_2					
I_3					
I_4	WB				
I_5	E	WB	WB		

Speed up $\rightarrow \frac{T_{wp}}{T_p} = \frac{32}{15}$

$\rightarrow 2.17$ //

$$\eta = \frac{2.17}{4} \times 100 = 53.33\ \%$$ //

ii)



no. of instructions not given take ideal case.

Delay 5 ns (only applied while pipelining)

$T_{wp} = 150 + 120 + 160 + 140 = 570$ nsec.

$T_p = \max(S_1, S_2, S_3, S_4) + \text{delay}$

$S_1 = 155$, $S_2 = 125$, $S_3 = 165$, $S_4 = 145$

5 added as delay in their

which is max 165 ns //

↑

hence it is the T_p //

If frequency asked

$$F = \frac{1}{T}$$

F → Hz

$$F = \frac{1}{165 \text{ ns}}$$

$$F = \frac{1}{165} \times 10^9$$

$$= \frac{1}{165} \text{ GHz} \quad (\rightarrow 10^9)$$

or

$$6.06 \text{ MHz} \quad (\rightarrow 10^6)$$

→ Hazards/dependency :-

Sometimes we cannot execute instructions with full efficiency in a pipeline because of certain dependency or hazards.

3 types

i) structural hazard
ii) control ~~hafd~~ hazard
iii) Data hazard

I) Structural hazard :

Even by having n stages in pipeline processor and assuming that the combinational circuit of every stage is different many time there will be dependency because of external hardware parts from CPU like system bus, memory etc.. coz of this stalls will occur which decrease speed up.

Structural hazard cannot be removed by programming. The only solution is replicate the hardware components (costly)

ex 4 phase with pipeline

- Fetch & decode
- Data read
- ALU
- Data write.

(need memory access)

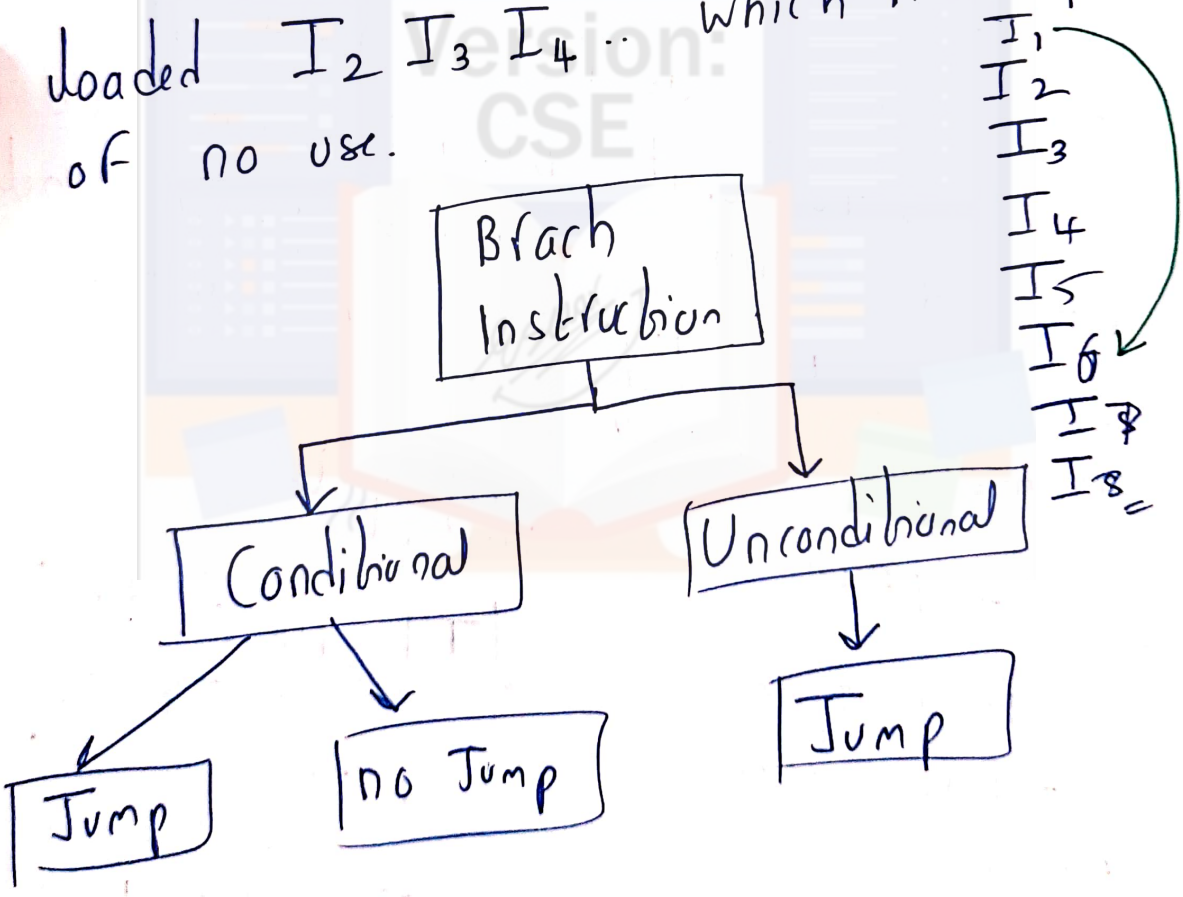
IF	DR	ALU	DW			
	X	IF	X	DR	ALU	DW
1	2	3	4	5	6	7

→ per 2 instruction due to SH structural hazard !!

II) Control Hazards :-

If some instruction I_1 is branch instruction & after executing the entire instruction we understood that there is jump & next instruction to be executed is I_{10}

This time we have partially loaded I_2 I_3 I_4 .. which is a problem of no use.



eg: consider 20% of instructions are branch instructions

assume CPI = 1

now what is CPI =?

if Branch instruction = 1 stall

$\Rightarrow \underbrace{(0.8 \times 1)}_{\text{non branch}} + (0.2 \times 2)$ (1 + 1)

$\Rightarrow 0.8 + 0.4 \Rightarrow 1.2$ CPI

if BI = 2 stall

$\Rightarrow (0.8 \times 1) + (0.2 \times 3)$ (2 + 1)

$= 1.4$

if BI = 3 stall

$\Rightarrow (0.8 \times 1) + (0.2 \times 4)$

$= 1.6$

Best solution for this is code rearrangement

III) Data Hazards: (Gate 2010)

ex

Instruction		meaning
3PO ← MUL	R2, R0, R1	R2 ← R0 * R1
6PO ← DIV	R5, R3, R4	R5 ← R3 / R4
1PO ← ADD	R2, R5, R2	R2 ← R5 + R2
1PO ← SUB	R5, R2, R6	R5 ← R2 - R6

5 stages
1 ← IF – Instruction fetch
1 ← ID – Instruction decode
1 ← OF – Operand fetch
(Based on op) PO – Perform operation
1 ← WO – Write Operand.

(without forwarding) (from memory)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IF	ID	OF	PO	PO	PO	WO									
.	IF	ID	OF	X	X	PO	PO	PO	PO	PO	PO	Wo			
.		IF	ID	~~OF~~	X	X	X	X	X	X	X	X	~~OF~~	~~PO~~	WO
.			IF	ID	~~OF~~	X	X	X	X	X	X	X	X	X	X

17	18	19	20	21
OF	PO	WO		

Note as their is dependency

$R_2 \leftarrow R_5 + R_2$

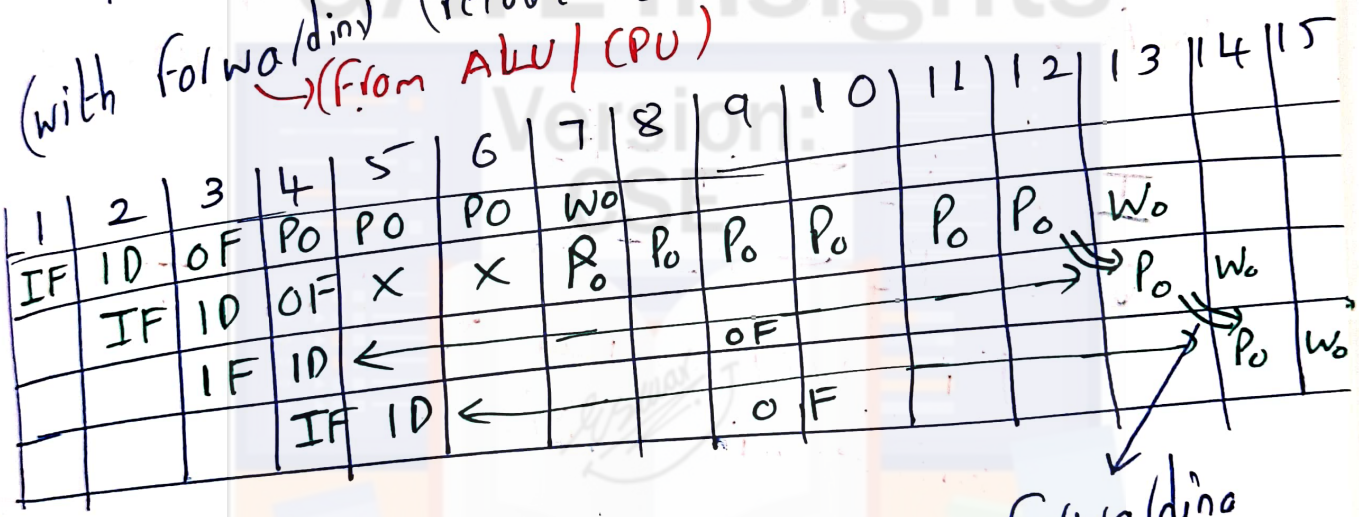
Both result will be ready after 13

$R_5 \leftarrow R_2 - R_6$

Both result will be ready at 15 without forwarding.

This approach is (with forwarding) (result taken ahead).

(From ALU / CPU)



as a simple logic

$I_2 \Rightarrow R_5 = R_3 / R_4$ (result will be ready at 13 clock).

hence in 14 clock we can perform I_3 by taking data ALU/CPU instead of memory (This is forwarding logic).

Data hazard occur when instructions that exhibit data dependence, modify data in different stages of pipeline

There are Bernstein condition

RAW (Read after Write) (Flow / True data dependency)
WAR (Write after read) (Anti Data dependency)
WAW (Write after write) (Output data dependency)

I :- $R_2 \leftarrow R_1 + R_3$
J :- $R_4 \leftarrow R_2 + R_3$

J is trying to read R_2 before I have written it (RAW).

I : $R_2 \leftarrow R_1 + R_3$
J : $R_3 \leftarrow R_4 + R_5$

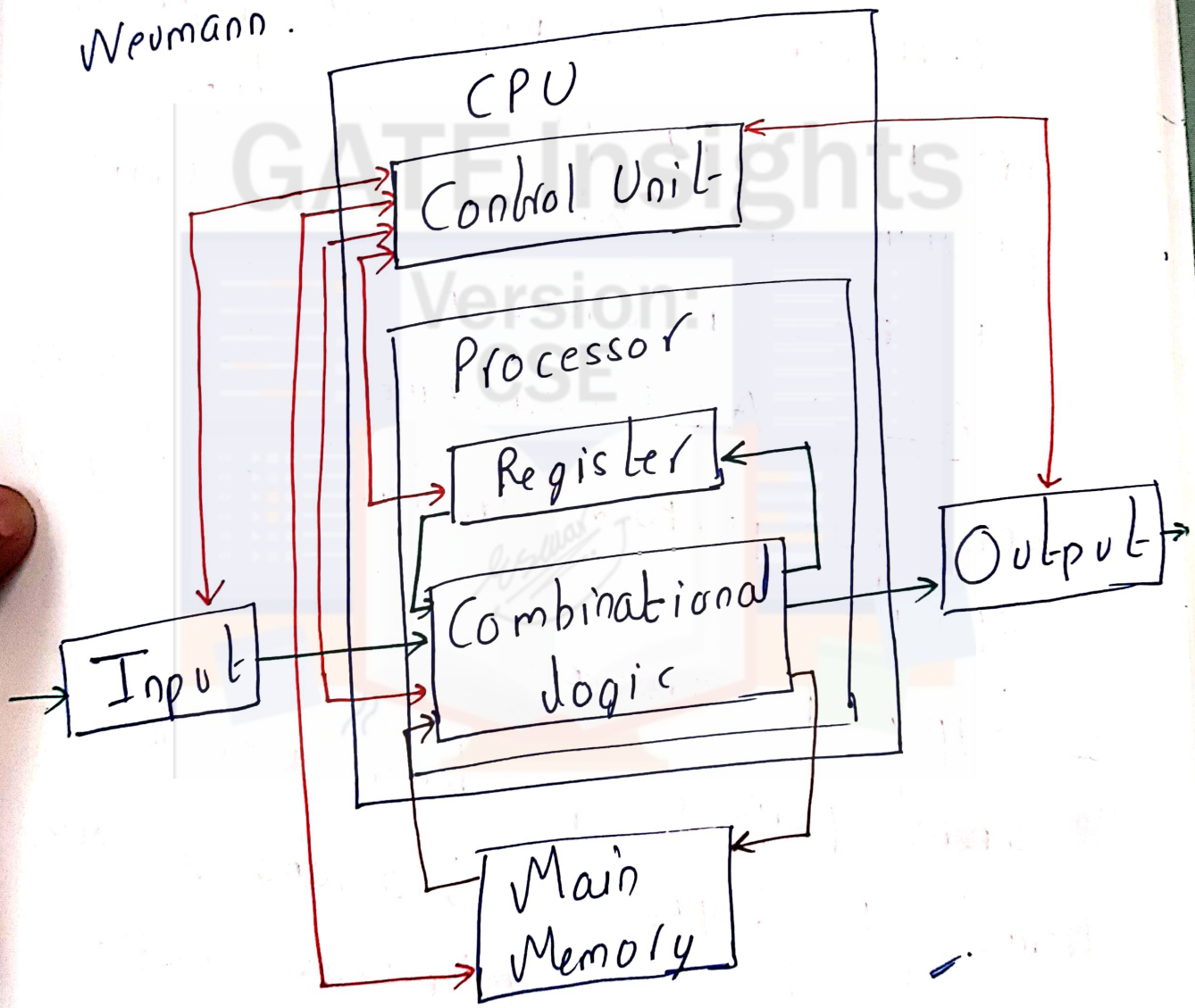
J trying to write R_3 before I read it (WAR).

I : $R_2 \leftarrow R_1 + R_3$
J : $R_2 \leftarrow R_4 + R_5$

J trying to write R_2 before I (WAW)

→ Von neumann architecture :-

The Von Neuman architecture is a computer architecture based on a 1945 description by John von Neumann.



3 main components:

Control Unit: Which generate control signal (master generator) to control every other part of the CPU.

Registers: few important registers are in every processor like program counter (which consist address of next instruction).

ALU: ALU (Arithmetic logic unit) which can perform arithmetic & logical operations.

2 type of Computer

General purpose	Dedicated device
Normal PC, Lappy, mobile etc..	Washing machine, microwave, ~~robot~~, ATM machine,

operation performing flow:

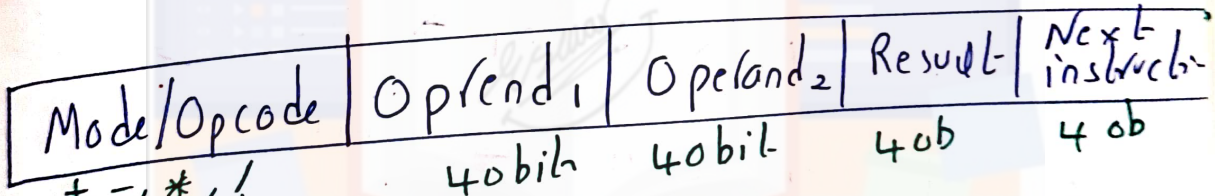
memory → register → ALU (do operation)
(Bring values)
↓
register
↓
memory //.

→ Instruction Format:

5 types

I) 4 Address Instruction:

4 address & 1 opcode/mode.

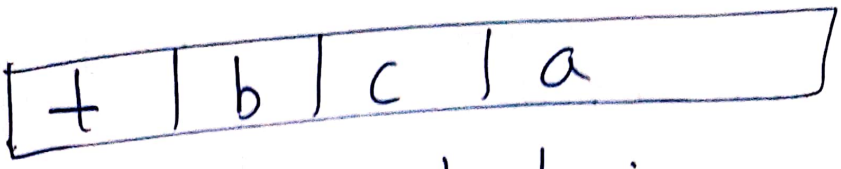


eg

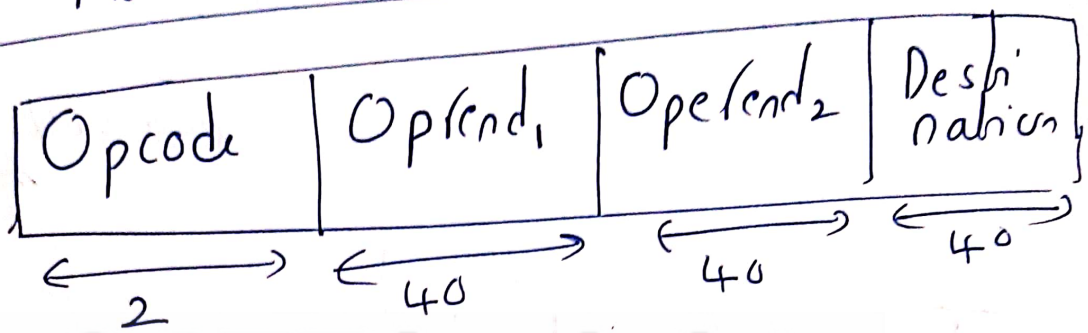
Adv: → Very simple, easy to use easy to understand.

Disadv: → Very lengthy & occupies a lot of space.

$$a = b + c$$



II) 3 Address Instruction:



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

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

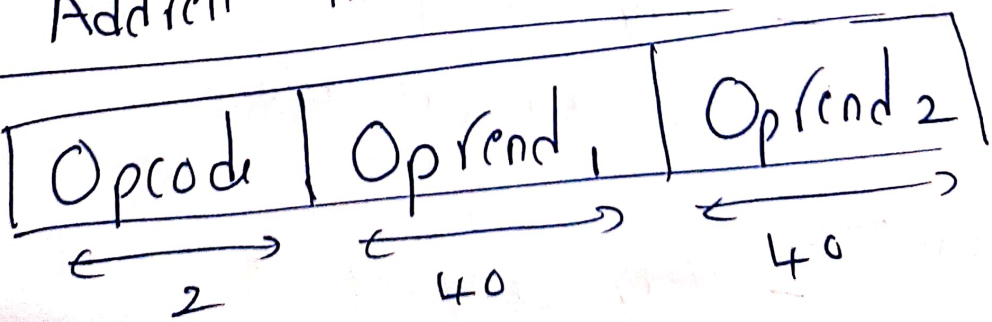
$R_2 \leftarrow M[C] + M[D]$

$M[X] \leftarrow R_1 * R_2$

Adv: Better than 4 address

Disadv: still lengthy.

III) 2 Address Instruction:



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

$R_1 \leftarrow M[A]$

$R_1 \leftarrow R_1 + M[B]$

$R_2 \leftarrow M[C]$

$R_2 \leftarrow R_2 + M[D]$

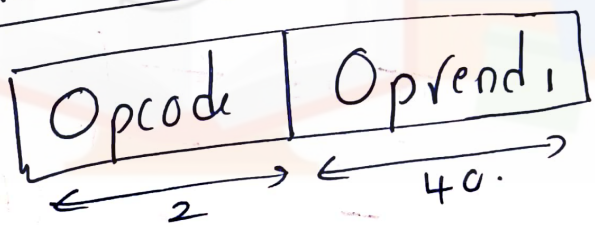
$R_1 \leftarrow R_1 * R_2$

$M[X] \leftarrow R_1$

Adv:- Well length, more efficient, no. of registers low.

disadv:- code optimization is relatively difficult.

IV) 1 Address Instruction:-



AC (Accumulator is used).

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

$AC \leftarrow M[A]$

$AC \leftarrow AC + M[B]$

$M[T] \leftarrow AC$

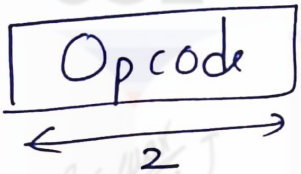
$AC \leftarrow MC$

$AC \leftarrow AC + M[D]$

$AC \leftarrow AC * M[T]$

$M[X] \leftarrow AC$

V) 0 Address Instruction:



Stack is used here.

$TOS \leftarrow A$

$TOS \leftarrow B$

$TOS \leftarrow (A+B)$

$TOS \leftarrow C$

$TOS \leftarrow D$

$TOS \leftarrow (C+D)$

$TOS \leftarrow (C+D) * (A+B)$

$M[X] \leftarrow TOS$

→ Addressing Mode :-

It specifies the different ways possible in which reference to the operand can be made.

Effective address :- It is the final address of the location where the operand is stored.

many types

I) Immediate mode addressing :-

It means the operand is itself part of instruction

MVI B, 30H

~~location~~ 30 data move to B.

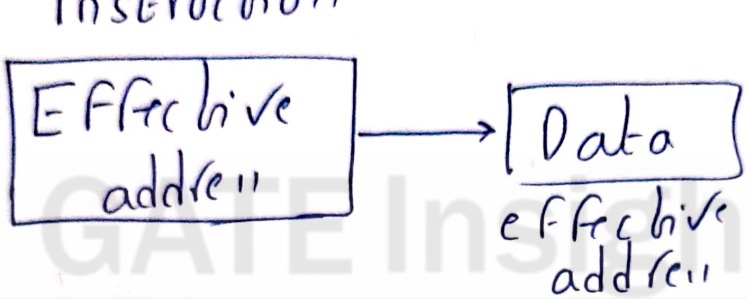
Adv :- Can be used for constants, where values are already known

Fast & easy.

Disadv :- Cannot access full space as its immediate it's size is limited.

II) Direct mode addressing:

It means instruction contains address of the memory where variable or data is present.



Adv:

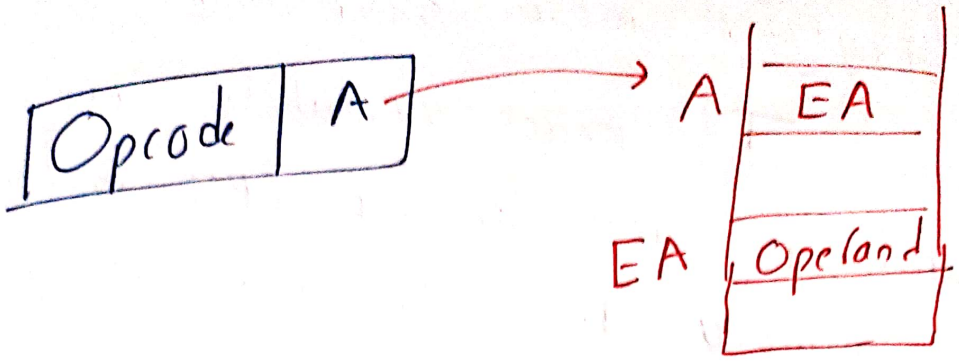
→ Best for unknow variables
→ No restricted range.

Dis Adv:

→ relatively slow compared to immediate mode.

III) Indirect mode Addressing:

Here in the instruction we store the address where the effective address is stored using which we can access the actual data.



Adv:
→ No limitation on no. of variable/size
→ Implementation of pointer are feasible

Disadv:
→ Relatively slow.

IV) Implied mode addressing :-

This mode is mostly used with 0-address instruction which means we only tell the operation & system know location

eg: AC (Accumulator)

eg: increment, HLT.

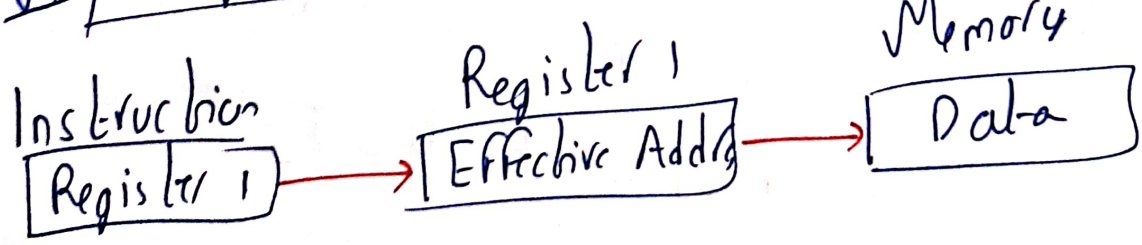
V) Register mode addressing:

It means variable are stored in register of CPU instead of memory. In instruction we give the register no.

Adv:
→ low no. of bit (as min no. of register exist)
→ extremely fast

Disadv:
→ few variable can only be stored.

VI) Register indirect mode:



VII) Base register (offset) mode

Here we save the starting address of the program in a register called base register.

effective address = base address + offset (instruction).

Now the advantage is even if we try to shift process in the memory, we only need to change the content of the base register.

VII) Index addressing mode :- (large array access).

Base address of the array ~~(in~~ in the instruction & the index which we want to access will be there in the register.

location : Base + Index.

Reduced Instruction Set Computer (RISC)

→ Used commonly & powerfull

→ few instructions
→ few address modes
→ Faster (hence hardwired)
→ all operations done with in the registers of the CPU.
→ Memory access limited to load & store
→ Fixed length easily decoded instruction.

Complex Instruction Set Computer (CISC)

→ large number of instructions, typically 100 to 250 instructions.
→ Some instructions that perform specialized tasks & are infrequently used.
→ a large variety of addresing modes typically from 5 to 20 different modes.