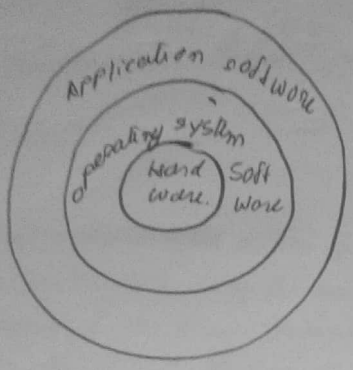
24/7/19

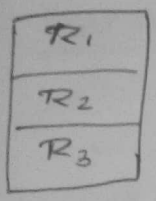


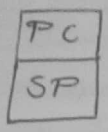
Particular program → Intermediate code or Assembly lang. code

```
int a,b,c;
scanf("%d,%d",&a,&b);
c=a+b;
printf("%d",c);
```

→

```
IN PORT A (INPUT PORT A)
LDA (Loading accumulator)
IN PORT A
LDM B (Loading into memory address B)
ADD B (Acc ← Acc + B)
MOV Acc C
OUT PORT B
```





* Program counter :- holds the address of the next instruction.

* Stack Pointer :- Stores the result of the previous PC

IR → Decoded → Timing & control unit → control signals.

Instruction :- holds the opcode → in assembly lang. MOV R_1, R_2 (MOV = opcode.)

in machine language

- MOV R_1, R_2

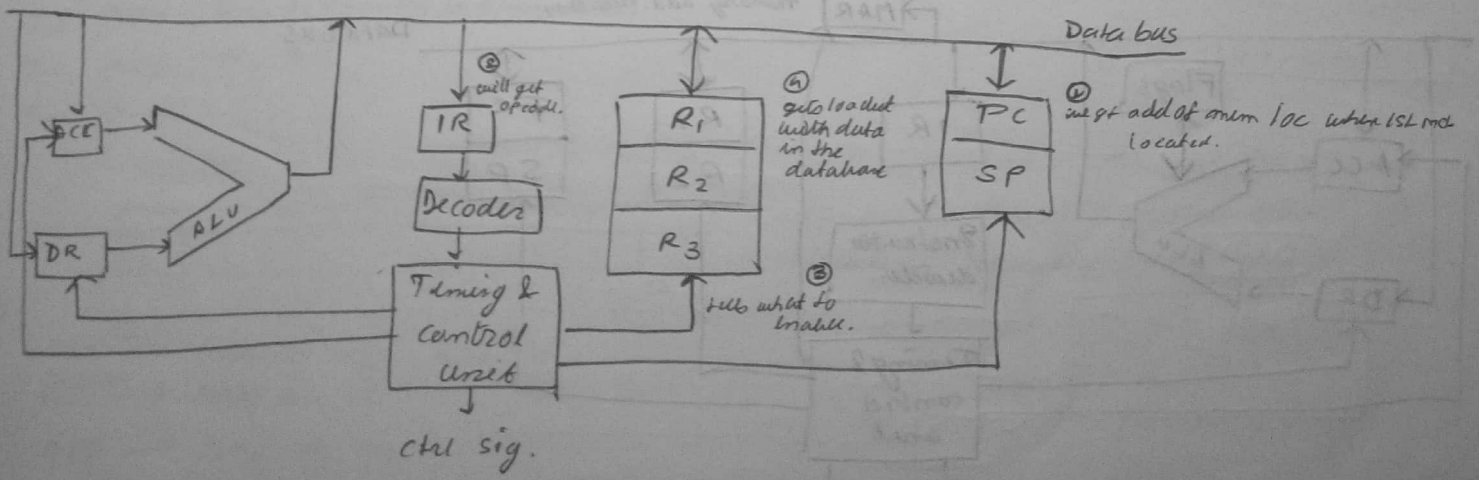
control signal: OE R_2, LD R_1

read.

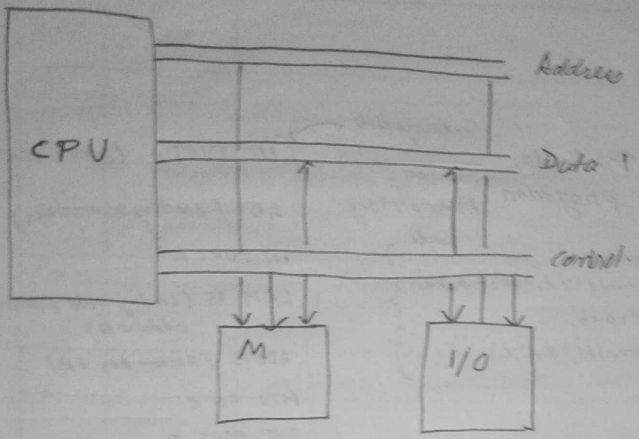
output enabled → to be stored in database

input enabled → reads the info from database.

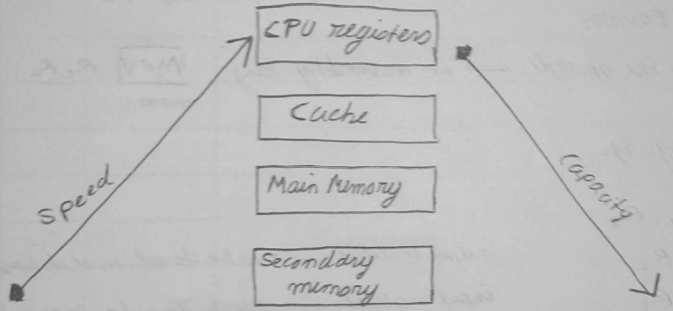
(so that R_1 does not read previous info. in the database)



Scanned by CamScanner



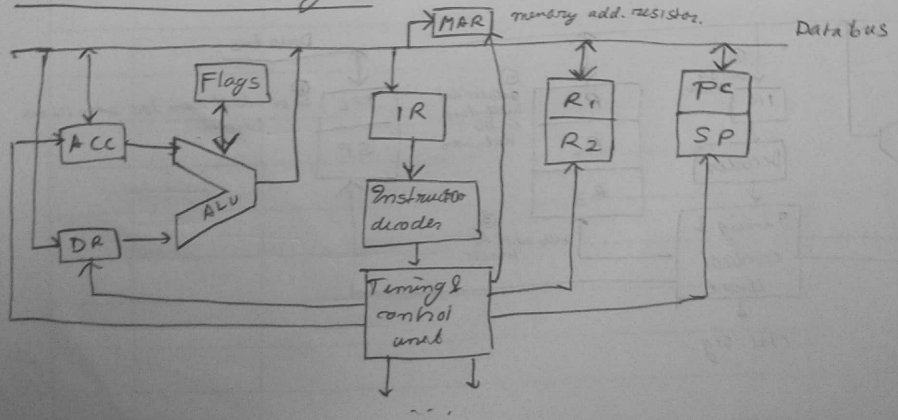
* Memory hierarchy



Info needs to be transferred to main memory from secondary memory.

25/7/19

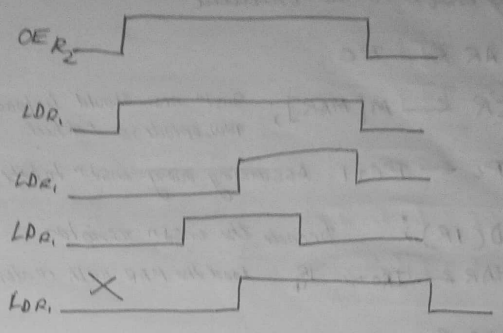
* Simple CPU Design



Scanned by CamScanner

MOV R_1, R_2 — contents of R_2 moved to R_1

generated { OE_{R_2}, LD_{R_1}



NP: because of the falling point of LD_{R_1} some other Register may be output enabled and R_1 may receive unwanted any info.

The timing & ckt unit makes care of this

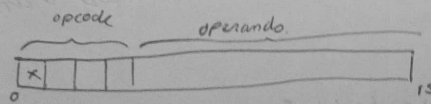
Instruction Read

→ fetch opcode
→ opcode decode
→ execute inst^n.

I_0 { ADD R_1 → repeated addition ⇒ multiplication
I_1 { CMP → has nand (uses flip-flops) performs ~~fraction~~ subtraction
I_2 { AND R_1 → Logical & NAND from CMP takes care of other logical op.

I_3: JMP
I_4: MOV R_1, R_2
I_5: MOV R_2, R_1
I_6: MOV Acc, R_1
I_7: MOV Acc, R_2
I_8: MOV R_1, Acc
I_9: MOV , R_2, Acc
I_{10}: MOV Acc, M (memory to accumulator)
I_{11}: MOV M, Acc
I_{12}: MVI

[Acc ← $\overline{\text{Acc}}$]



Instruction register.

Eg:
111 → register ref. inst^n =
~~001~~ → MOV Acc, M
010 → MOV M, Acc
011 → JMP

Scanned by CamScanner

* For any instⁿ to be executed

T0: MAR ← PC

T1: { IR ← M[MAR]; instⁿ reg. should be loaded with the instⁿ in MAR now, opcode is fetched.
PC ← PC+1 Assuming every instⁿ to take only 1 memory location

T2: { DCD(IR); Decode the instⁿ register
MAR ← IR0 ... IR11 Load the MAR with content from IR0:11

T3: ADD R1
DR ← R1 DR is loaded with contents of R1 so that ALU can access it.

T4: ACC ← ALU_ADD (ACC, DR), TO (result is given down to ALU)
(Time stamps)

move back to T0 (time stamp)
next instⁿ fetched.

MOV R1, R2

T3: R1 ← R2

next T0.

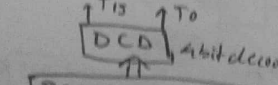
MOV ACC, M

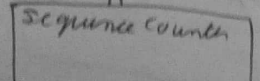
T3: ACC ← M[MAR] Acc gets the contents of the memory address which is already loaded to IR0:11 (meaning loaded to MAR)
T0;

#

Sequence counter (4 bit) ← its size depends on complexity of instⁿ

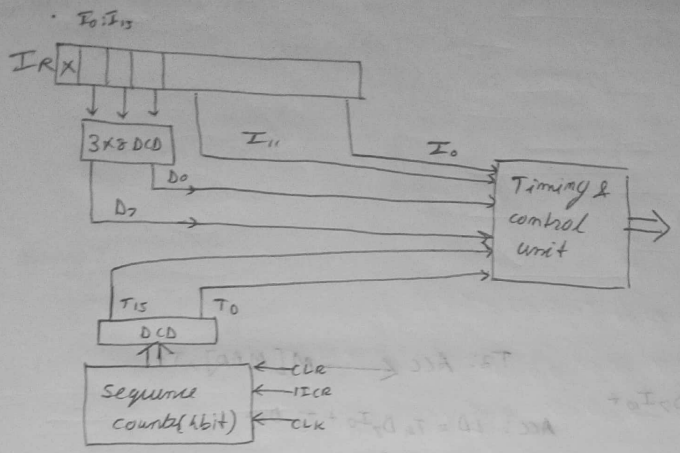
16 time stamps. ← Assuming that the instruction will not take more than 16 bit of time stamps.





← CLR every time we go back to T0

Scanned by CamScanner



* Usually in 8085 machine the 0 address has a Jump instⁿ
Jumps to application area

Timing stamp T_0

PC : $OE = T_0 +$ (enabled)

MAR : $LD = T_0 +$ (enabled)

Timing stamp T_1

PC : $INCR = T_1 +$

IR : $LD = T_1 +$

Timing stamp T_2

MAR : $LD = T_2 +$

IR : $OE = T_2 +$

For ADD R_1 , Timing stamp T_3

DR : $LD = T_3 D_7 I_0 +$

R_1 : $OE = T_3 D_7 I_0 +$

For MOV ~~ACC,~~ R_2, R_1 , Timing stamp T_3

~~ACC~~ R_2 : $LD = T_3 D_7 I_5 +$

R_1 : $OE = T_3 D_7 I_5 +$

T_4,

$ACC = T_4 D_7 I_0 +$

$ALU = ADD = T_4 D_7 I_0 +$ } register ref. instⁿ (DCD) — corresponds to D_7.

SEQCNTR : $INCR = \overline{T_0 + T_1 + T_2 + T_3 D_7 I_0 +}$ ~~$T_4 D_7 I_0 +$~~

$CLR = T_4 D_7 I_0 +$

Scanned by CamScanner

For MOV R_1, R_2,

R_1: $LD = T_3 D_7 I_{14}+$

R_2: $DE = T_3 D_7 I_4+$

For MOV Acc, R_1

R_1: $OE = T_3 D_7 I_6+$

Acc: $LD = T_3 D_7 I_6+$

31/7/19

MOV Acc, M

T_3,

Acc: ~~$LD\ D_7 I$~~ $LD = T_3 D_7 I_{10}+$

~~MAR :~~.

T_3: $ACC \leftarrow M[MAR], T_0$

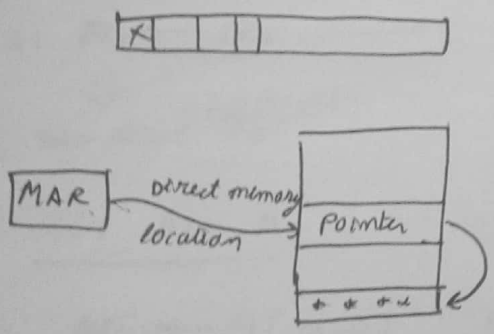
Acc: $LD = T_4 D_7 I_0 + T_3 D_1+$

~~ALU: ADD = $T_4 D_7 I_0+$~~

SEQCTR: INCR =

$CLR = T_4 D_7 I_0 + T_3 D_1+$

#



Indirect memory location.

MOV Acc, M

Acc: $LD = T_3 D_1 I_{15}'$

MOVI Acc, M

Acc: $LD = T_3 D_1 I_{15}$

(Indirect addr.)

★ Hardwire control Unit → Design — Proper circuitry is used to generate control signals.

Adv: Very fast

Disadv: Ctrl not scaled.

★ Microprogrammed ctrl unit design. → alternate adv: calculate as programming is allowed.

disadv: ∵ programming allowed. memory is used up and hence slow.

★ control memory (CM) → control words.

Scanned by CamScanner

In time stamp T_0,

LD_{MAR} — C_0 designated bit of CM

OE_{PC} — C_1 " 1st bit of CM

0 . . . . . 11 Q1

Branch add. (BA)

During T_1,

LD_{IR} — C_2

RD_M — C_3 (read memory)

$INCR_{PC}$ — C_4

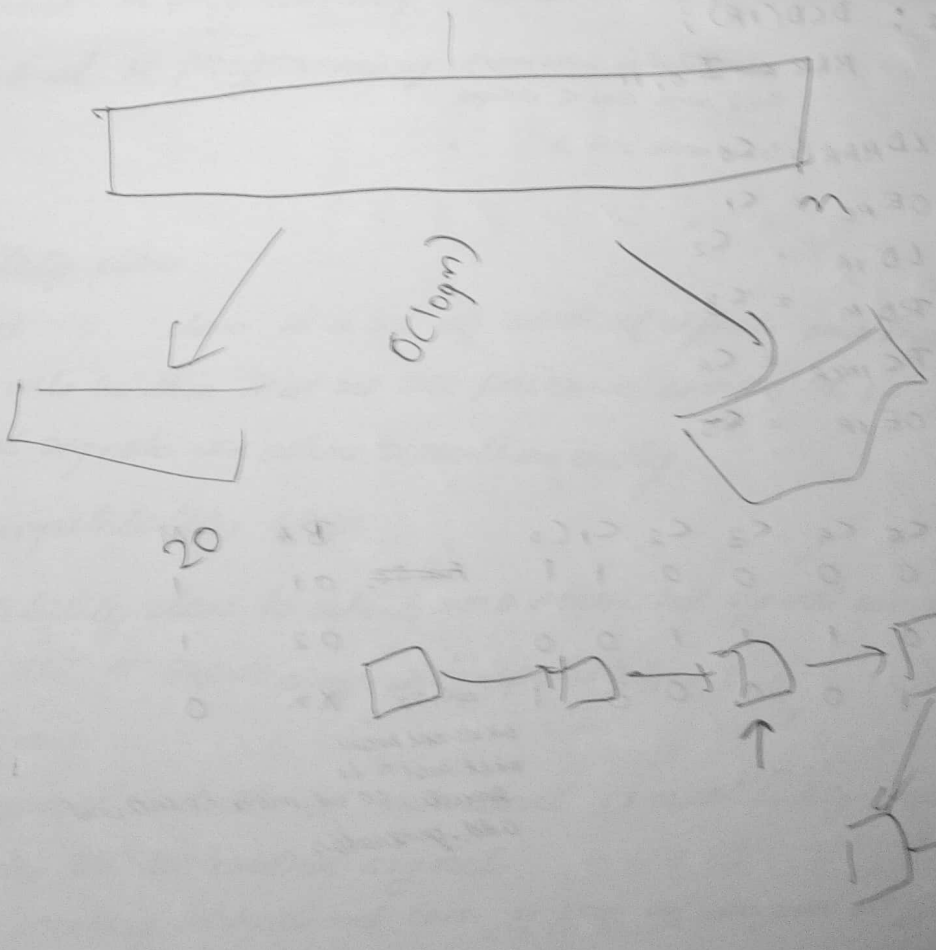
0 . . . . . 0111 00 Q2

During T_3

OE_{IR} — C_5

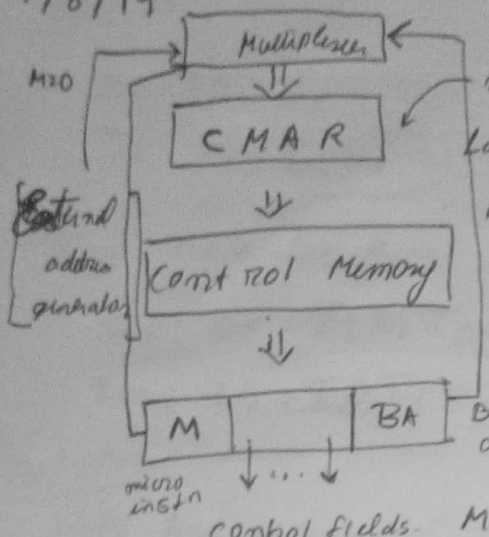
0 . . . . . 0100001 xx

sum(arr[ ])



Scanned by CamScanner

7/8/19



many inputs loaded

Load add. of memory location to be sent to ctrl mem.

M=0

Carries out ctrl signals. Except this, all others take part in decision making as to which instn to be executed next

Extend address generator

Branch address

M = Mode (acts as select line)

After opcode fetch & decode, we don't know which memory add to jump to to execute next instn.

Microprogram sequencer → all other units except the ctrl memory.

↓

+ control memory = Microprogrammed ctrl unit.

Timestamps;

T_0: MAR ← PC

T_1: IR ← M[MAR]; PC ← PC+1

T_2: DCD(IR);

MAR ← $I_{0:11}$

A

$LD_{MAR} = C_0$

$OE_{PC} = C_1$

$LD_{IR} = C_2$

$RD_M = C_3$

$PC_{INCR} = C_4$

$OE_{IR} = C_5$

★

	C_5	C_4	C_3	C_2	C_1	C_0		BA	M
For T_0,	0	0	0	0	1	1	~~For T_0,~~	01	1
For T_1,	0	1	1	1	0	0		02	1
For T_2,	1	0	0	0	0	1	after T_2, we do not know what instn to execute, so we move to internal add. generator.	XX	0

Scanned by CamScanner

Eg: MOV R_1, R_2

$C_7\ C_6\ C_5 \ldots\ldots$ BA Mode

1 1 0 ..... 0 0 0 1

$OE_{R_2} = C_6$

$LD\ R_1 = C_7$

Eg: ADD R_1

T_3: $DR \leftarrow R_1$

T_4: $ACC \leftarrow ALU_{ADD}(ACC, DR)$; T_0

$C_{12}\ C_{11}\ C_{10} \blacklozenge C_9\ C_8\ C_7\ C_6 \ldots\ldots$ BA Mode

0 .... 0 1 1 0 0 ..... 0 0 1

1 1 1 0 ..... 0 0 0 1

$LD\ DR = C_8$ $\quad ALU_{ADD} = C_{11}$

$OE_{R_1} = C_9$ $\quad OE_{ALU} = C_{12}$

$LD_{ACC} = C_{10}$

Micro instruction design

- Horizontal μ programming — signal inst^n to be decoded / encoded — n, no of bits $[\log_2 n]$
- Vertical μ programming — multiple inst^{n}s to be decoded / encoded
- Diagonal μ programming (somewhere in between) — requires > bits than VMP, < bits than HμP

★ Compatibility class

The ~~multiformati~~ class is a set of control signals such that the ctrl signals within that set are pairwise compatible i.e, no two ctrl signals are active simultaneously.

★ Maximal compatibility class

It is a compatibility class to which no other ctrl signal can be added without introducing incompatibility.

★ Minimal cover.

It is a minimal subset of maximal compatibility class, which includes all the control signals.

(We will go for encoding in Minimal cover in case of diagonal μP).

Scanned by CamScanner

Control signals

I_1 : a b c g

I_2 : a c e h

I_3 : a d f

I_4 : b c f

a, b, c, d, e, f, g, h — control signals input.

S_1 : a, b, c, d, e, f, g, h — write all control signals

S_2 : bd, be, bh, cd, cf, cg, dh, ef, eg, fg, fh, gh

pair of signals which are never together

S_3 : make a group of 3 ctrl signals

bde, bdh, efg, bfgh, cde, cdg, cdh, def, bgh, dgh, deg, fgh, cfg

S_4 : 4 ctrl signals together

ϕ

o/p/n

cover table

	a	b	c	d	e	f	g	h
K_1 = a	×							
K_2 = cd			×	×				
K_3 = bde		×		×	×			×
K_4 = bdh		×		×				×
K_5 = deg		↓		×	×		×	
K_6 = dgh		dominated		×			×	×
K_7 = efg				dominating		×	×	
K_8 = fgh						×	×	×

dominated by a subset of the dominating

essential cover

dominating

Strike off dominating columns

So, K_1, K_2 are essential covers

Scanned by CamScanner

Reduced covertable

	b		e	f		h
K_1 = bde	×		×			
K_2 = bdfh	×					×
K_3 = deg			×			
K_4 = dfgh						×
K_7 = efg			×	×		
K_8 = fgh				×		×

we will strike off dominated rows and not dominating rows.

minimal cover = $\{(K_1, K_2), K_3, K_8\}$

or

$\{(K_1, K_2), K_4, K_7\}$

⇓

"all the control signals have been covered"

we need to choose these in such a way that all the alphabets are covered and not struck off in the reduced covertable.

* Pipelining

Assuming 4 inst^{n}s executed

I_1
I_2
I_3
I_4

⇒ Following

EX				I_1			I_2			I_3			I_4			
OD			I_1			I_2			I_3			I_4				
OF		I_1			I_2			I_3			I_4					
IF	I_1			I_2			I_3			I_4						

16 time stamps reqd. for 4 instns. non pipeline method.

- inst^n fetch — IF
- operde fetch — OF
- operde decode — OD
- execution — EX

Best case scenario :-

EX				I_1	I_2	I_3	I_4	
OD			I_1	I_2	I_3	I_4		
OF		I_1	I_2	I_3	I_4			
IF	I_1	I_2	I_3	I_4				

(IF gets free once I_1 is fetched)

7 time stamps reqd.

Scanned by CamScanner

Worst case :-

Data dependency Hazards :

- RAW - read after write
- WAR - write after read
- WAW - write after write

I_0: MOV R_1, R_2 (R_1 written on)
I_1: ADD R_1 Acc ← Acc + R_1 (R_1 is now read)
I_2: MOV R_1, R_3 (R_1 is being written on)
I_3: ADD R_2, R_4

(I_0, I_1) → RAW dependency
(I_1, I_2) → WAR
(I_0, I_2) → WAW
(I_0, I_3) → WAR

ADD R_2, R_4
MOV R_5, R_4
} Both of them are reading from same source, this is not dependency.

Q. Find all possible dependencies present in the following program fragment when you are using a 4-stage pipeline.

I_0: MOV R_1, R_2 ① ③
I_1: ADD R_3, R_4 ③ ⑤
I_2: SUB R_1, R_5 ① ⑤
I_3: ADD R_2 ③
I_4: MOV R_5, R_6 ⑤ ⑥
I_5: ADD R_5 ⑤
I_6: MOV R_7, R_1 ① ⑦

Soln. (I_0, I_2) → WAW ~~(I_0, I_3) → X~~
(I_0, I_6) → RAW (I_2, I_4) → WAR
(I_2, I_6) → RAW ~~(I_2, I_5) → X~~
(I_4, I_5) → RAW

Scanned by CamScanner

* Assuming I₂ & I₃ is dependent on each other.

* To make sure the instⁿ that are depends on releases its products not before the one is executed

9/8/19

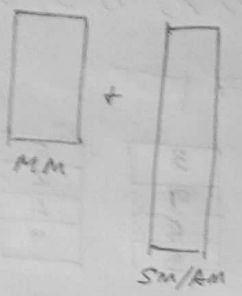
MEMORY MANAGEMENT

* Virtual Memory

made to believe that we have available

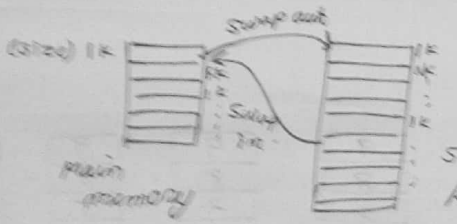
- Paging
- Segmentation

} using these we can apply the concept of virtual memory in our system.



Paging

"main mem. can be divided into same sized pages"



when there is a demand for mem. & the MM cannot provide it, then:

Swap out :- some page are swapped out from MM ⟹ SM

Swap in :- some page are swapped in from SM ⟹ MM (empty slot)

Choosing page to swap ⟶ on the basis of some Algorithm.

Page replacement algorithms

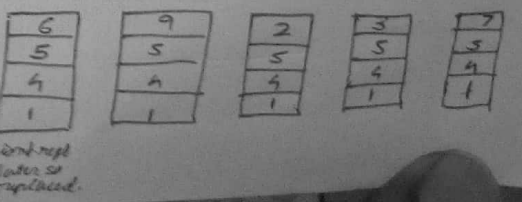
- FIFO (First in first out)
- LRU (Least recently used)
- MFU (Most frequent used)
- Optimal replacement

Eg: 1, 4, 5, 1, 6, 9, 4, 2, 3, 1, 7

page reference stream

Assuming ⟶ each page $\xrightarrow{\text{size}}$ 1K

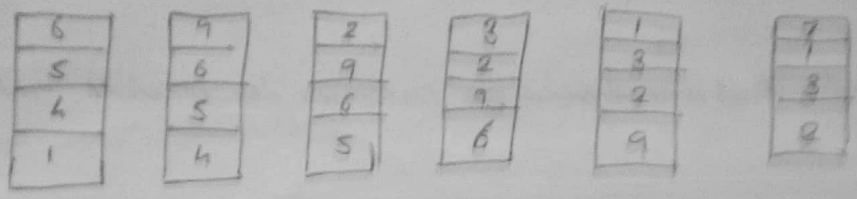
main memory $\xleftarrow{\text{accommodate}}$ 4 page.



Scanned by CamScanner

FIFO :=

(like queue)

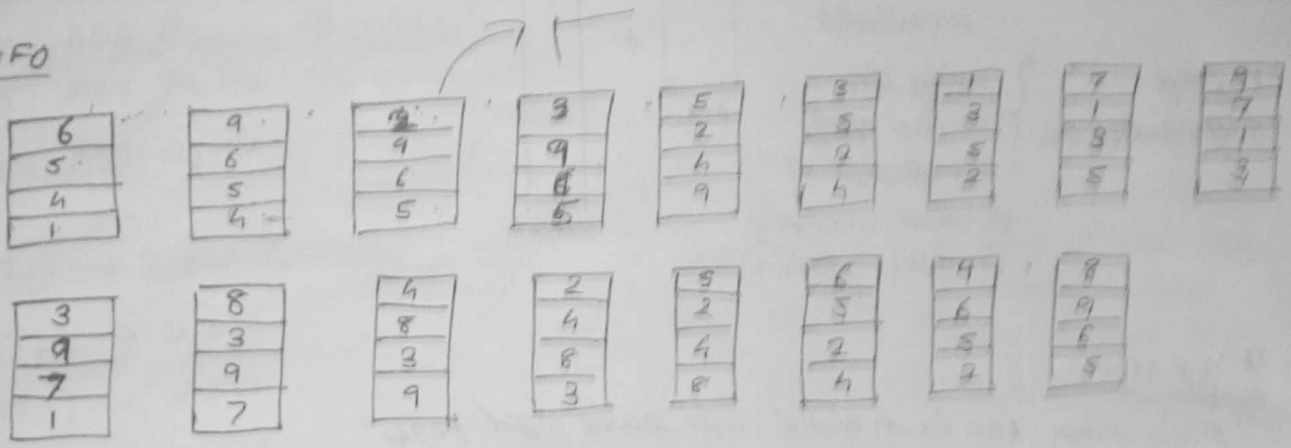


XRV :-

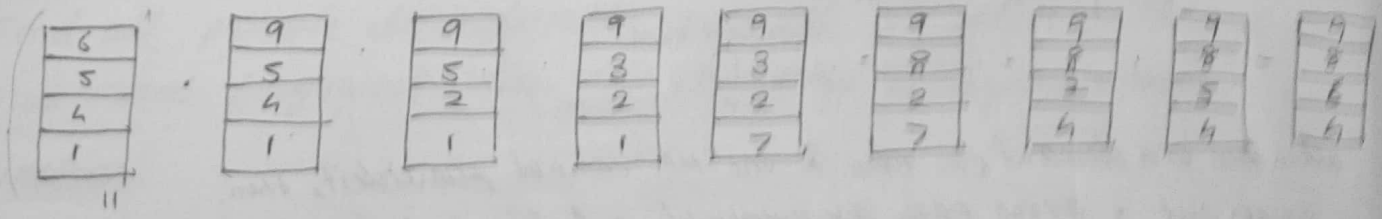
* No. of page misses.

1, 4, 5, 1, 6, 9, 4, 2, 5, 3, 1, 7, 9, 3, 8, 4, 2, 5, 6, 9, 8.

Soln- FIFO

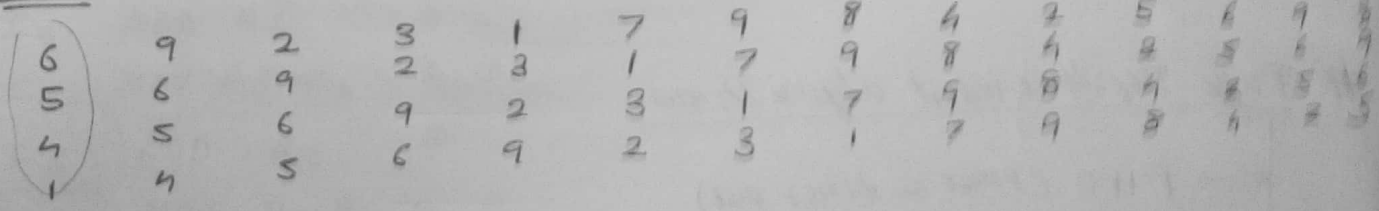


Opt. rep.



+8 +4 ⟶ 12

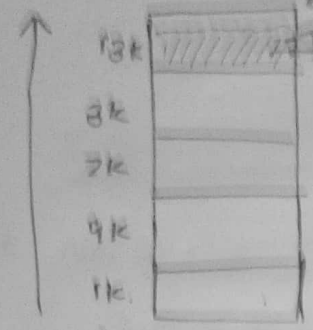
LRU FIFO



13+4 ⟶ 17

Scanned by CamScanner

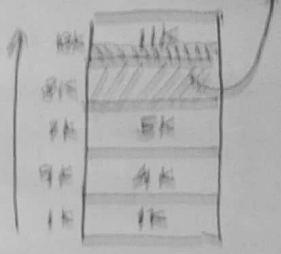
Segmentation



scanning from lower bits to higher.

- Best fit
- First fit
- Worst fit.

External fragmentation



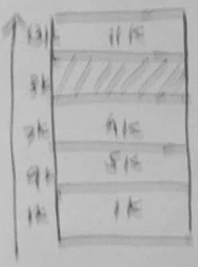
→ Best fit

aim :- to do minimum fragmentation.

[4k, 1k, 5k, 10k, 4k]

→ First fit

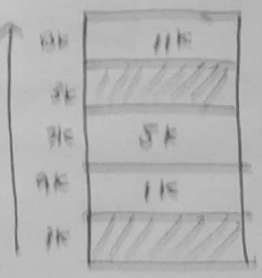
[11k, 1k, 5k, 10k, 4k]



→ Worst fit

aim :- maximize fragmentation

[11k, 1k, 5k, 10k, 4k]



★

DRAM (Dynamic RAM) (made of capacitors) → Main memory

SRAM (Static RAM) [costly] → Cache memory

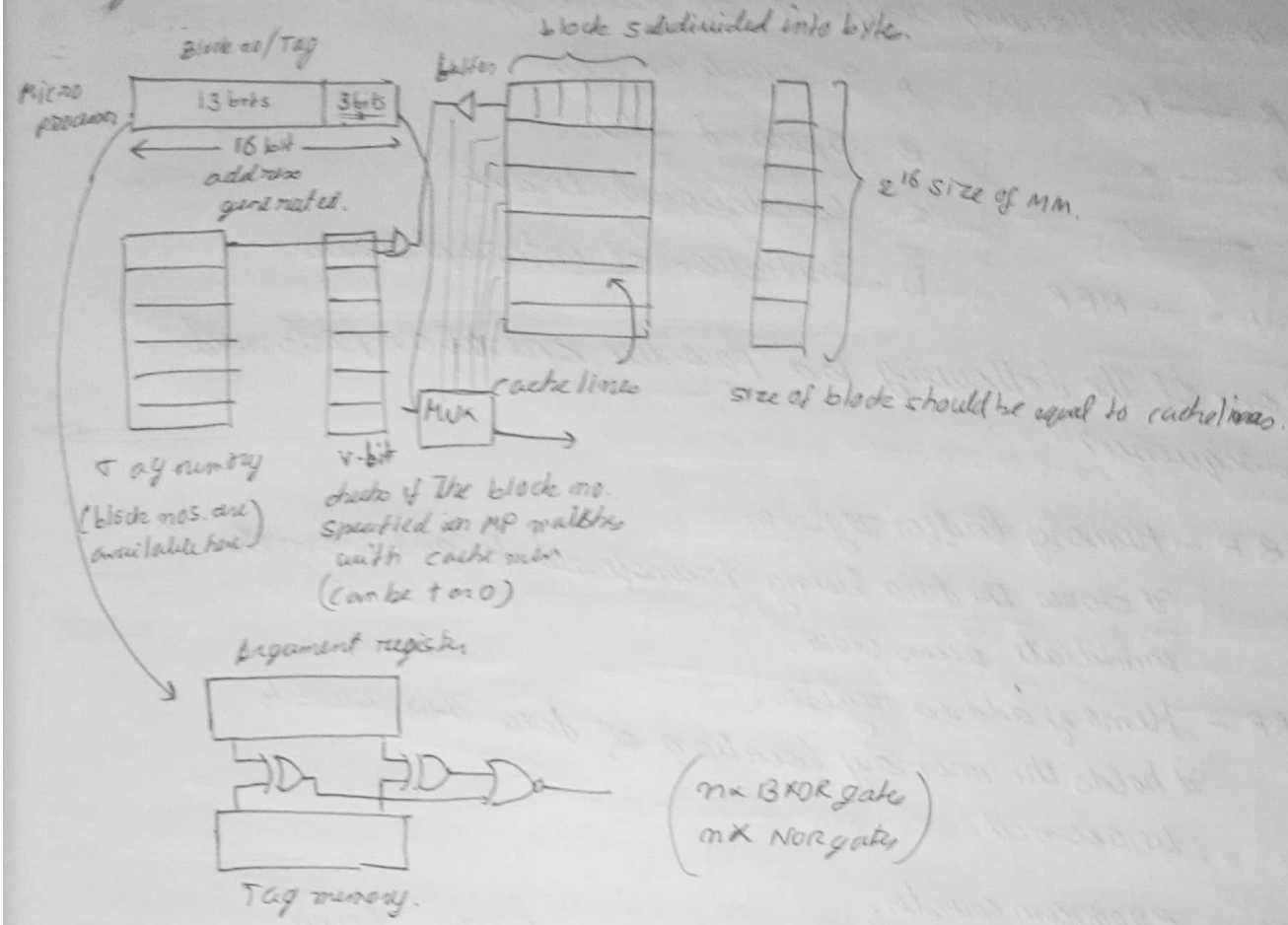
Built-in.

★★ Cache memory

- Associative
- Direct mapped
- Set Associative

Scanned by CamScanner

Pages can be subdivided into blocks.



22/8/19

Q. Consider a CPU where all instructions require 7 clock cycles to complete execution. There are 140 instructions in the instruction set. It is found that 125 control signals are needed to be generated by the control unit. While designing the horizontal microprogrammed control unit, single address field format is used for branch control logic. What is the maximum size of the control word & control address register?

Sol^n - Each instruction takes 7 cycles

∴ 140 instructions take = (140×7) cycles

= 980 cycles

Now for $2^m \geq 980$

$m \geq 10$

∴ (10 + 125) bits ⟶ control word

10 bits control address register.

Scanned by CamScanner

Q. Consider the following sequence of micro operations.

MBR ← PC	A. Instruction fetch
MAR ← X	B. Operand fetch
PC ← Y	C. Conditional branch
Memory ← MBR	D. Initiation of interrupt service.

Which one of the following is a possible operation performed by this sequence?

Sol: MBR – Memory buffer register
It stores the data being transferred to and from the immediate auxo store.

MAR – Memory address register.
It holds the memory location of data that needs to be accessed.

PC – Program counter.
It contains the address of the instruction being executed at the current time.

The 1st instruction places the value of PC into MBR.
The 2nd instruction places an address X into MAR
The 3rd instruction places an address Y into PC.
The 4th instruction places the value of MBR (old PC value) into memory.

D.

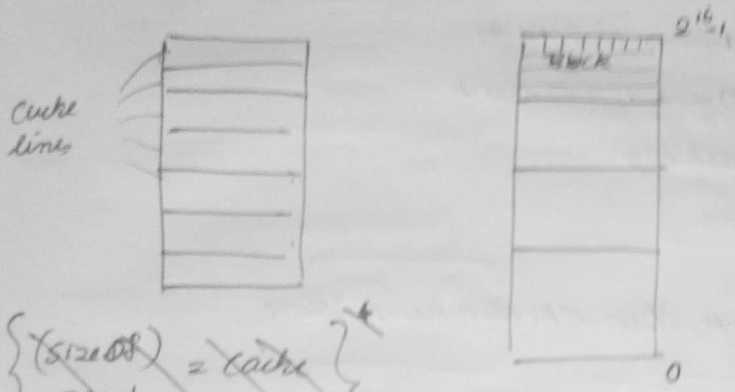
Scanned by CamScanner

23/8/19

* Cache Memories — Association

SRAM (static ram)
makes up cache memory.

CM < MM < SM



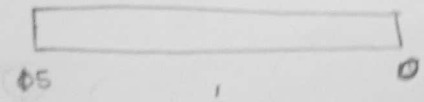
{(size of) Block in each page = cache memory} *

user programs cannot be placed in cache memory.

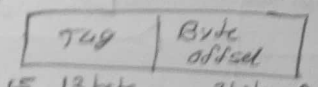
CPU —first search→ CM —then search→ MM

cache miss / cache hit
(does not get) (gets it)

microprocessor = address

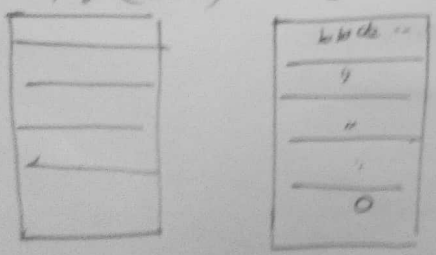


⇒ 05 divide the 16 bits to reach through CM



this part is called Tag

Tag memory stores address of blocks in CM

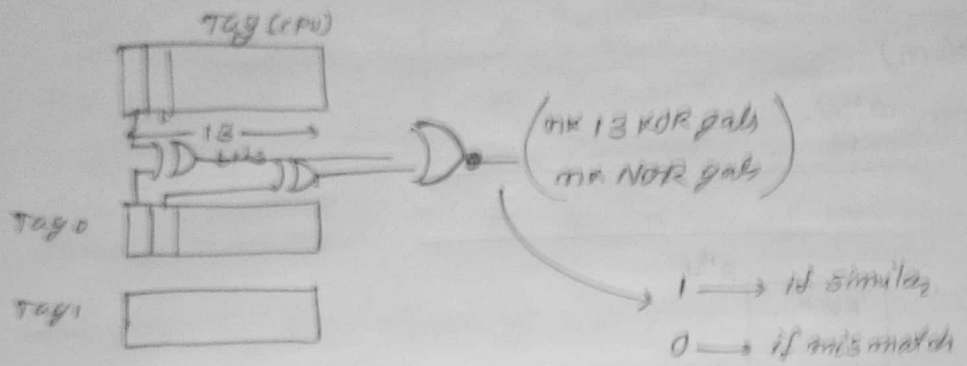


Any blk of the MM can be placed in any cache line

We search the Tag sequentially for a block till we get it —if we don't→ Cache miss.

Scanned by CamScanner

Remove the Tag & place it in Argamed register.

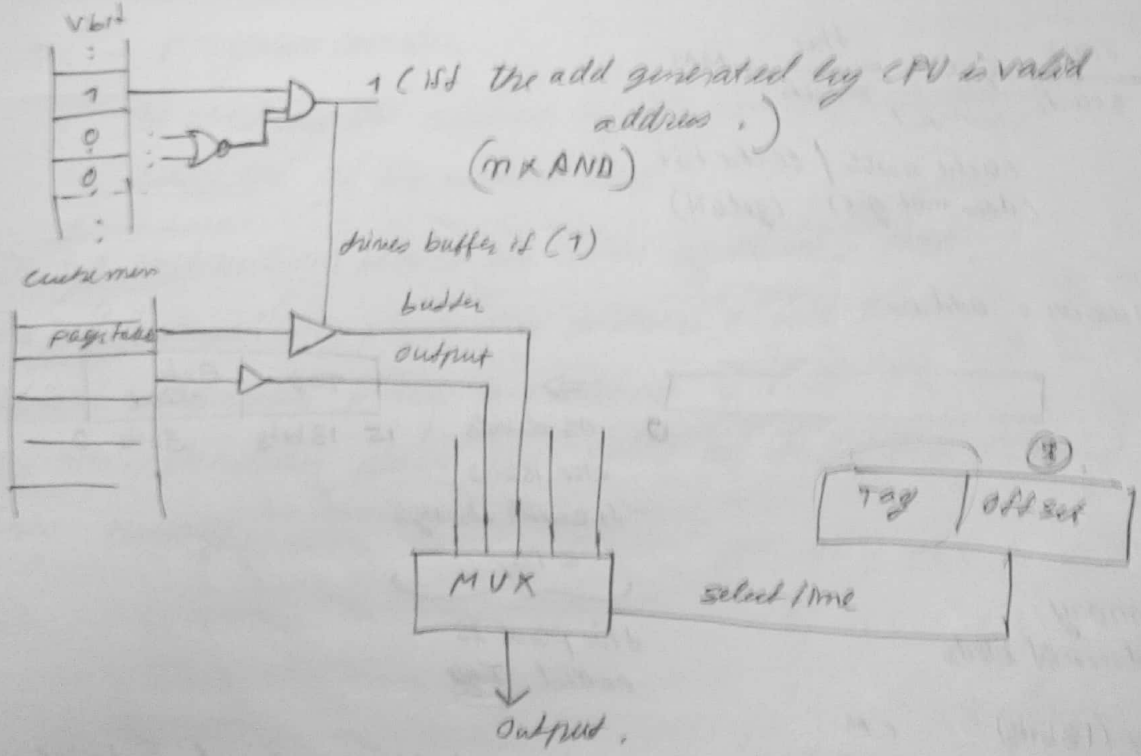


V-bit

- One bit associated with each tag
- Initially all bits are 0.

* As we have not cleared CM, then CM can have some garbage value.

When a block is sent from MM → CM then the VBit is set to 1 (ISS the add generated by CPU is valid address.)

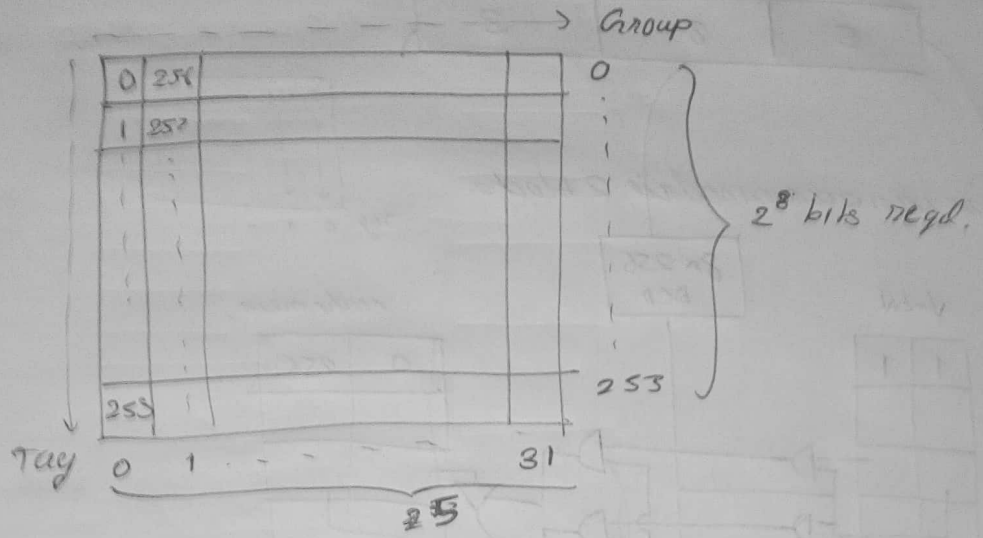


Scanned by CamScanner

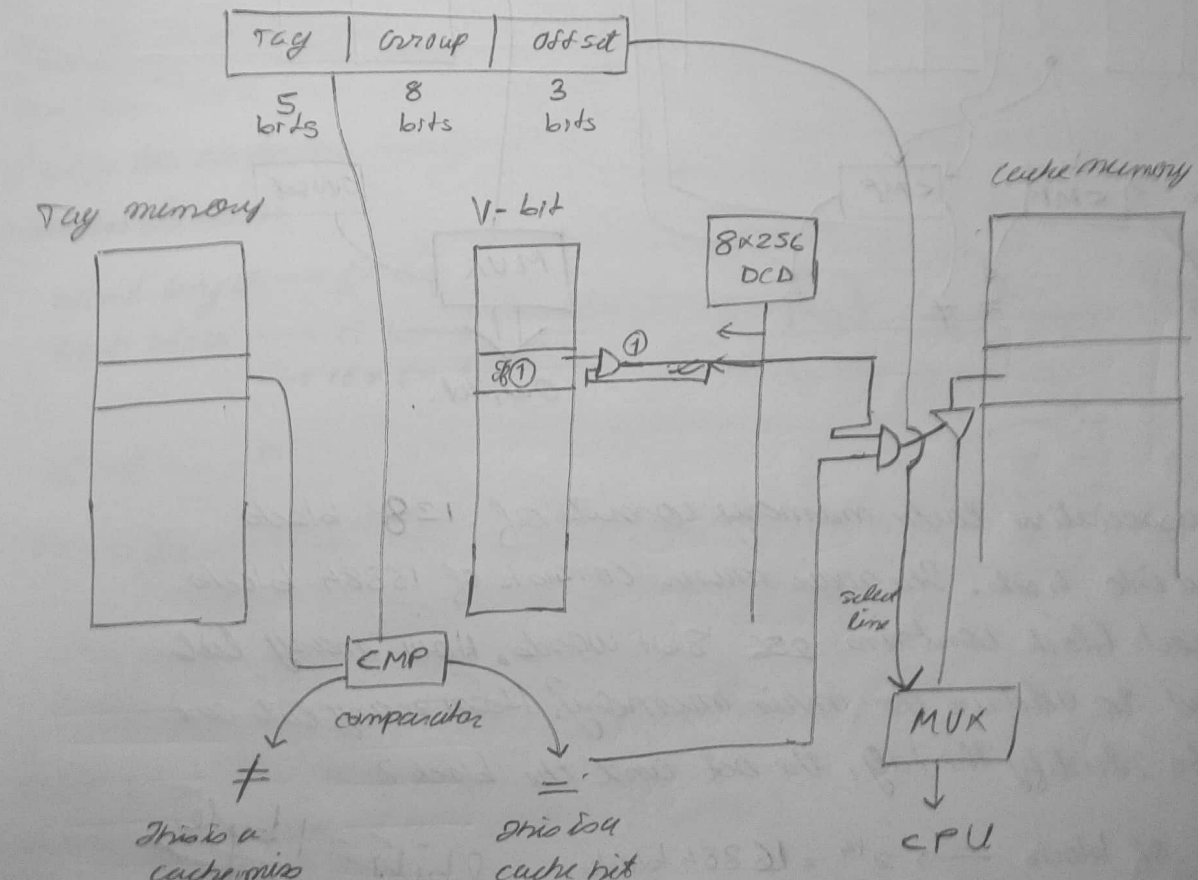
* Direct Mapped Cache

Rest. which block will be allowed to move to which cache line

Now, cache is not 1D structure



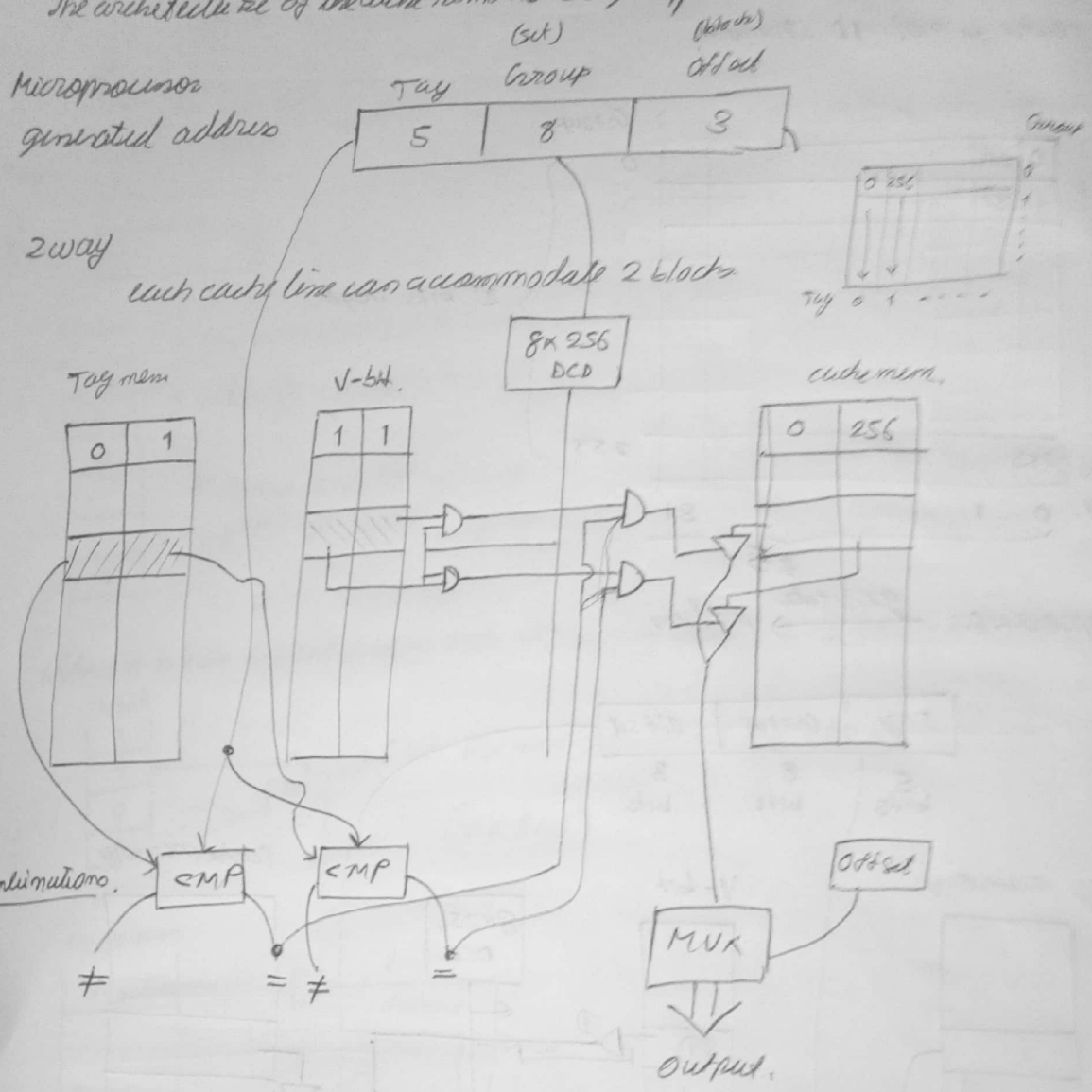
Microprocessor generate → address



Scanned by CamScanner

Set Associative Cache

The architecture of the cache remains 2D group.



Q. A set associative cache memory consists of 128 blocks divided into 4 sets. The main mem. consists of 16384 blocks and each block contains 256 8bit words. How many bits are reqd. to address the main memory? How many bits are reqd. to identify the tag, the set and the block?

Soln- No. of blocks $\longrightarrow 2^{14} = 16384$ blocks.

each block $\longrightarrow 256 \times 8$ bits.

$16384 \times 256 = 2^{22}$

8 bits = 1 byte

22 bits MM

i) 22 bit reqd. to address main memory.

Scanned by CamScanner

⇒ To identify block → 256
= 28

8 bits for block identification.

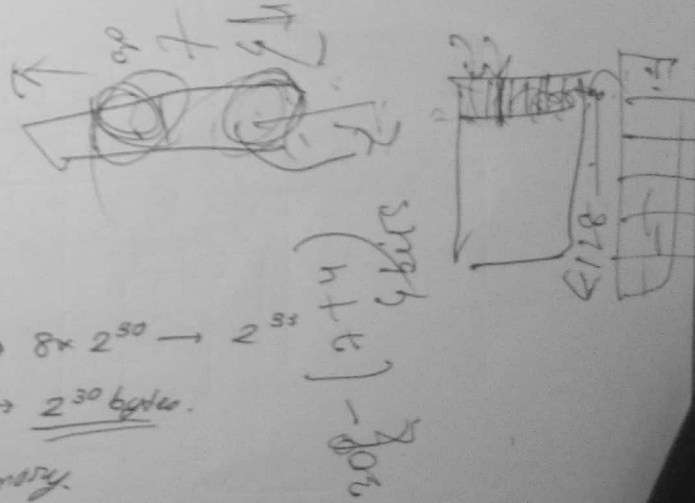
No. of Block in cache = 128

128/4 → 4 acts
= 32
= 2^5

5 bits reqd to set identification.

Bits reqd for tag identification → 32 − (5+8)

Q. Computer has an 8gb memory with 64 bit word size. Each block of memory store 16 words. Computer has a direct mapped cache of 128 blocks and use word level addressing. What is the addressing format if we change the cache to a 4 way set associative cache, what is the new address format?



Sol: word length → 64 bit.
each block → 16 words.
→ 16 × 64 bits.

$2^4 \times 2^6 \rightarrow 2^{10}$

For 1gb → 2^{30} bits. 8gb → $8 \times 2^{30} \rightarrow 2^{33}$
→ 2^{30} bytes.

30 bits reqd to address main memory.

Each block → 16 words
4 bits for offset identification. [offset].

1 (4 way set ass)

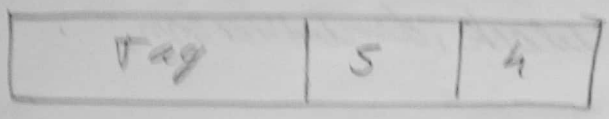
block identification, (128 blocks) → 2^7
(7 bits reqd)

tag ~~group~~ → 30 − (7+4)

Scanned by CamScanner

In 4 way set ass → $\frac{128}{4}$ → divided into 4 sets cache.

→ 32 → 2^5.



Scanned by CamScanner