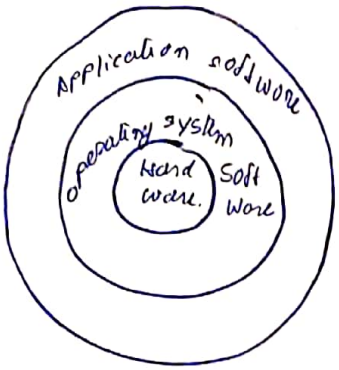
24/7/19



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 → Decoder → Timing & control unit → control signals.

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

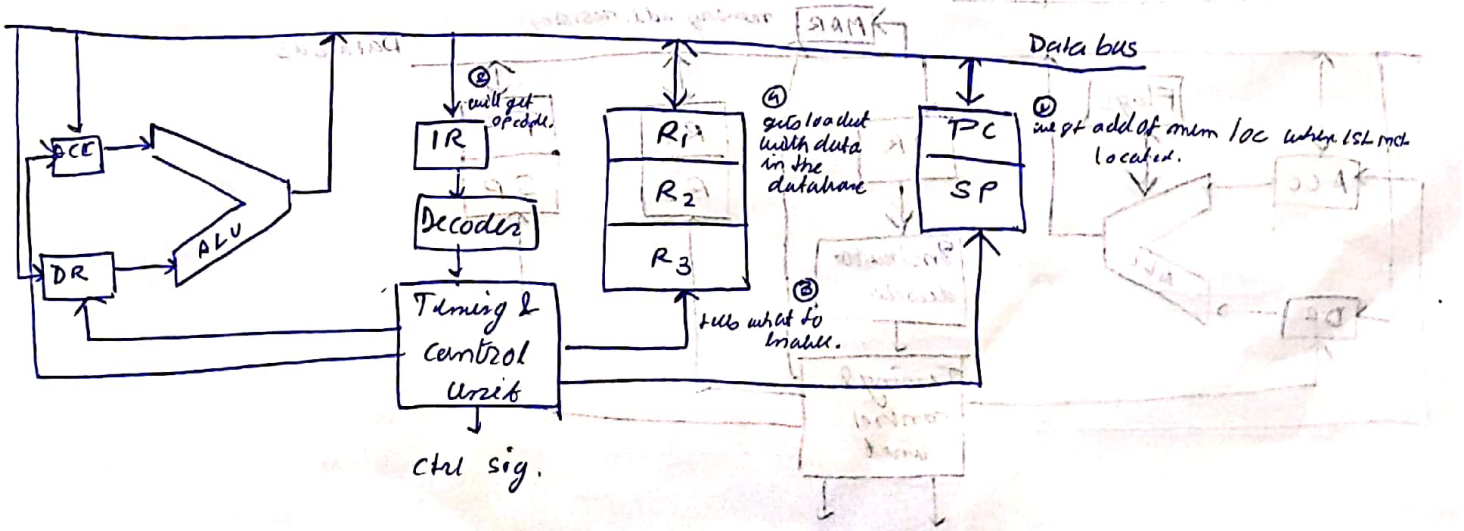
Decoder → in machine language

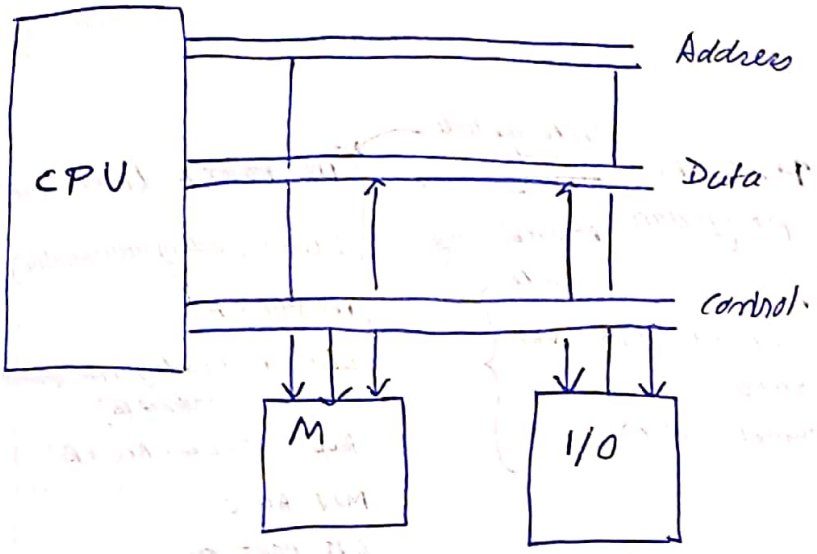
● MOV R_1, R_2

control signal read: OE R_2, LD R_1

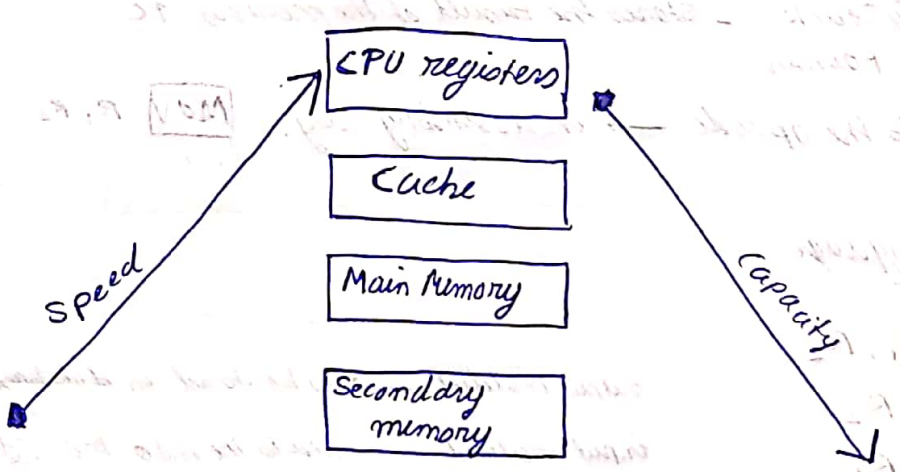
output enabled → to be stored in database
input enabled → reads the info from database

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





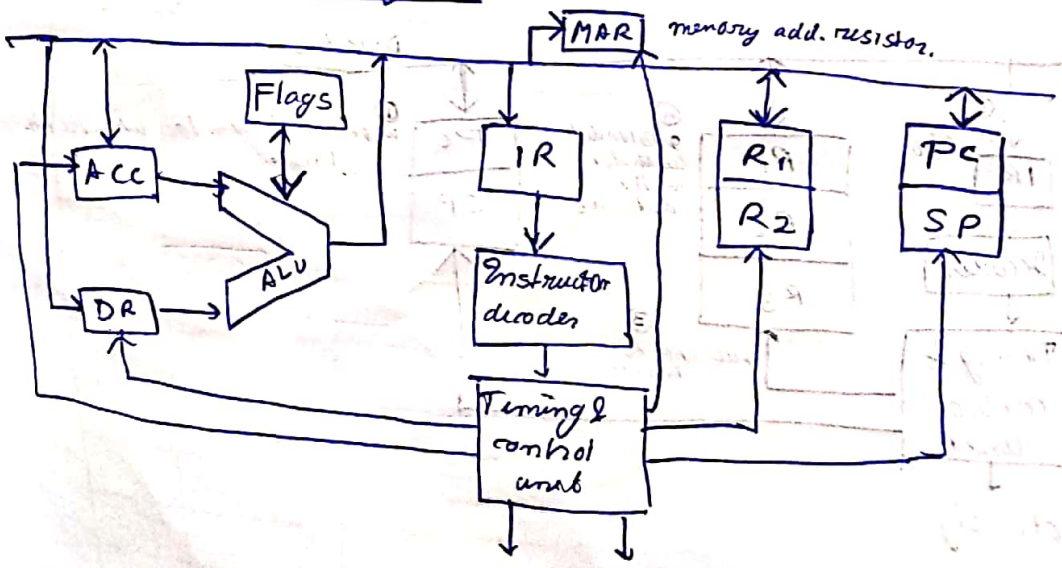
* Memory hierarchy



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

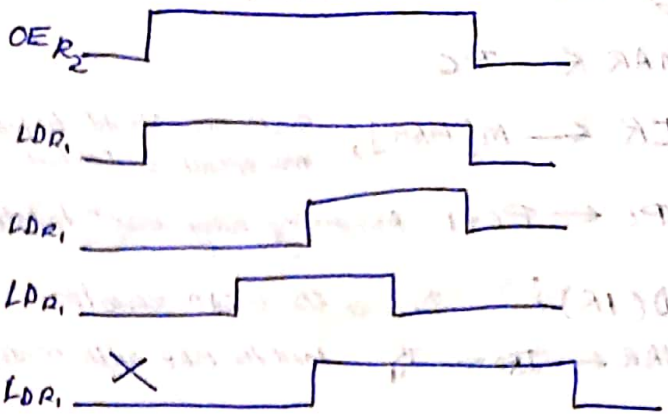
25/7/19

* Simple CPU Design



MOV R_1, R_2 ← contents of R_2 moved to R_1

generated { OE_{R_2}, LD_{R_1}



NP. because at the later point of time some other Register may be output enabled and R_1 may receive unnecessary info.

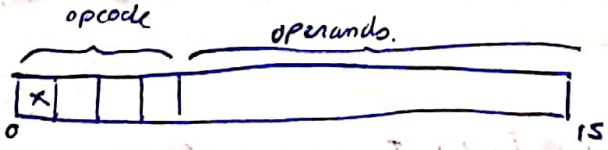
the timing & ctrl unit makes care of this

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 [Acc ← $\overline{Acc}$]
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



Instruction register.

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

* For any instⁿ to be executed

T_0: MAR ← PC

T_1: { IR ← M[MAR]; Instⁿ reg. should be loaded with the instⁿ in MAR now, opcode is selected.
PC ← PC+1 Assuming every instⁿ to take only 1 memory location

T_2: { DCD(IR); Decode the instⁿ register
MAR ← $IR_{0:11}$ Load the MAR with content from $IR_{0:11}$

T_3: ADD R_1
DR ← R_1 DR is loaded with contents of R_1 so that ALU can access it.

T_4: ACC ← ALU_{ADD} (ACC, BR), T0 (result is given down to ALU)
(Time stamps)

move back to T_0 (time stamp)
next instⁿ fetched.

MOV R_1, R_2

T_3: R_1 ← R_2
⇓
~~mov~~ next T_0.

MOV ACC, M

T_3: ACC ← M[MAR] Acc gets the contents of the memory address which is already loaded to $IR_{0:11}$ ~~and~~ (meaning loaded to MAR)
T_0;

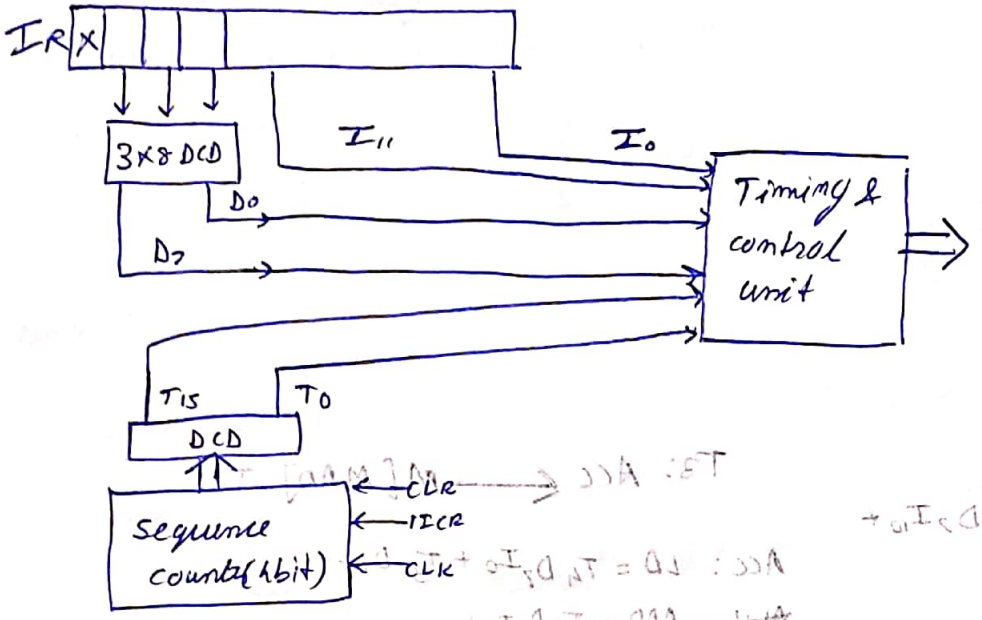
#

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

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

T_{15} T_0
DCD 4 bit decoder
Sequence counter
← CLR every time we go back to T_0, we clear the sequence counter
← INCR
← CLK clock.

- $I_0 : I_{15}$

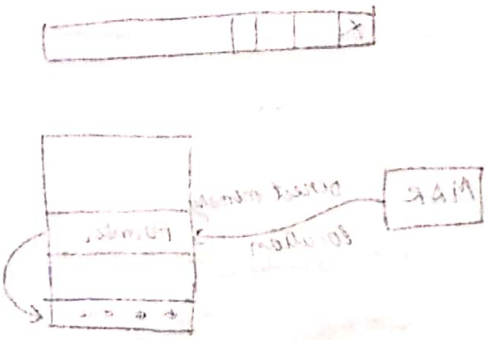


* Usually in 8085 machine the 0 address has a Jump instⁿ
  Jumps to application area

Timestamp T_0

PC: $OE = T_0 +$ (enabled)

MAR: $LD = T_0 +$ (enabled)



Timestamp T_1

PC: $INCR = T_1 +$

IR: $LD = T_1 +$

Timestamp T_2

MAR: $LD = T_2 +$

IR: $OE = T_2 +$

For ADD R_1 : Timestamp T_3 T_4,

DR: $LD = T_3 D_7 I_0 +$

R_1: $OE = T_3 D_7 I_0 +$

$ACC = T_4 D_7 I_0 +$

$ALU = ADD = T_4 D_7 I_0 +$ [register ref. instⁿ (OCD) → 111]

corresponds to D_7

SEQCNTR: $INCR = T_0 + T_1 + T_2 + T_3 D_7 I_0 +$ ~~$T_4 D_7 I_0 +$~~

$CLR = T_4 D_7 I_0 +$

For MOV ~~Acc,M~~ R_2, R_1, Timestamp T_3

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

R_1: $OE = T_3 D_7 I_5 +$

For MOV R_1, R_2,

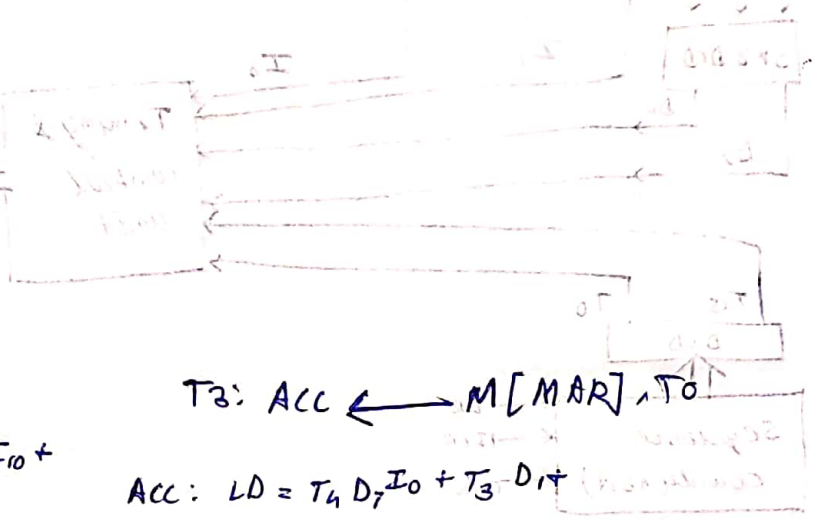
R_1: $LD = T_3 D_7 I_{14}+$

R_2: $OE = T_3 D_7 I_4+$

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 D7I~~ $LD = T_3 D_7 I_{10}+$

~~MAR :~~,

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

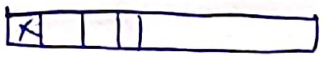
Acc: $LD = T_4 D_7 I_0 + T_3 D_1+$

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

SEQCTR: INCR =

CLR = $T_4 D_7 I_0 + T_3 D_1+$

#



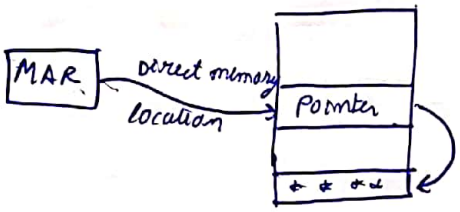
MOV Acc, M

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: Ito not scaled.

* Microprogrammed ctrl unit design. → alternate
  adv: scalable as programming is allowed.
  disadv: ↓ programming allowed. memory is used up and hence slow.

* control memory (CM) → control words.

In time stamp T_0,

LD_{MAR} — C_0 designated 0th bit of CM

OE_{PC} — C_1 " 1st bit of CM

0 . . . . . 11 01

Branch add.

(BA)

During T_1,

LD_{IR} — C_2

RD_M — C_3

(read memory)

$INCR_{PC}$ — C_4

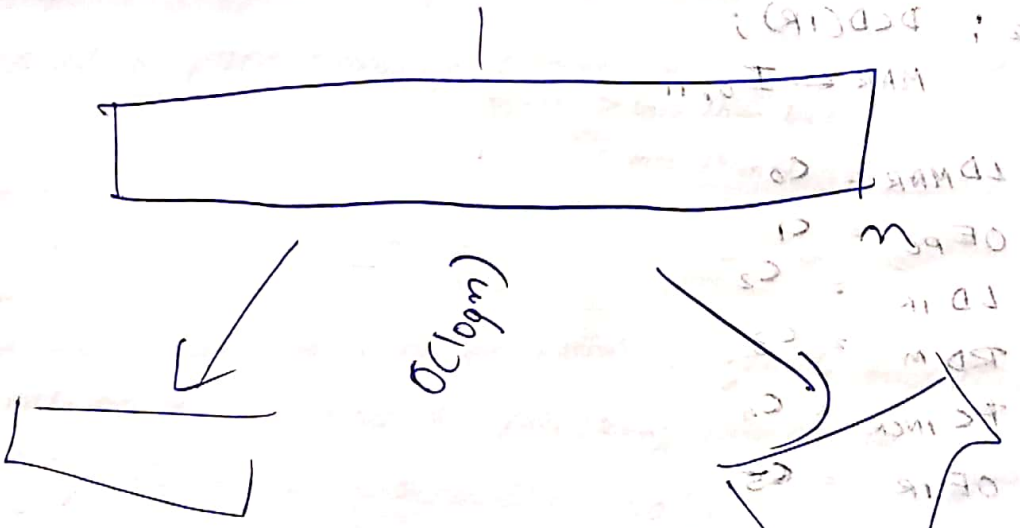
0 . . . . . 0111 00 02

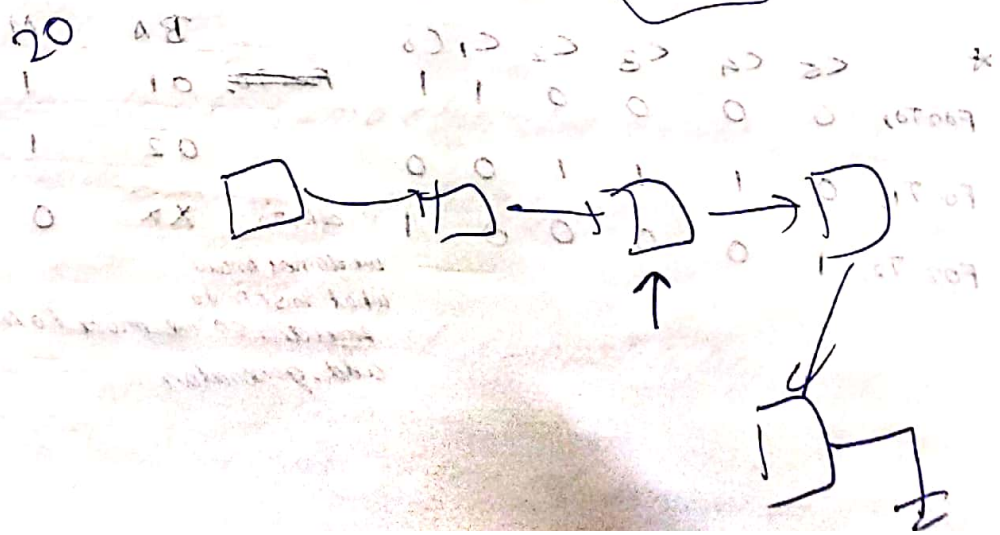
During T_3

OE_{IR} — C_5

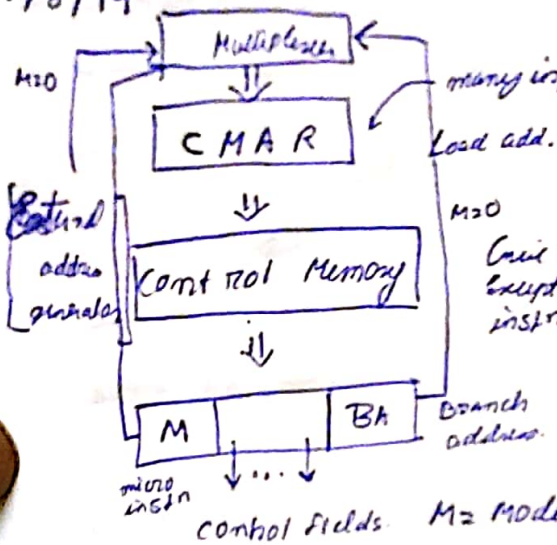
0 . . . . . 0100001 xx

sum(arr[ ])





7/8/19



M = mode. (acts as select line)

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

Micropogram 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 ← $I_{0:11}$

$LD_{MAR} = C_0$

$OE_{PC} = C_1$

$LD_{IR} = C_2$

$RD_M = C_3$

$PC_{INCR} = C_4$

$OE_{IR} = C_5$

	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.	XX	0

we do not know what instn to execute, so we move to unknown add. generator.

Eg: MOV R_1, R_2

C_7 C_6 C_5 ..... 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} C_9 C_8 C_7 C_6 ..... BA Mode.

0 .... 0 1 1 0 0 ..... 0 R=1 1

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

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

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

$LD_{ACC} = C_{10}$

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 — (some where in between) — requires > bits than VμP, < 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 covers.

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).

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 — ctrl signals reqd.

S_1 : a, b, c, d, e, f, g, h. — next, after S_2, strike off all signals not appearing together

S_2 : bd, be, bh, cd, de, dg, dh, ef, eg, fg, fh, gh

pairs of signals which are never together

S_3: make a group of 3 ctrl signals

bde, bdh, edg, ~~bfg~~, ~~cde~~, ~~cdg~~, ~~cdh~~, ~~def~~, ~~bgh~~ dgh

deg fgh

~~edg~~

S_4 : 4 ctrl signals together

ϕ

8/8/19

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							×	×
K_7 = efg							×	×
K_8 = fgh							×	×

dominated

dominating

dominating.

essential covers.

dominated is a subset of the dominating.

Strike off dominating columns.

So, K_1, K_2 are essential covers.

Reduced covertable

	b	c	e	f	g	h
$K_3 = bde$	×		×			×
$K_4 = bdh$	×					
$K_5 = deg$			×			×
$K_6 = dgh$						
$K_7 = efg$			×	×		
$K_8 = fgh$				×		×

we will strike off dominated rows and not dominating rows.

Minimal cover = $\{(K_1, K_2), K_3, K_8\}$ or $\{(K_1, K_2), K_4, K_7\}$

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

⇓

"all the control signals have been covered"

* 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 inst^{n}s non pipeline method.

- Inst^n fetch — IF
- opcode fetch — OF
- opcode 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				

(IR gets instruction I_1 is fetched)

7 time stamps reqd.

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 ②
I_1 : ADD R_3, R_4 ③ ④
I_2 : SUB R_1, R_5 ① ⑤
I_3 : ADD R_2 ②
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

* Assuming I_0 & I_8 dependent on each other.

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

9/8/19

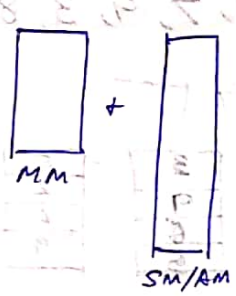
MEMORY MANAGEMENT

* Virtual Memory

Made to believe that we have available

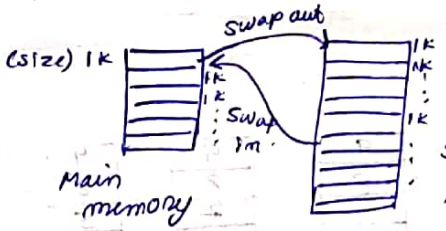
- Paging
- Segmentation

} 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.

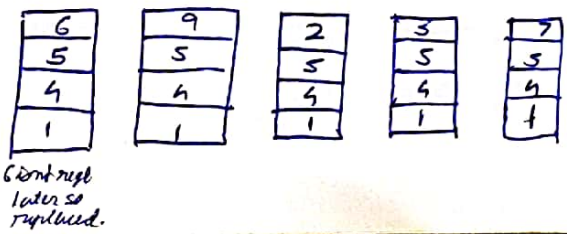
- FIFO (First in first out)
- LRU (Least recently used)
- MFU (Most frequent used)
- Optimal replacement

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

page reference string

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

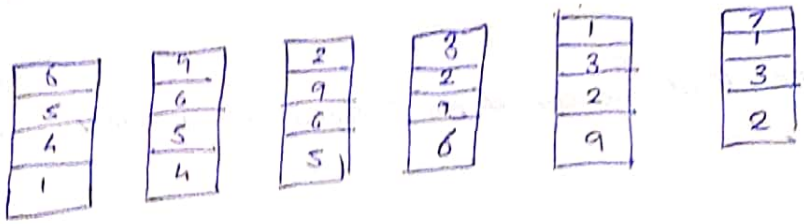
main memory $\xleftarrow{\text{accommodate}}$ 4 pages.



Cannot replace later so replaced.

FIFO:=

(like queue)

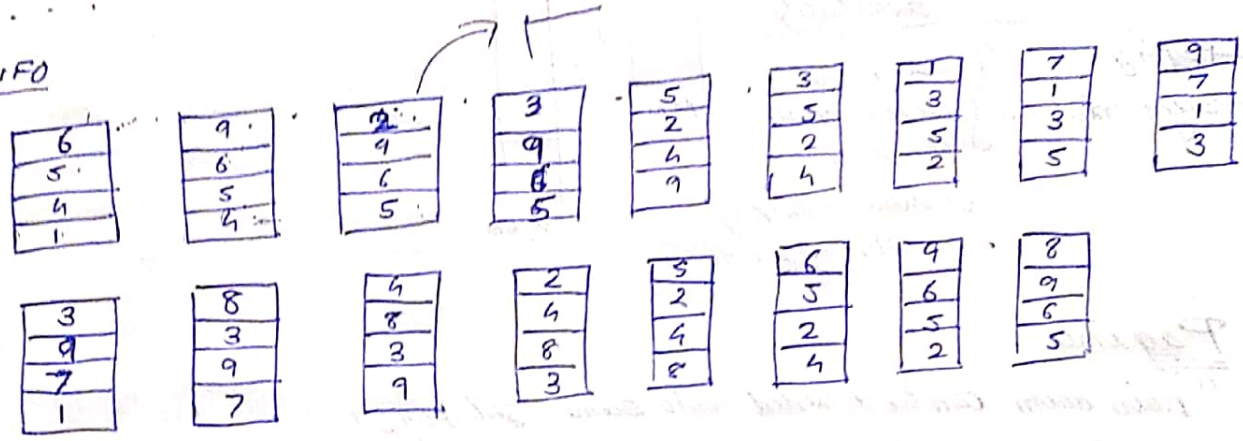


~~FIFO~~:-

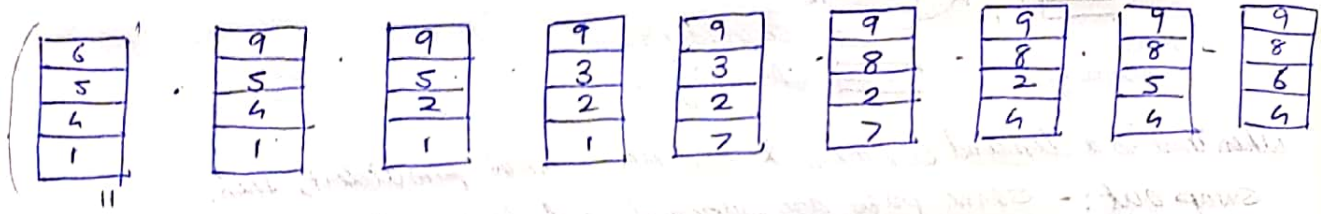
* No. of page misses.

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

sol:- FIFO



opt. rep.



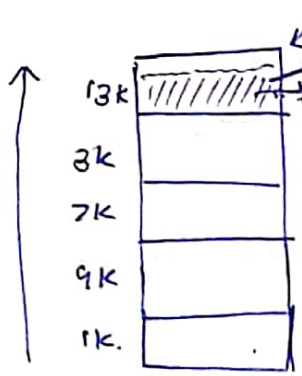
+8 +4 ⟶ 12

~~RF~~ FIFO

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

13+4 ⟶ 17

Segmentation



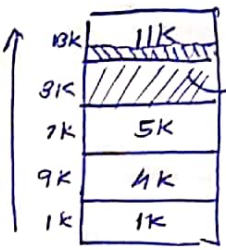
2K from secondary storage space

rest of the 11K cannot be used to store any thing

→ Internal fragmentation

- Best fit
- First fit
- Worst fit.

External fragmentation



→ Best fit

aim :- go for minimum fragmentation.

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

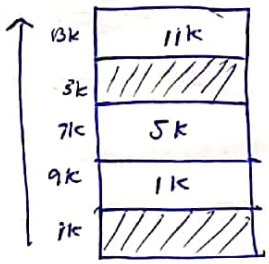
→ First fit

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

→ Worst fit

aim :- maximize fragmentation

[11K, 1K, 5K, 10K, 4K



★

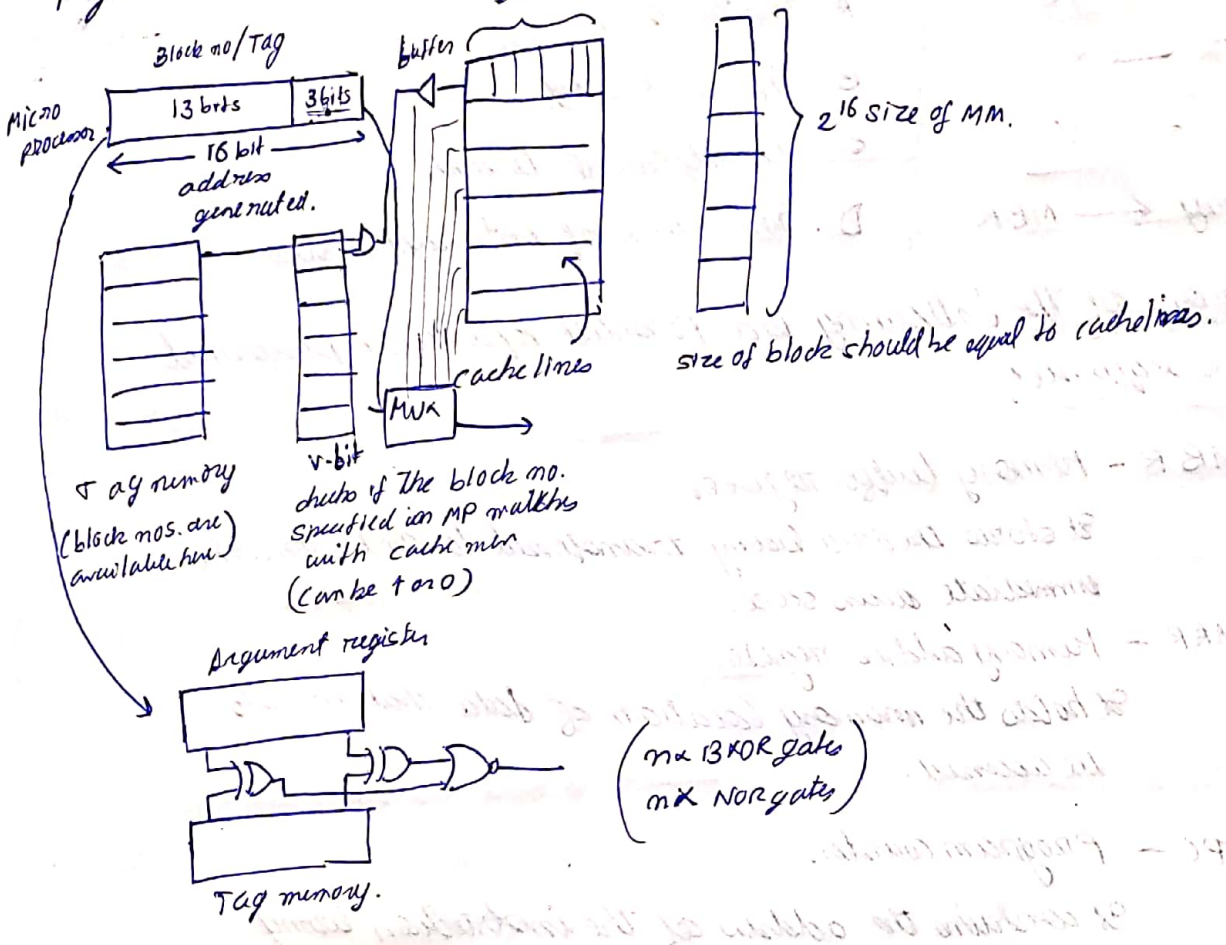
DRAM (Dynamic RAM) [made of capacitors] → Main memory
Builds.

SRAM (Static RAM) [costly] → Cache memory

★★ Cache memory

- Associative
- Direct mapped
- Set Associative

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:- Each instruction takes 7 cycles

$\therefore$ 140 instructions take = (140×7) cycles

= 980 cycles

Now for $2^n \geq 980$

$n \geq 10$

$\therefore$ (10 + 125) bits → control word

10 bits control address register.

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 access 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.

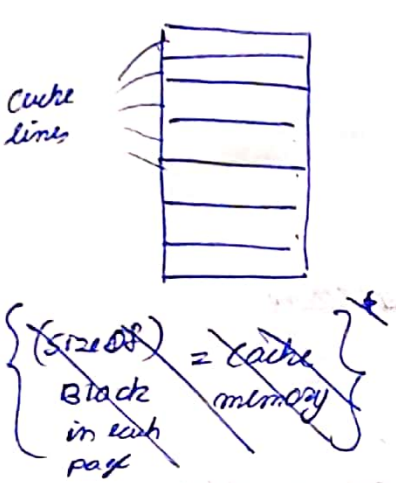
D.

23/8/19

* Cache Memories — Associative

SRAM (static ram) makes up cache memory.

CM < MM < SM



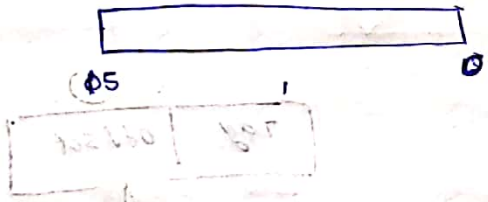
~~(size of Block in each page) = (cache memory)~~

user programs cannot be placed in cache memory.

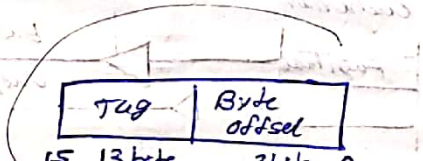
CPU —first search→ CM —then search→ MM

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

microprocessor = address

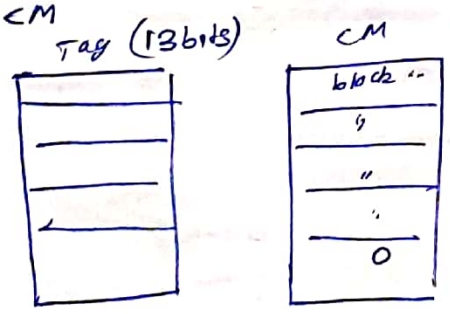


⇒ OS divides the 16 bits to search through CM



this part is called Tag

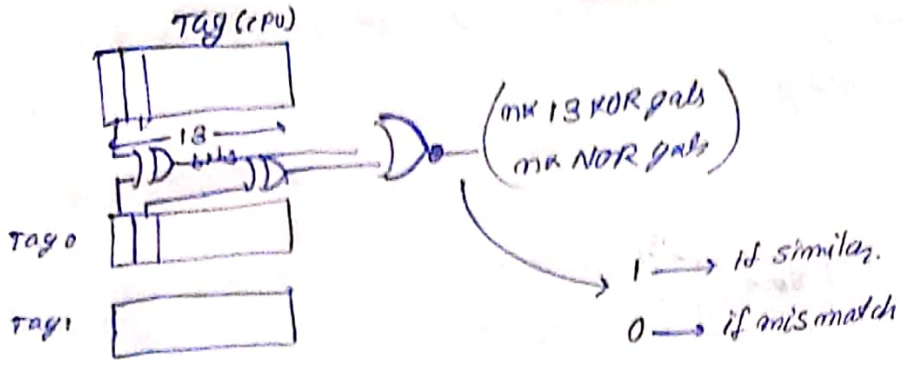
Tag memory stores address of blocks in CM

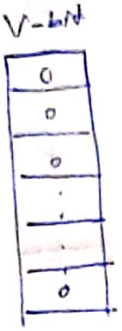


Any blk of the MM can be placed in any cache line

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

Remove the Tag & place it in Argument register.

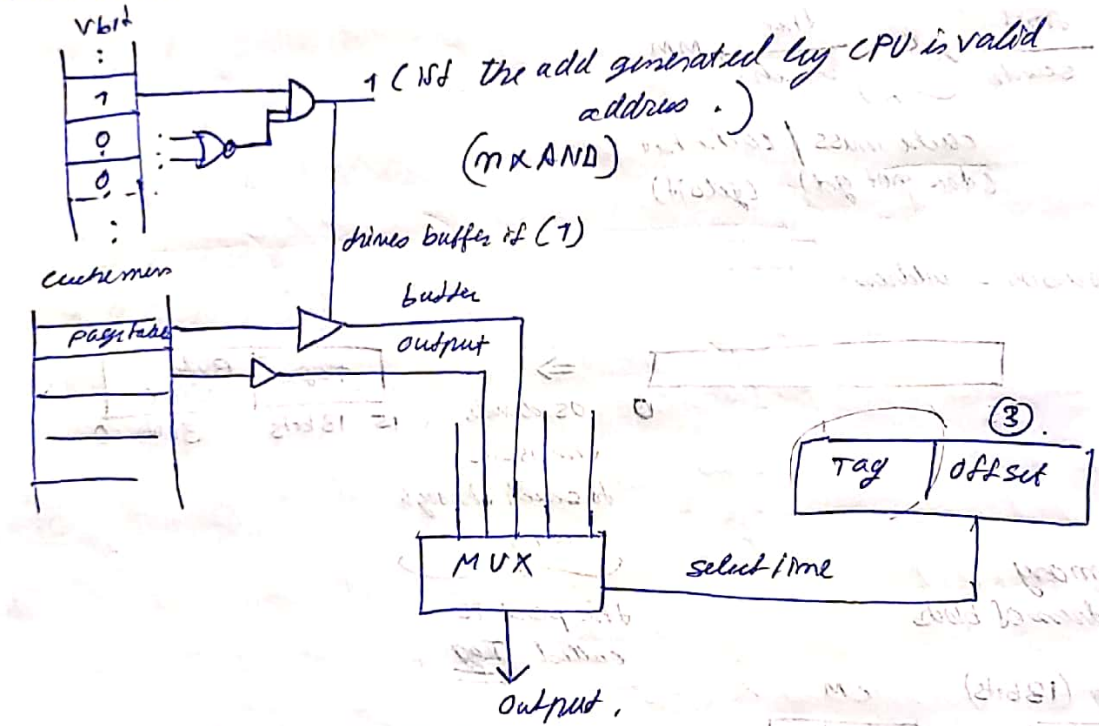




- one bit associated with each tag
- Initially all bits are 0.

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

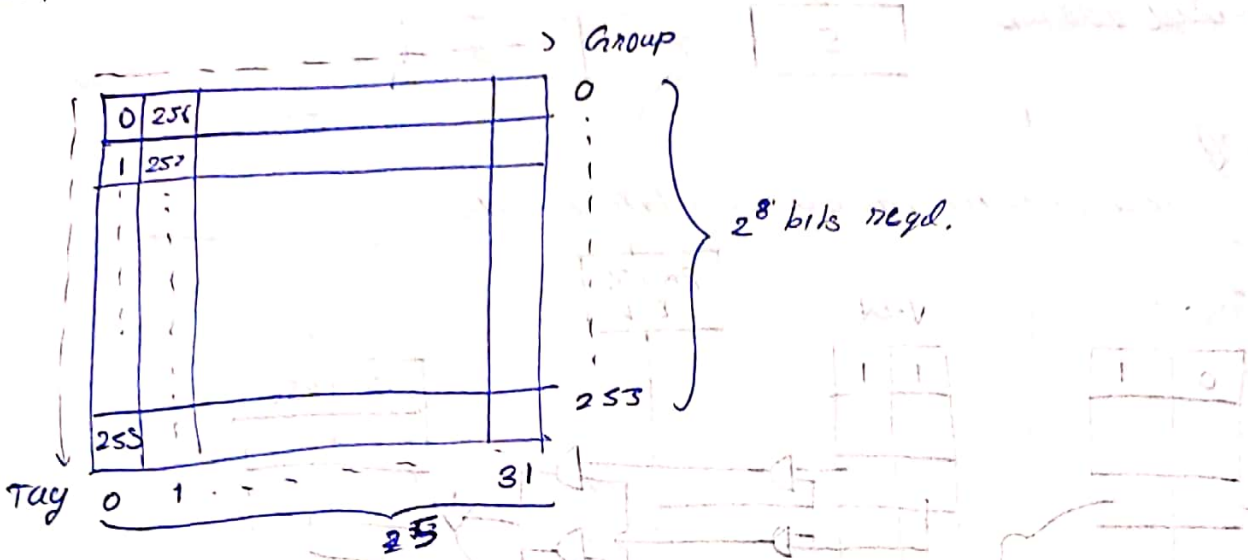
When a block is sent from MM → CM then its V Bit is set to 1.



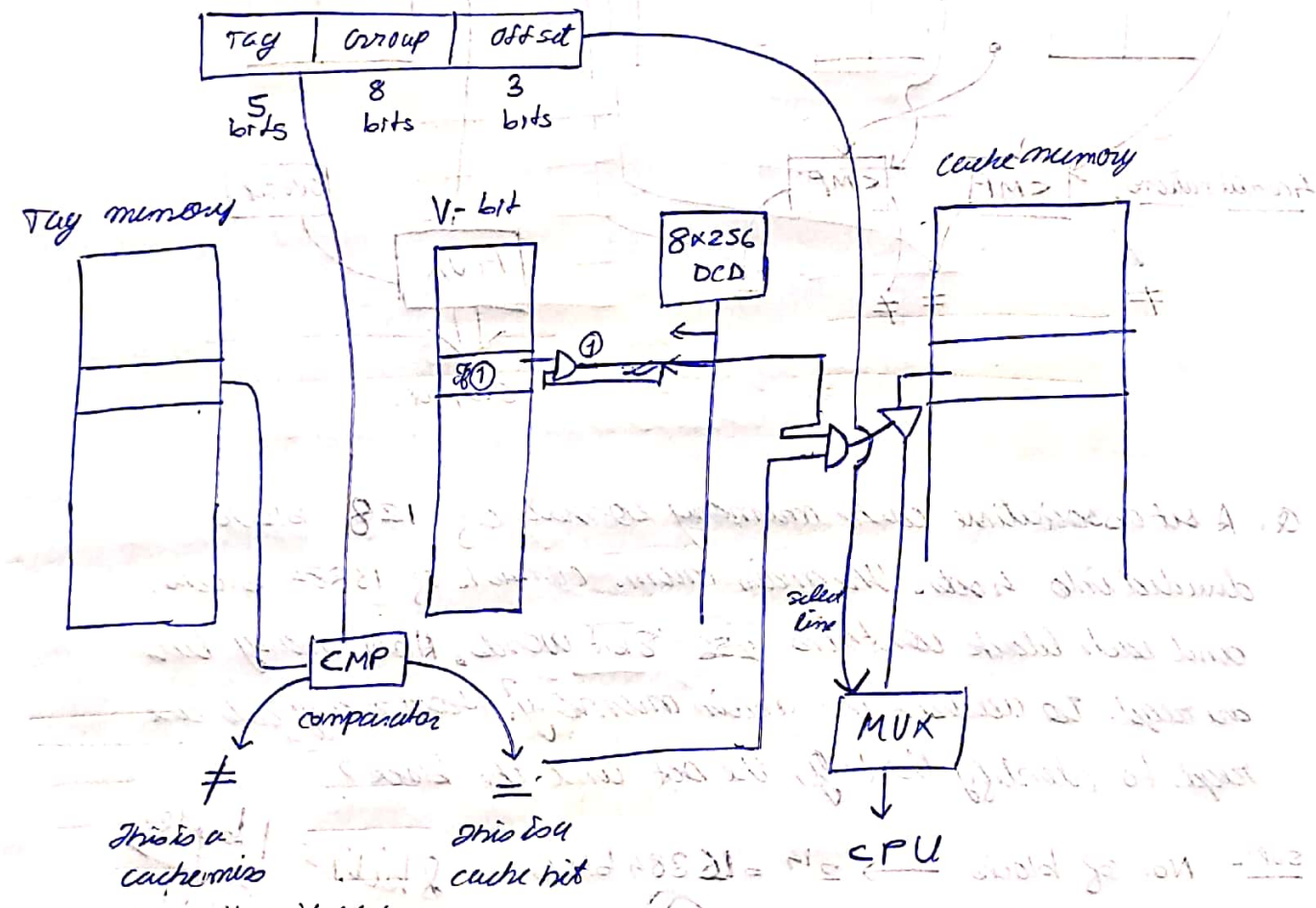
* Direct Mapped Cache

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

Now, cache is not 1D structure

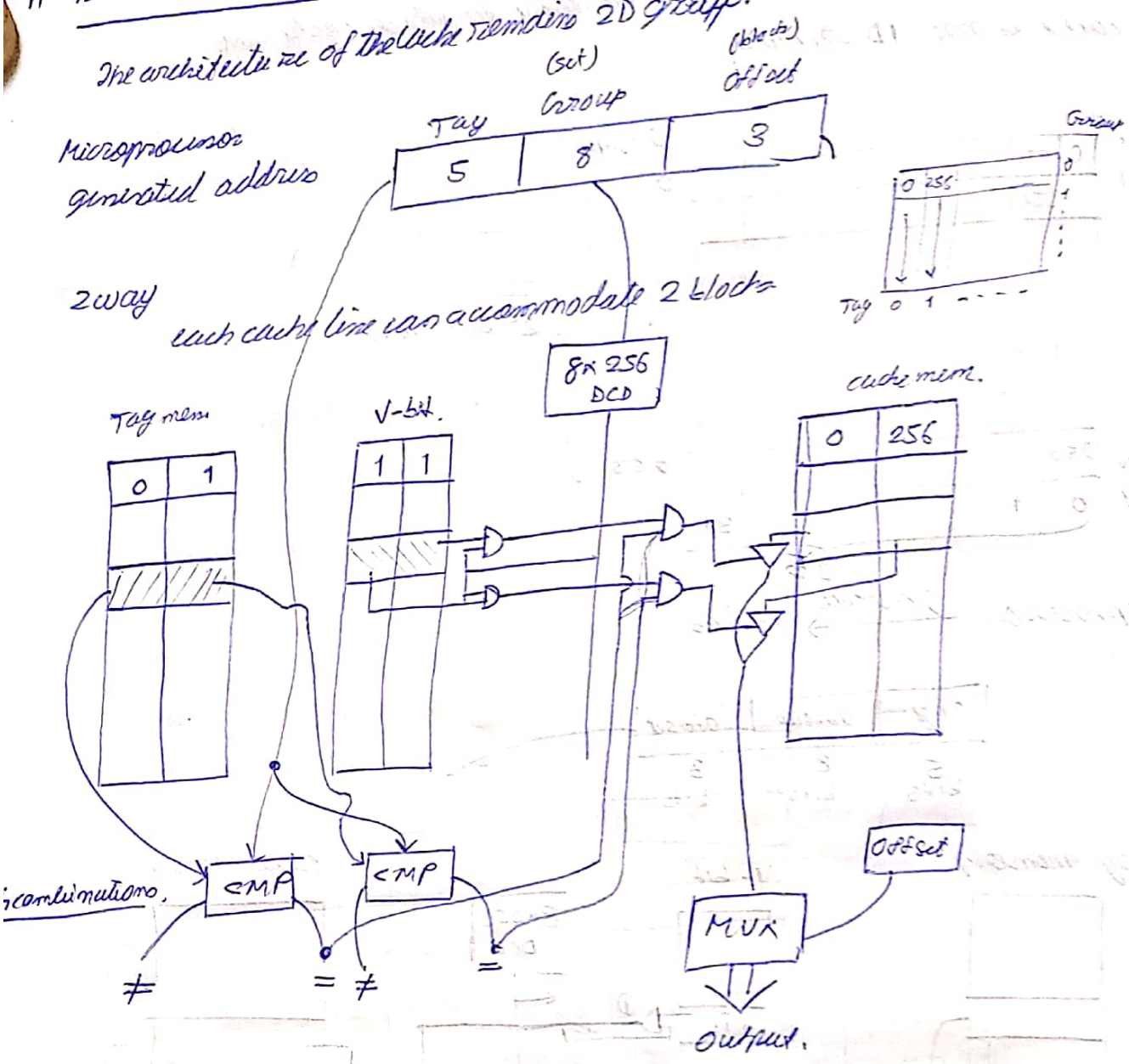


Microprocessor generates address



Set Associative cache

The architecture of the cache is in 2D grid.



Q. A set associative cache memory consists of 128 block 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?

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

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

8 bits = 1 byte

22

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

$\Rightarrow$ 22 bit reqd. to address main memory

MM

⇒ To identify block → 256
= 28
8 bits for block identification.

No. of Block in cache = 128

128/4 → 4 sets
= 32
= 2^5
5 bits reqd. to set identification.

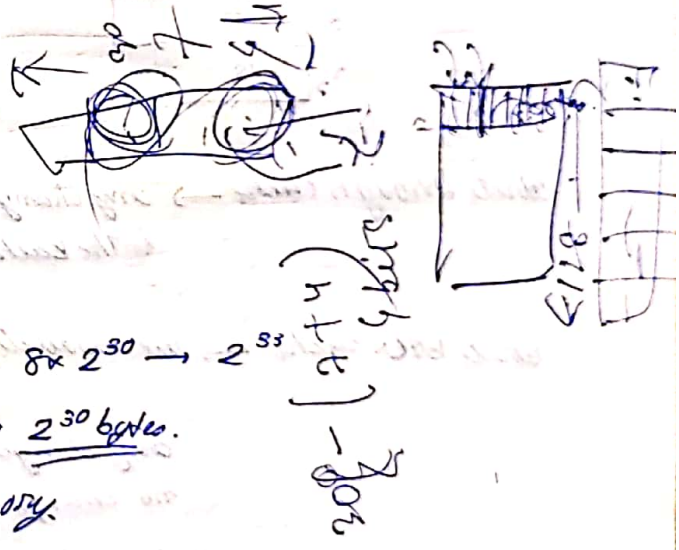
Bits reqd. for tag identification → 22 − (5+8)

Q. Computer has an 8gb memory with 64 bit word size. Each block of memory stores 16 words. Computer has a direct mapped cache of 128 blocks and uses 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 1 gb → 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 identification. [offset].
offset.

(4 way set ass)

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

tag ~~group~~ → 30 − (7+4)

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

→ 32 → 2^5.

Tag	5	4

4/9/19

* MULTI LEVEL CACHES

L_1 — on chip cache [hence fastest]

L_2, L_3 } unified / split → it is in motherboard [slower than L_1]

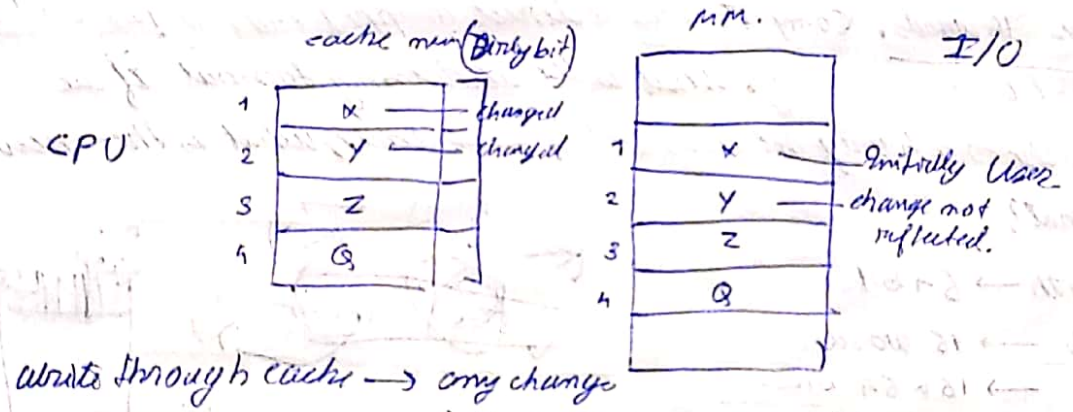
↓

requires data path to transit

but distance travelled is less & hence faster than MM.

L_1 ← instruction related info.

L_2 ← all other data goes to L_2.



write through cache → any change in the cache is written through MM

write back cache → we associate dirty bit with each cache line

dirty bit

any change → 1

no change → 0

check if a block in MM is in CM. & then check if dirty bit is set = 1, then write back of the cache is written.

Q1. A set associative cache consists of 64 lines divided into 2 line sets. The main memory contains 4K blocks of 128 words each. Show the format of main memory address.

Q2. Consider computer with the following characteristics. MM = 1mb, word size = 1 byte. Block size = 16 bytes. cache size = 64 kb. For the main memory, addresses of F0010, 01234, give the corresponding tag, cache line address, and the word offsets for a direct mapped cache.

Soln 1. Main memory ⟶ 4K 1024 blocks. (128 words)

$= 2^2 \times 2^{10} \times 2^7$

$= 2^{19}$

19 bits reqd to address main mem.

Set ass. cache ⟶ 64/2 = 32 = 2^5

5 bits.

blocks ⟶ 4×1024 = $2^2 \times 2^{10} \times 2^{12}$

Block offset ⟶ 128 = 2^7.

7 bits.

Tag = 19 − (7+5) = 7 bits

Tag	Set ass.	block offset.
7	5	7

Soln 2. Main memory ⟶ 1mB = 2^{20}

word size ⟶ 1 byte.

block size ⟶ 16 bytes ⟶ 2^4

cache size ⟶ 64 kb

no. of cache lines = $\frac{64\text{ kb}}{16\text{ bytes}} = 4 \times 2^{10} = 2^{12}$.

Tag = 20 − (12+4) = 4

Tag	Set ass.	Block #
4	12	4

F0010

1111 0000 0000 0001 0000

Tag. Cache line add. Word offset.

01234 0000 0001 0010 0011 0100