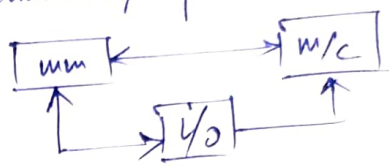
Computer Org. & Arch.

Basics

Architecture :- - CPU - I/O - Memory

CPU design
- → Addressing modes
- → Instⁿ
- → Data format → Number → fixed point, floating point
  - → Char → ASCII, EBCDI

Organization :- How the components are actually connected/implemented.

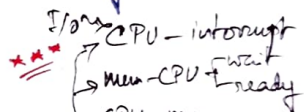


Von-Neuman Arch :- - Stored Prog Arch.

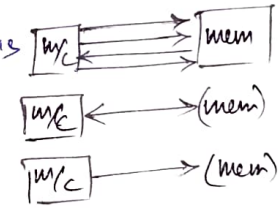
Disadv - Less throughput
- Single mem for instⁿ/data

Havard Arch :-
- Stored prog Arch.
- Diff mem for inst and data.

Disadv - At a time instⁿ and data doesn't req. simultaneously
- costly



Components of computer :-
1) CPU
2) I/O
3) Mem
4) CPU reg
5) System Bus → control bus, Data bus, Add bus



Mem Cycle time :-

MCT = mem. Access Time + Extra latency

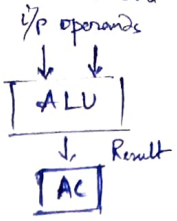
⇒ This is the time elapsed betⁿ the cpu got the request and tell it ready to accept the new signal.

CPU registers :-

(i) General purpose registers :- used to store the general contents

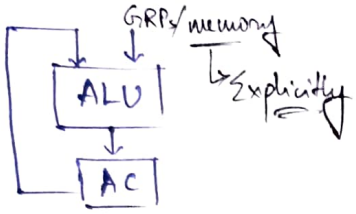
(ii) Special purpose registers :-
1) AC
2) PC
3) IR
4) SP
5) PSW
6) AR / MAR
7) DR / MDR / MBR

* **Accumulator (AC)/ACC** :- Used to store result of ALU and sometimes one of the operand for ALU also.



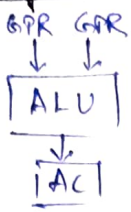
* **Types of CPU arch. based on ALU inputs** :-

① **AC based Arch.**



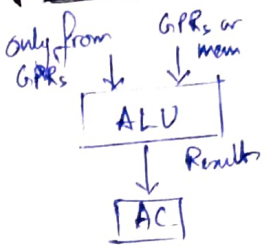
Inst^n = operand + operation

② **Reg. based Arch.**



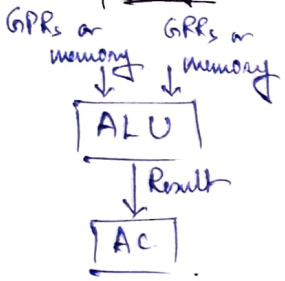
Inst^n = Operation + 2 operands

③ **Reg.-Mem Based Arch.**



Inst^n = operation + 1 operand + (other operand in fixed)

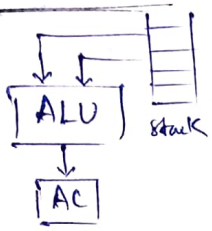
④ **Complex system Arch.**



Mem-Mem
Mem-Reg
Reg-Mem
Reg-Reg

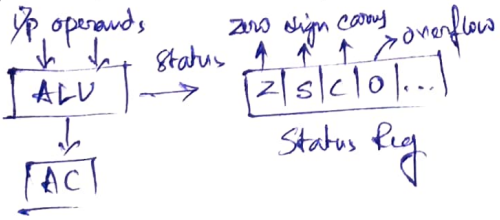
Inst^n = operation + ~~various~~ min 2 operand.

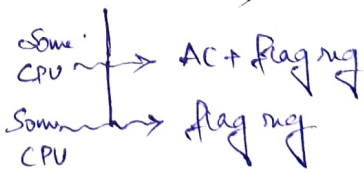
⑤ **Stack Based Arch.**



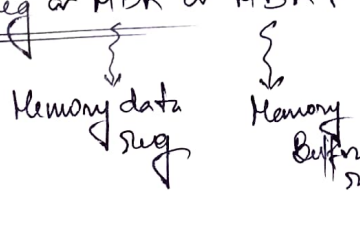
Inst^n = only operation (operands taken from tos & content of stack)

* **Program Counter (PC)** :- Stores address of next inst^n to be executed.
* **Instruction Register (IR)** :- Stores the current instruction to be executed.
* **Stack Pointer (SP)** :- Stores the address of TOP (top of the stack)
* **Flag registers or Status Reg.** :- (PSW - Program Status words)





* **Addrss Reg or MAR** :- Used to send address to memory
* **Data Reg or MDR or MBR** :- Used to send data to memory and receive data from memory



* Memory Read :

1) CPU sends add. to mem via add bus
2) CPU will enable read mem control signal.
3) Memory goes to specified address and reads the content and send it to CPU via data bus.

* Memory Write :

1) CPU sends add. using add. bus to memory.
2) CPU sends data. using data bus to memory.
3) CPU sends write control signal as enabled to memory
4) Memory goes to specified address & enters the received data/content on that cell.

Types of Arch (Based on size of I/P in ALU)

32-bit Arch.

32 bits, 32 bits → ALU → Result → A2

(max size of I/Ps) IW = 32 bits

64-bit Arch.

64 bits, 64 bits → ALU → Result → A8

IW = 64 bits

Memory type

1 Nibble = 4 bits

Byte Addressable (By default)

IW = 1B

1B
1B
1B
⋮

Word Addressable

IW = 2B
= 4B
⋮

4B
4B
4B
4B
⋮

IW = 4B

* **Micro-Operations:** The operations executed on values stored in registers.

Symbolic Notation to describe the micro-ops: Register Transfer Lang. (RTL)

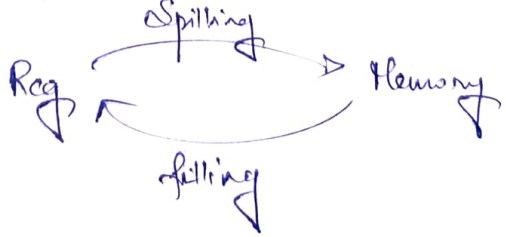
* Min time needed to perform 1 micro-op → 1 CPU cycle time.

1. Register Transfer: $R_2 \leftarrow R_1$ or $R_1 \leftarrow R_2$
2. Comma: $R_2 \leftarrow R_1,\ PC \leftarrow PC+1$
3. Memory Transfer:

Memory Read	Memory write
CPU ← Memory	M[address] ← DR
DR ← M[address]	or
M[1000]	M[1000] ← DR
M[AR]	or
	M[AR] ← DR

4. Logical μop
   - AND
   - OR
   - X-OR
   - X-NOR
5. Shift μop:
   - Logical shift
     - Left: 1011 → 0110
     - Right: 1011 → 0101
   - Circular shift
     - Left: 1011 → 0111
     - Right: 1011 → 1101
   - Arithmetic shift: After operation sign should remain same
     - Left – Same as logical shift but CPU performs it when the sign doesn't change.
     - Right – 11010 → 11101

*** **Register Spill:** If no of registers are not enough to keep intermediate operands then one or some values can be moved to memory for temporary basis.



Instruction :- Group of bits which instruct computer to perform same operation.

| opcode | operand info |

* **Instruction set Arch** :- Defines the collection of distinct inst^n. , ISA = $2^{|opcode\ bits|}$

Types of Inst^n based on Operand Info

1) **4-address Inst^n**

| opcode | Add1 | Add2 | Add3 | Add4 |

operands — Add of next inst

- used in comp. w/o pc
- disadv ☹
  - inst^n size large
  - inst^n fetch takes more time
  - Relation of prog is costly

2) **3-address Inst^n** : max^n 3 add. can be specified in an inst^n

| opcode | add1 | add2 | add3 |

operands

- used in modern comp.

3) **2-address Inst^n** max^n 2 add. can be specified in an inst^n

| opcode | add1 | add2 |

- one of the operand is used as both src and dest^n
- more inst^n for a prog. compared to 3-add inst^n

4) **1-address Inst^n**

| opcode | add |

- AC is used as 2nd operand implicitly.

5) **0-address Inst^n** No any add. is specified within an inst^n

| opcode |

- both operand taken from stack
- implemented on stack-based Arch.

Instruction

- fixed length inst^n → opcode variable length
- variable length inst^n → fixed opcode length.

Fixed Length instruction

- Start with that inst^n which has min^n opcode bits

CPU receives an inst^n | 10 10 10 |

2-add inst^n | 10 | 10 | 10 |

used opcode - 00,01,10 for 2 add inst^n

CPU check if opcode 10 has been used for 2-add inst^n or not

used → Take it as 2 add inst^n

not used

Total = $2^2 = 4$

used opcode = 3

1

1-addrss. | 10 10 | 10 |

* Only those inst^n which are having starting 2 bit 1!.

* max = (1) $\times 2^2 = 4$ inst^n possible

used opcode $\times 2^{n-m}$

n = opcode size in 1-add

m = opcode size in 2-add.

- min will always 1. bcaus if 0 then that inst^n not supported.

Variable length instr

Q) Consider 3 types of inst^n in system.

1. Register operand inst^n : One opcode and 2 registers.
2. Memory operand inst^n : One opcode, 1 register and 1 memory address
3. Immediate Operand inst^n : One opcode, 1 register and 1 immediate operand.

Number of registers = (64) → 6

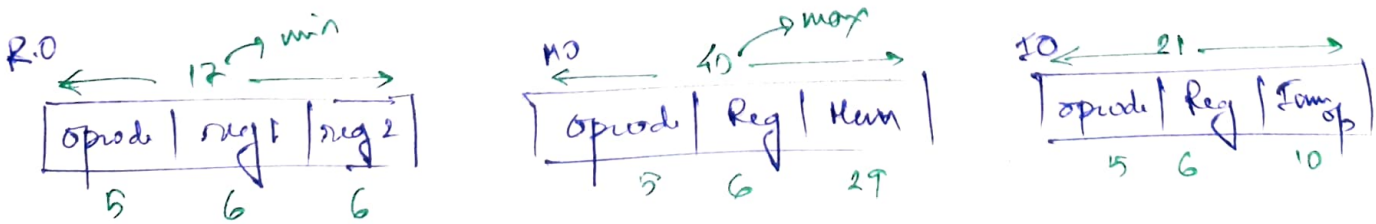
Number of bit in immediate operand = 10 bits

Memory size = 512 Mbytes (byte addressable) → $2^9 \times 2^{20} = 2^{29}$

Total inst^n : (1) Reg Operand type : 10
(2) Memory Operand type : 12
(3) Immediate Operand type : 4

(26) $= 2^5$ opcode = 5 bits

Maximum and Minimum inst^n length are? Amb 40,17.



Effective Address: Address of operand in computation type inst" (ADD, SUB, ...)
(or)
The target address in a branch-type inst".

Branch type inst"
- Unconditional
  - ex: goto
  - always jumping
  - always change PC value by target address.
- Conditional
  - ex: JNZ, JZ, BNZ
  - Jump only when cond" is true
    - Condition is true (Branch is taken) → PC updated by target address
    - Cond" false (Branch is not taken) → No change in PC value

* Instruction Cycle: 6 phases to execute one inst".

Inst" fetch (IF) ⇒ Inst" Decode (ID) ⇒ Effective Address Calculation (EAC) ⇒ Operand fetch (OF) ⇒ Execution (EX) ⇒ Write Back Result (WB)

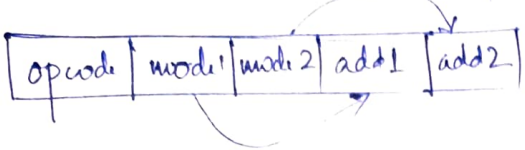
→ PC increments during IF and it incremented by the size of inst" which is fetched.

Inst" Cycle
- Fetch cycle (IF)
- Execution Cycle (ID, EAC, OF, EX, WB)

→ Execution Cycle – 5 phases
→ Execution phase – 1 phase
→ Inst" Execution – all 6 phases

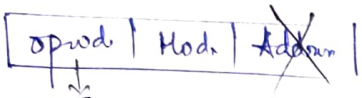
Addressing Modes

It specifies how to use the address field of inst^n to get the operand.



* We use addressing modes to calculate eff. add.

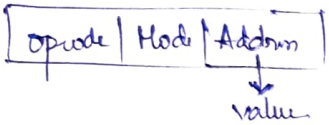
1) Implied mode :- Opcode definition itself specifies the operand.
- No any explicit address specified for operand.



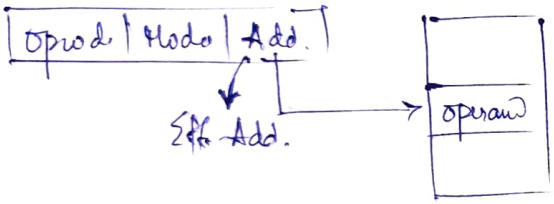
operation + operand.

opcode ⇒ Increment Acc.

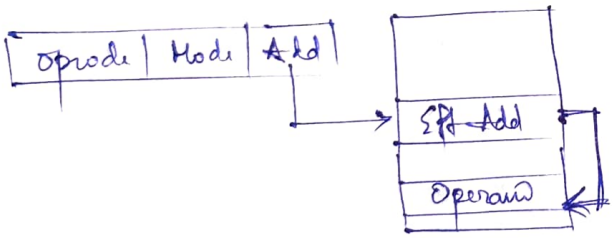
2) Immediate Mode :- Add field of inst^n specifies the operand value.
- Used to initialize reg with const values.



3) Direct mode :- / Absolute Mode :- Add fields of inst^n specifies effective address.
- Easy to implement for mem operands.

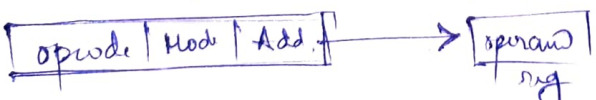


4) Indirect mode :- The add field contain the add of eff add.

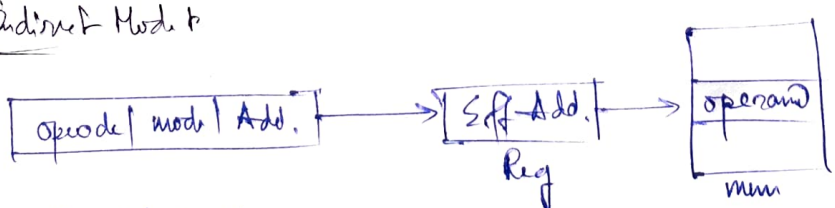


- 2 mem access to get the operand
- Used to implement pointers

5) Register Mode :- Add field of the inst^n specifies a reg which contain the operand.



5.) Register Indirect Mode :-



- use to shorten the inst length.

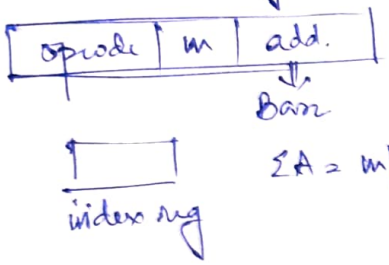
⑦ Auto inc/dec mode : (Auto indirect)

- Variant of reg indirect mode, but content of reg (eff. add) is automatically incremented/decremented to access a table of constant (array) sequentially.

⑧ Index mode / Index Register Mode :

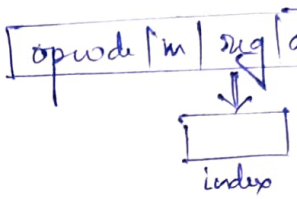
2 implementation

CPU have special purpose index reg.



EA = m[add] + content of index reg

CPU doesn't have specific index reg.

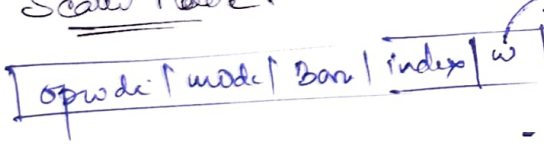


EA = m[add] + content of the index reg field.

- If prog data is relocated, the new base add has to be updated in inst"

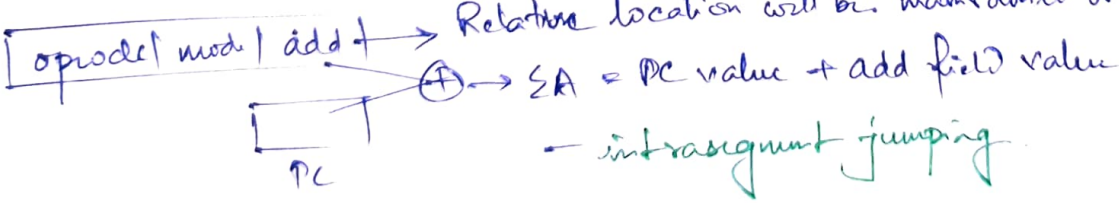
⑨ Base, Index Reg mode : - Keep both index, base in reg only
- EA = base reg value + index reg value

⑩ Scaled Mode :



- Advance version of index mode
- EA = base + index * w

⑪ PC relative Mode : Particularly used for branch inst"



Relative location will be maintained or offset

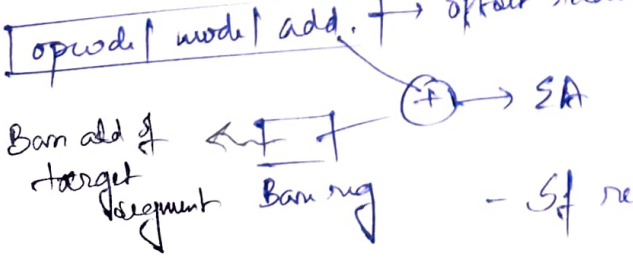
EA = PC value + add field value

- intrasegment jumping

+ve value if forward jumping

-ve value if backward jumping

⑫ Base - Reg mode :



offset relative to base add of target segment

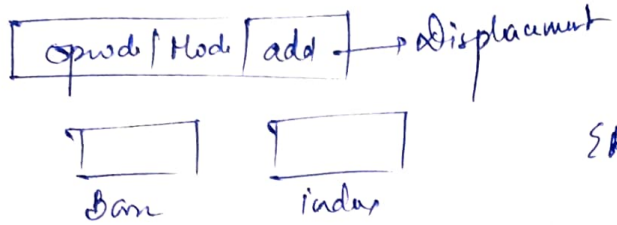
EA

- used to jumping but base, index is used to access array elements

- If relation happens no need to change the inst"

13) Base, Index Register Displacement mode :

- It is used to access array of structures



To reach specific member of the structure

EA = BA + index + displacement

To reach specific array elements

NOTE: (11) & (12) are used for jumping / branching other are used for data.

14) Absolute Addressing mode :

- The address of the operand is inside the inst^{n}.

NOTE:

The effective address of the following inst^{n} is MUL 5(R1, R2)

↳ 5 + [R1] + [R2]

The addressing mode used is base with offset and index.

CPU organisation

CPU - Basics

* **CPU cycle time :-** Amount of time Req. by CPU to perform a small operation (micro-operation).
* **CPI :-** # CPU cycles req. to execute an Instruction. = No. of CPU cycles to execute one Inst".
* **CPU clock rate :-** $\frac{1}{\text{CPU cycle time}}$
* **Execution time :-**

1 Inst" execution time = CPI avg * cycle time

n Inst" execution time = n * CPI avg * cycle time

n Inst" execution time = $\frac{n * CPI_{avg}}{\text{clock rate}}$

$$CPI_{avg} = \sum_{i=1}^{K} CPI_i * n_i \Big/ \sum_{i=1}^{K} n_i$$

* **MIPS :-** (Million Inst" per sec)

If n inst" executed in t time, $MIPS = \frac{n}{t * 10^6}$

$$MIPS = \frac{\text{clock rate}}{CPI_{avg} \times 10^6}$$

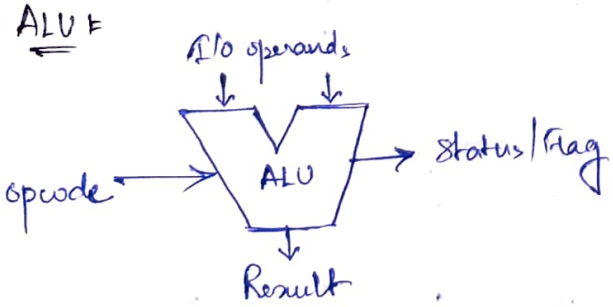
(*) CPI may not be same for all inst" hence consider CPI avg.

(*) Disadv : MIPS doesn't consider the complexity or capability of the inst"

↳ Because of this reason FLOPS is used.

↳ floating point operations per sec.

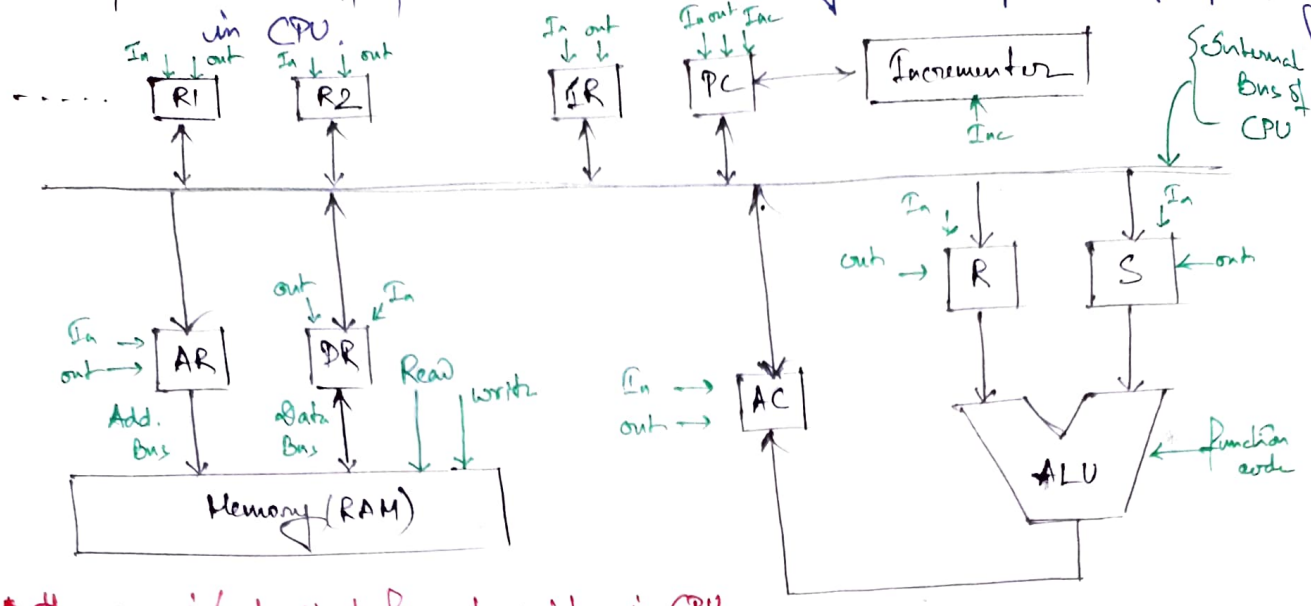
* **ALU :-**



(*) 32 bit CPU, ALU, that takes 2, 32 bit input operation can be done in single cycle.

(*) 64 bit architecture is faster than 32 bit architecture.

* **Datapath :** A set of functional units to carry on data processing operations in CPU.



* there are in/out signal for each register in CPU.

⇒ **Instruction fetch :** Bring an inst^n from mem to IR in CPU.

1. $AR \leftarrow PC$ — AR_{IN}, PC_{out}
2. $DR \leftarrow M[AR]$ — $AR_{out}, mem_{Rd}, DR_{in}$
3. $IR \leftarrow DR$, $PC \leftarrow PC+1$
   - $IR \leftarrow DR$: DR_{out}, IR_{in}
   - $PC \leftarrow PC+1$: PC_{INC}, Incrementer$_{Inc}$

* **Control Unit :**

It generates control signals and sends those signals to respective unit. Then the unit performs its respective operation

Control variable : The name of specific control signals

Control word : Collection of all control signals generated by control unit

Control unit Org.

- Hardwired CU
- Microprogrammed CU
  - Horizontal
  - Vertical
  - nano

Control word

eg.

	IR		AR		D1		D1		MUX1	MUX2	PC	ALU	mem	
	In	out	In	out	In	out	In	out	Select bin	Select bin	In out Inc	opcode	R	W
1. AR ← PC	0	0	1	0	0	0	0	0	1	1	0 1 0	1 1 0 1 0	0	0

↳ This is actual word that is responsible for μ-op $AR \leftarrow PC$

Hardwired Control Unit :

The control logic is implemented using the hardwares.

Adv :- Faster mode of operation

Disadv :- control logic updation is diff.

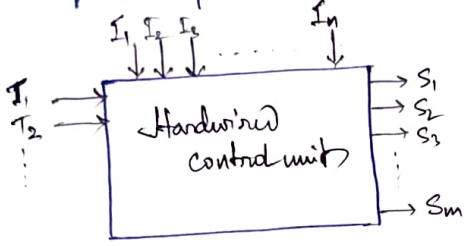
CPU can execute 'n' types of instruction $(I_1, I_2, \ldots I_n)$

$I_1 \Rightarrow$ μ-operation (1, 2, 3. . . . 10)

$I_2 \Rightarrow$ ... (but diff seq. of μ-op.) ...

(*) for each μ-operation a control word is generated (unique)

$S_i = (\text{interms of } I_i \text{ & } T_i \text{ Reqd})$



eg. $S_1 = T_1 I_1 + T_2 I_3 + \ldots$

$S_2 = T_1 (I_1 + I_2) + T_3$.

Microprogrammed Control Unit :

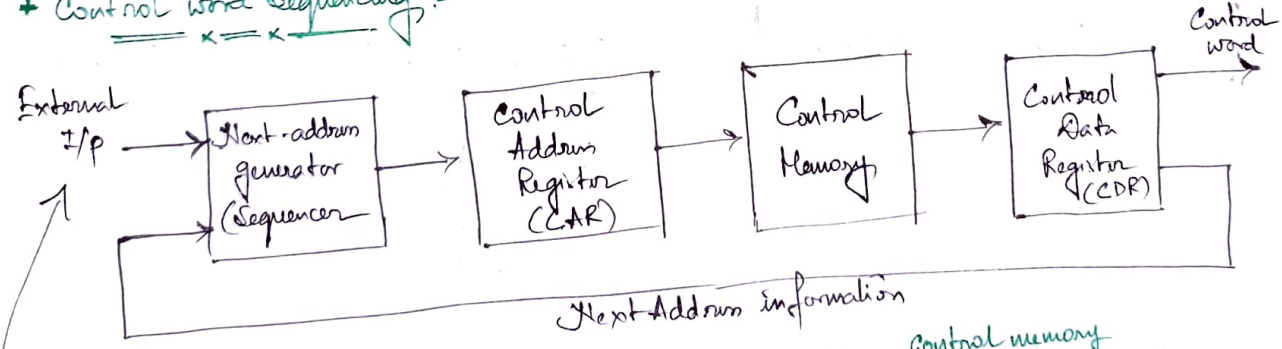
(*) Control memory stores every possible control word. Based on requirement, the corresponding control word is fetched and the signals are sent to respective unit.

Adv :- Easy updation

Disadv :- Slower (+ mem acc.)

(*) Control words will stored inside ctrl. mem. which is attached with ctrl. unit and it is inside CPU.

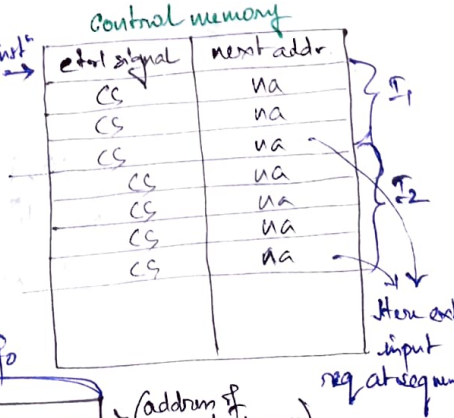
Control word Sequencing :



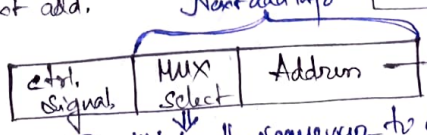
(*) all μ-operation for a instruction is stored in seq. $\Rightarrow$ No need of addr. calculation $\rightarrow$ [Faster]

(*) at each addr. in ctrl mem.

This is for the last control word of an inst^n that needs external input to know what will be the next add.



Micro-inst^n (Basic format) :

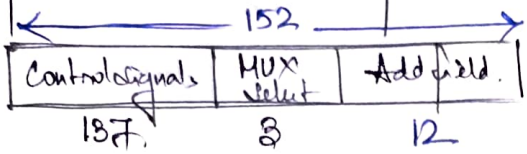


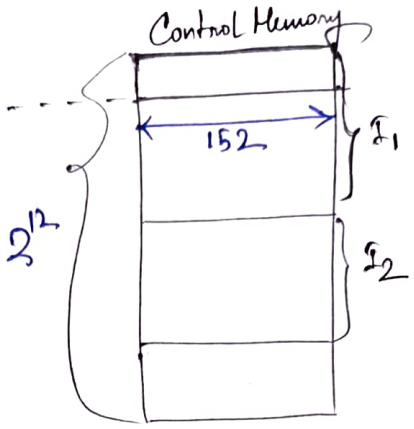
This will help the sequencer to decide which option will generate next address (external or control mem)

Q> A system supports 256 distinct inst". Each inst" requires a sequence of 16 micro-operations. Each micro-op" in control memory has format:

Control mem size = ? (bits)

Given:





dist" inst" × micro op"

= 256 × 16

Total micro-op" = $2^8 \times 2^4 = 2^{12}$ → Add size = 12 bits

Control mem size = $2^{12} \times 152$ bits

= 608 K bits

* Types of Microprogrammed Control Unit:

- {Horizontal}
- {Vertical}

{Horizontal}

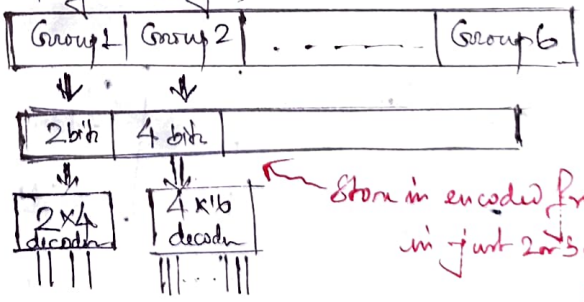
In control word, 1 bit stored for each control signals

{Vertical}

all signal, are divided into multiple groups and each group info is stored in encoded form.

Each group such signals among such signals among those only one can be enabled at a time.

4 signals, 16 signals,



Store in encoded form in just 2 or 3 or 4... bits

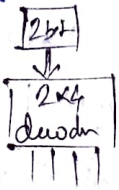
Speed:

Hardwire > horizontal > Vertical

(default case)

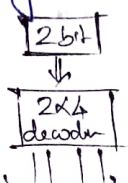
Every time atleast 1 signal in active from each group

eg. 4 signals in a group



It might be possible that no any signal is active from a group (atmost 1) at a time

eg. 4 signals in a group



no one is active

Horizontal

1) One bit for each signal
2) No decoder required
3) Control word is large
4) Faster
5) Max no of signals enabled at a time = total no of signals.
   (Degree of parallelism is high)

Vertical

1) Signal stored in encoded form in groups.
2) decoder req.
3) Small. - CW.
4) Slower
5) Max no of signals enabled at a time = No of groups.

NOTE: If any signal cannot be kept in any group then such signals are stored in horizontal manner.

RISC

(Reduced Instruction set computer)

1) Less Number of Inst^n supported.
2) Fixed length inst^n
3) Simple inst^n
4) Simple and less number of addressing modes.
5) Easy to implement using hardwired control unit.
6) One cycle per inst^n
7) Register-to-register arithmetic operation only
8) More number of registers.

CISC

(Complex Instruction set computer)

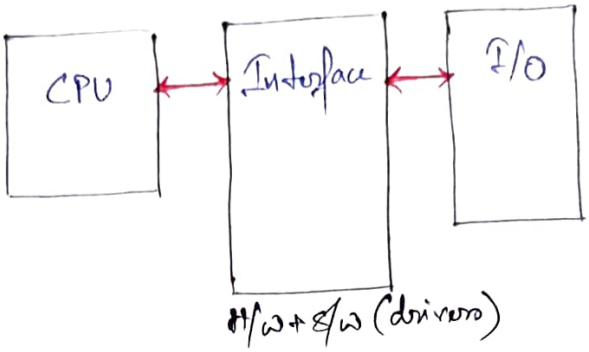
1) More number of Inst^n.
2) Variable length inst^n
3) Complex inst^n.
4) Complex and more number of addressing modes.
5) Difficult to implement using hardwired control unit.
6) More than one cycle per inst^n
7) Register to Memory and Memory-to-reg
8) Less number of registers.

⇒ In nano programmed ctrl unit the repeatative ctrl signals are not repeated, unique ctrl signals are stored in another mem and in control mem we store the pointer that will take less time.

I/O Organization

* **Peripherals** : devices connected to processor internally.

- Mainly 3 types:
  - Input
  - Output
  - Storage (persistent)



* **Need for interface**:
  - conversion of signal reqd.
  - synchronization reqd.
  - conversion of format reqd.
  - peripheral should'nt distrubs operation of other.

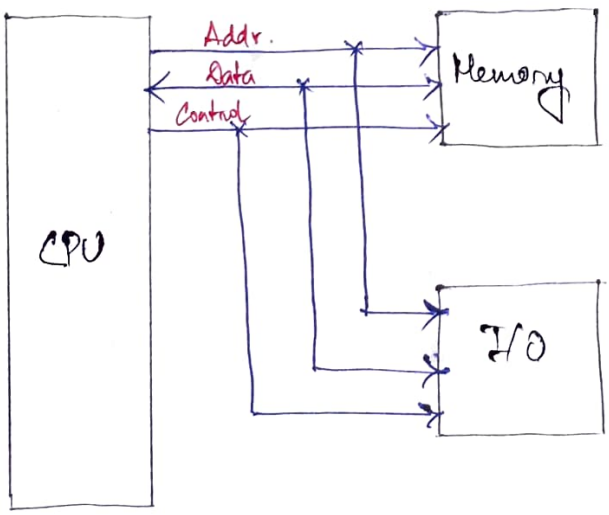
* **I/O processor** { Interfacing + DMA controller + I/O inst^n execution.

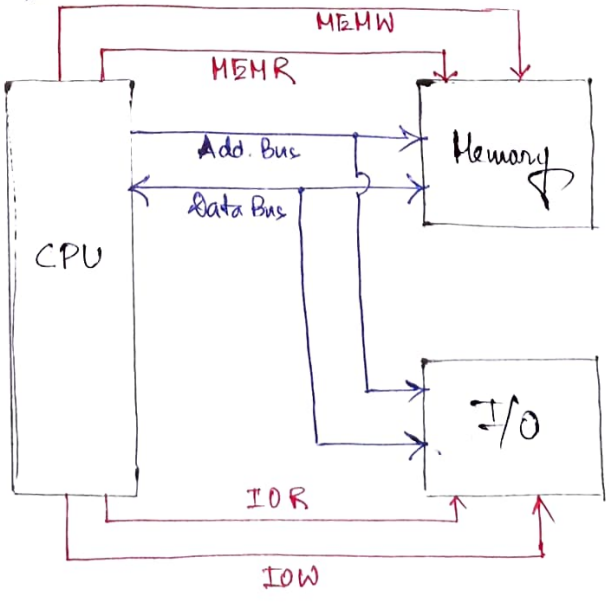
Memory Mapped I/O

1) Memory wastage.
2) All memory accen inst^n used for I/O accen also.
3) No separate add. space for I/O.
4) More inst^n for I/O accen.
5) More addressing modes for I/O accen.
6) More I/O devices connected.

I/O Mapped I/O.

1) No memory wastage.
2) I/O accen and memory accen. inst^n are different.
3) I/O have their own separate address space
4) Less inst^n for I/O accen.
5) Less addressing modes for I/O Accen.
6) Less I/O devices connected.





Data Transfer

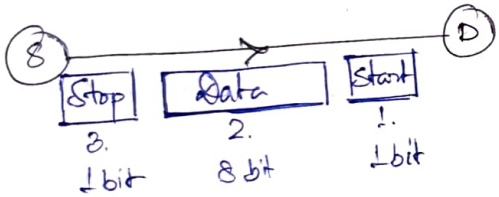
Synchronous vs Asynchronous Transfer :

- **Synchronous** : All components/parts working w/ same clock pulse.
  - * i.e. it will work with the speed of the slowest.
- **Asynchronous** : Each device will have its own independent clock signals.
  i.e. synchronization has to be provided externally.
  - * In synchronous : **Bottleneck = SLOWEST device**

* Serial vs Parallel :

- **Serial** : Single comm link b/w 2 device & 1 bit at a time.
- **Parallel** : B/w 2 devices, there are multiple lines.
  If there are K lines b/w them, then we can send K-bits at a time.

* Serial Asynchronous Data transfer :



(*) Since method is asynchronous, we need to provide external synchronization.

⇒ Modes of Data Transfer :

1. Programmed I/O. } CPU ↔ I/O
2. Interrupt driven I/O. }
3. Direct Memory Access. } MEM ↔ I/O

Programmed I/O : ~~X~~

- More importance given to I/O
- Wastage of CPU time for asking each and every device after certain time interval.
- If any device has its status set then CPU performs data transmission for it.

Total time = Time to req to check the status of I/O + Data Tx time

- (Flag req sending time from I/O) + (Flag checking time in CPU) — negligible
- Data Tx time → Depends on I/O speed.

2> Interrupt driven I/O: * Starting Add. of ISR is called Vector Address.

ISR():
1. CPU received interrupt.
2. CPU completes execution of current inst^n.
3. Return addr. (+ state) stored in CPU stack.
4. Interrupt. Serviced.
5. Using POP stack - Original Program restored.

Interrupt.

Vectored
= Send interrupt along with addr. of ISR.

Non-vectored
* CPU only receive intr. but no ISR addr.
* CPU executes default ISR, which helps to find the Correct ISR.

* Types of Interrupts:

(i) Maskable: (Low Priority) { CPU can Accept or reject } on situation
(ii) Non-Maskable: (high Priority) { CPU always accept }

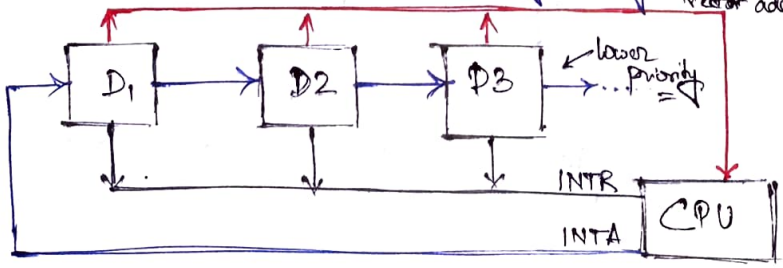
(Based on source)
(i) External: (H/w) devices
(ii) Internal: (S/w) during instruction execution.

* Interrupt overhead*:
1. push the value into stack
2. Resolve priority
3. Branching to ISR.
Everything includes before servicing the interrupt.

* Priority Based interrupt Handling:

S/w Sol^n
(Polling is used)
Check [High.... Low)

H/w Sol^n
Serial (Daisy chaining)
Parallel..



* Here we can't change the priority of the devices.

- Starvation may occur (Practically very rare)
(*) give highest priority to the device which is electrically closer to CPU.

Total time = Interrupt overhead* + data transfer

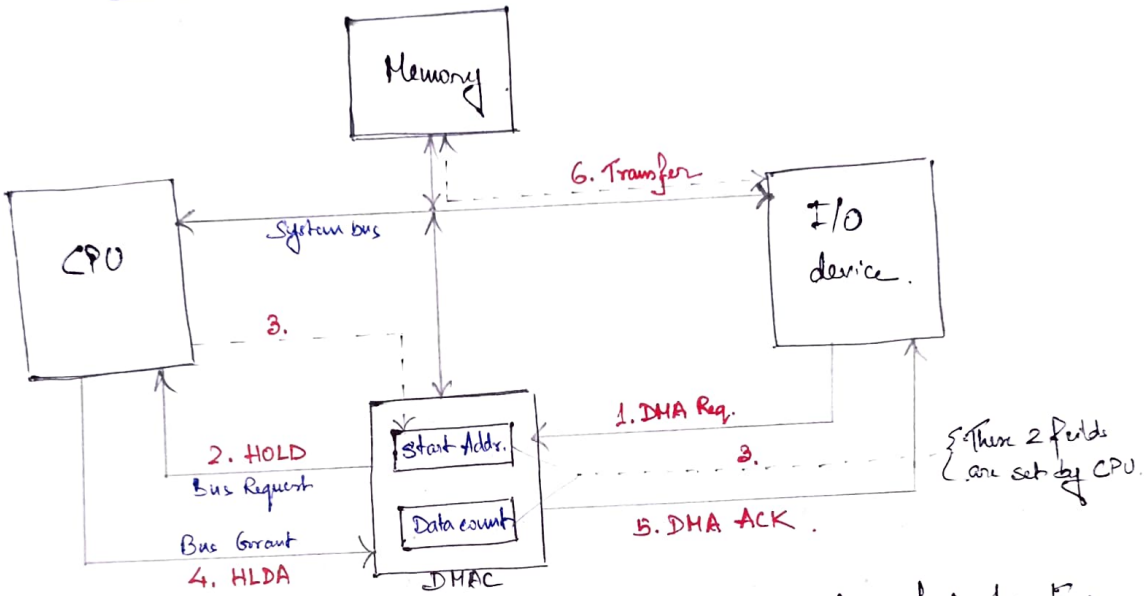
Interrupt Cycle :- If interrupt of a device is not accepted, then after how much time again the interrupt is generated.

* In the question the device transfer speed represents the internal speed of the device not the tx speed before main I/O or CPU.

- Fraction of processor time consumed by I/O device = $\frac{1 \text{ B time}}{\text{interrupt processing time}}$

3) Direct Memory Access (DMA) :-

- Tech enables data transfer b/w I/O and mem w/o CPU intervention.
- Need a H/W - DMAC.



(*) the CPU can grant HLDA at any point of time (at any phase of instruction execution)

⇒ During DMA transfer, CPU can only perform those which do not require system Bus.

(*) DMA is also special kind of processor.

Time = DMA initialization + Data Transfer.

* Modes of Data Transfer :- CPU can perform only those operations which don't require system buses, during DMA tx. which means CPU will be block mostly.

(i) Entire amount of data at a time (BURST Mode) → 2 implementation possible → (1) Full data (2) 1 blk at a time

(ii) One block at a time (BLOCK Transfer mode)

(iii) Cycle stealing.

- How much data is transfer in one time through DMA. before CPU takes back the ctrl of the buses.

* Cycle stealing :- Slow I/O device takes some time to prepare and to transfered. During that time CPU keeps the control of the buses. When the data in ready CPU gives the control of the buses to DMA Controller for "1 word cycle". On that cycle, prepared word is transferred to memory. And then CPU takes back the control of the buses. *

Assume,

I/O takes time to prepare data = t_x.

Data transfer time to memory = t_y

$$\% \text{ of time CPU is blocked} = \frac{t_y}{t_x + t_y} \times 100\% \quad \text{[Burst mode]}$$

$$\% \text{ of time CPU is blocked} = \frac{t_y}{t_x} \times 100\% \quad \text{[Cycle stealing mode]}$$

$\therefore$ t_y time will overlap w/ t_x time.

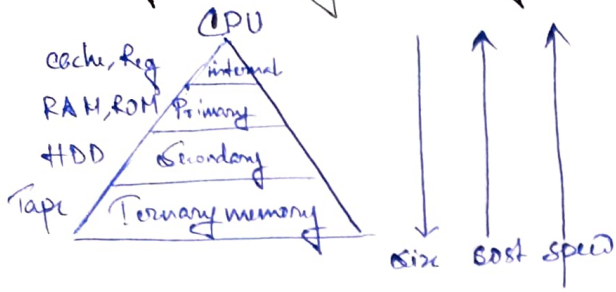
ie when t_x is larger than t_y, then : $\{t_x > t_y\}$

* Interleaving DMA :- CPU gives the control of the buses to DMAC only when it does not need the buses (it operates internally)

(here 0% CPU blocked due to DMA)

Memory Organization

* **Memory & Memory hierarchy :**



* Maximize memory accen speed
* Minimize per bit storage cost
* { S_i → size of memory M_i }
  { C_i → per bit cost of M_i }

* Average per bit storage cost $= \frac{S_1C_1 + S_2C_2 + S_3C_3}{S_1 + S_2 + S_3}$

* Memory Accen rate $= \frac{1}{\text{memory accen (cycle) time}}$

* **Memory Representation :** = no of cells × 1 cell size (capacity)
  = No of mem addr. (location) * no of bits in 1 location

Addren size = $\log_2 N$

- If 1 cell capacity = 1B ⇒ Byte Addressable mem (By default)
  (Addren assigned per Byte)

Memory → Word Addressable
       → Byte Addressable

- CPU 32 bit Archi ⇒ 1 word = 32 bit = 4B

* Default storage unit = bit

* **Main Memory :** Mem to store current running progs. and their data.

* **Memory Decoding :** addren size × Total no of addresses in the decoder size

→ ~~Random access Memory (RAM)~~

- Non volatile
- 2 job it is doing while the comp. gets on → 1) H/w check (Power-on self test)
  → 2) Booting : Bootstrap loader is in ROM
  - B prog. is in 2ry mem,
  - After Booting over. CPU doesn't use ROM till the comp turned off.

→ Read Only Memory (ROM)

→ Random access Memory (RAM)

↳ Used to store running prog. and the data along with as running prog.

* Initially CPU executes instruction from ROM (POST)
  After POST → [Booting Process] → Load OS into RAM. (Bootstrap)

RAM : Bootstrap ROM : Bootstrap loader.

* **Types of ROM :**

(i) EROM (Erasable)
(ii) PROM (Programmable)
(iii) EPROM (Erasable - Programmable)
(iv) EEPROM (Electrically Erasable Program)
(v) EAPROM (Electrically Alterable ...)

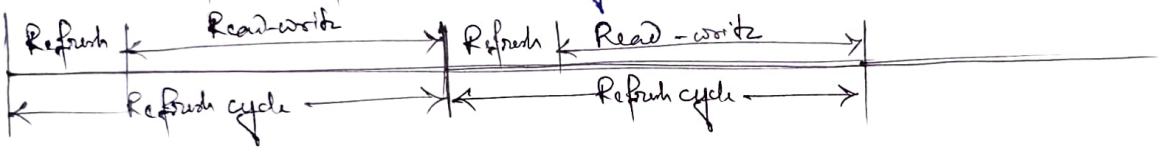
* **Types of RAM**

(i) Static
(ii) Dynamic.

Static	Dynamic
1) Implemented using flip-flops	1) Implemented using capacitors
2) No refresh required	2) Periodic refresh is required.
3) Faster Read/Write	3) Slow Read/write
4) Used for cache	4) used for main memory
5) Low idle power consumption	5) High Idle power consumption.
6) High operational power consumption	6) Low operational power consumption.

* DRAM & SRAM Refresh Logic & cell Arrangements :

- In 1 refresh 1 row of cells can be refreshed.
- Time to refresh complete chip = no of rows of cells × 1 refresh time
- Time to refresh n no of chip = 1 chip refresh time, All chips can be refreshed parallelly.
- Whenever refresh is going on, no r/w can happen in the chip. One entire chip is refreshed then only R/W can be done.



- Main memory is 16 bit wider, cell capacity = 16 bits or at each address data req by CPU is 16 bit (Arrangement includes here)

*

A	B	A*B result	A*B mul. table	A+B Add" table	A+B result
n bit	n bit	2n bit	$2n \times 2^{2n}$ bit	$(n+1) \times 2^{2n}$ bit	(n+1) bit

* No of pins required in RAM, ROM (chip select, Read, write, data, add line. in RAM, (CS, Data, add in ROM)

* Using multiple chips in a mem system :

Case 1: Total no of addresses increases : (Vertical + Decoder)

Case 2: Total no of data increases (word size) : (Horizontal + no decoder)
At same time addr. line provided to both RAM

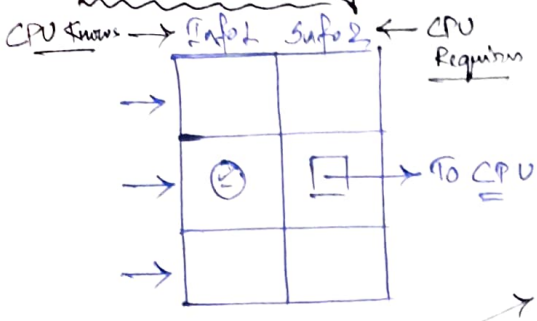
Case 3: Both required Data & Addr more : (Hybrid Arrangements)

128 × 8 bit ⇒ 7 bit Add
128 × 8 bit ⇒ 7 bit Add

256 × 8 bit ⇒ 8 bit Add line ⟶ Among this 8 bit 1st bit is used as chip select.

* In multiple chip memory to decode output select one entire horizontal arrangements.
* CPU can initiate the memory request only when memory is ready.

Associative Memory :- (Content-Addressable mem.)



- costly, fast, used in Cache & TLB.
- If data matched in more than one place then the content will send sequentially.
- parallel comparisons req.

* CPU always generates mm address (even to access cache too)

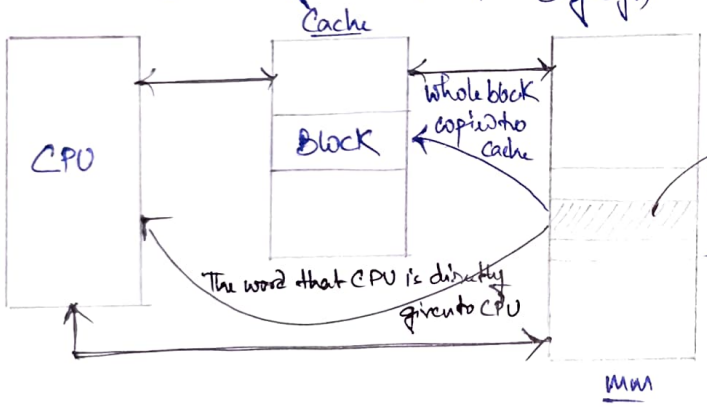
* **Locality of Reference :-** → Temporal (Time) / → Spatial (Space)

If CPU has requested one addr. for mem. access then that particular add. or nearby addr will be accessed in further.

Temporal (Time) / Spatial locality (space)

* **Cache Memory :-** Based on locality of reference concept current demanded locality are kept into the smaller & faster mem known as cache memory.

* when miss in a cache occur a block containing demanded content is brought to the cache. (Birthday, taking help of a small boy eg:)



Demanded Word (HM)

CPU ↓ Cache

* this is done parallely.

Requested Byte/Word that is missed in cache.

Block

$$\text{hit ratio} = \frac{\#\text{hits}}{\#\text{References}}$$

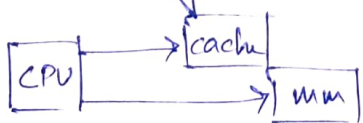
$\therefore$ miss ratio $= (1-h)$.

* **Average Memory Access Time :-**

$$T_{avg} = H * \text{Time when hit in cache} + (1-H) * \text{time when miss in cache} \quad \text{—— (1)}$$

Simultaneous Access :- (By Default)

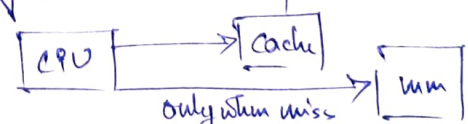
* Req for mm and cache are generated simultaneously.



$$* \; T_{avg} = H * T_c + (1-H) T_{mm} \quad \text{—— (2)}$$

Hierarchical Access :- ["level", "hierarchy"]

- only cache is access first



$$* \; T_{avg} = H * T_c + (1-H)(T_c + T_{mm}) \quad \text{—— (3)}$$

$$* \; t_{avg} = t_{cm} \text{ if } H = 100\%$$

* Every time there is an read miss, a block is brought from mm to cm.
* Every time there is a write miss, a block doesn't come from mm to cm (no write allocate)

When to use which formula

Check T_{cm}, T_{mm} given

- No → Use general formula Eqn - (1)
- Yes → check "level" or "Hierarchical" given
  - No → Simultaneous Access Eqn (2)
  - Yes → Hierarchical Access Eqn (3)

NOTE: Cache lookup time for hit/miss is ignored.

* T_{avg} when locality of Ref. is used : [Of a block accessed in case of cache miss]

* Simultaneous

$T_{avg} = h(t_c) + (1-h)(t_{block})$ (block access time)

* Hierarchical

$T_{avg} = h(t_c) + (1-h)(t_c + t_{block} + t_c)$

* t_{block} = Block size * t_{mm}

* T_{avg} when transfer time included :-

* Simultaneous

$T_{avg} = h(t_{cm}) + (1-h)(t_{block} + t_{cm})$

(t_{cm}: To write in cache)

* Hierarchical

$T_{avg} = h(t_c) + (1-h)(t_c + t_{block} + t_c)$

* In one t_{cm} whole block in cache is accessed but in t_{mm} only one word in mm can be accessed not whole block.

* if in ques transfer time is given directly then that includes T_{cm} (write time) + T_{mm} × Block size (Block access time) (Reading time).

- Cache writes or Write Propagate : [Original content always must be there in mm]

CPU update the duplicate copy in cache mem but the mm copy is not updated this is called cache write issue.

(1) Write through :-

The updation of mm content is done immediately/simultaneously along with cache.

Adv :- mm always contain the correct updated value. (mm contains consistent data)

Dicadv :- Wastage of time (Time consuming process)

* for write operation mm is accessed irrespective of hit or miss

(2) Write Back :-

immediately we don't write. Dirty/modified bit maintained to denote change.

Do write operation during block replacement.

Adv :- Time saving (Faster) (In case of cache hits, mm is not accessed)

Disadv :- Inconsistency in data.

Memory Co-herence Problem :- 2 devices [CPU/I/O] accessing same content from 2 diff mem.

* Write Allocate / No Write Allocate :-

Miss

Read
Always bring missed block from mm to cache

Write
→ Write Allocate :- Bring block from mm to cm.
- for better performance write back use this.
→ No write Allocate :- the block is modified in the mm and not loaded into the cache
- for better performance write through use this.

* Tavg in Write through Cache :-

1) $T_{avg(read)} = H * t_{cm} + (1-H) * t_{mm}$ [Simultaneous]

or

$H * t_{cm} + (1-H)(t_{cm} + t_{mm})$ [Hierarchical]

2) $T_{avg(write)} = \max(t_{cm}, t_{mm}) = T_{mm}$

3) $T_{avg(overall)} =$ fraction of read $\times T_{avg(read)} +$ frac^n of write $\times T_{avg(write)}$

4) Effective hit ratio = read hit ratio × % of read.
[Because incase of write we always access the mm]

- In write through the whole block is not get transferred, the amount of data CPU want to update is sent to mm.

* Tavg in write Back Cache :- [Implicitely we talk about locality of reference]

1) $T_{avg\ (read/write)} = H * t_{cm} + (1-H)(t_{block} + t_{write\ back})$ [Simultaneous]

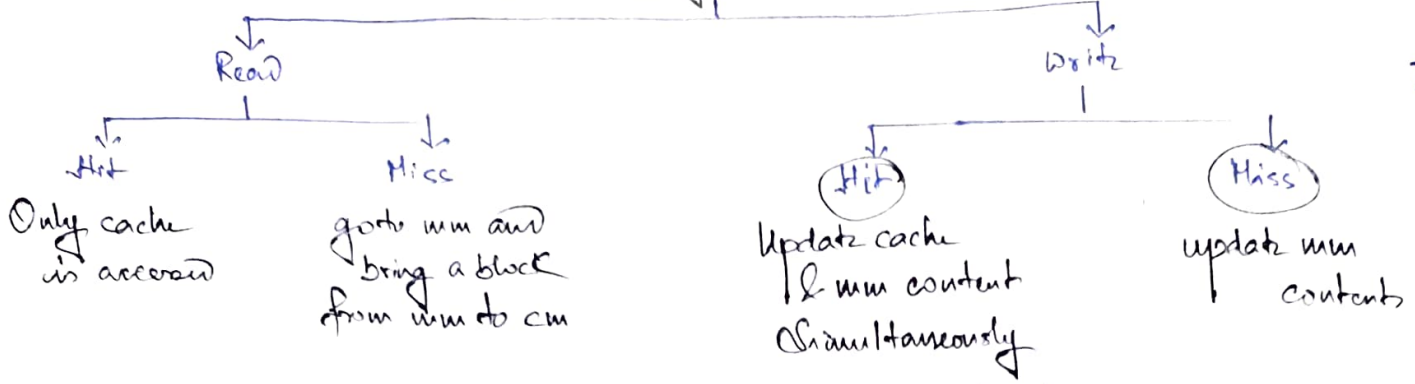
2) $T_{avg\ (read/write)} = H * t_{cm} + (1-H)(t_{block} + t_{cm} + t_{write\ back})$ [Hierarchical]

3) $t_{write\ back} = x * t_{block}$.
↓
% of dirty blocks

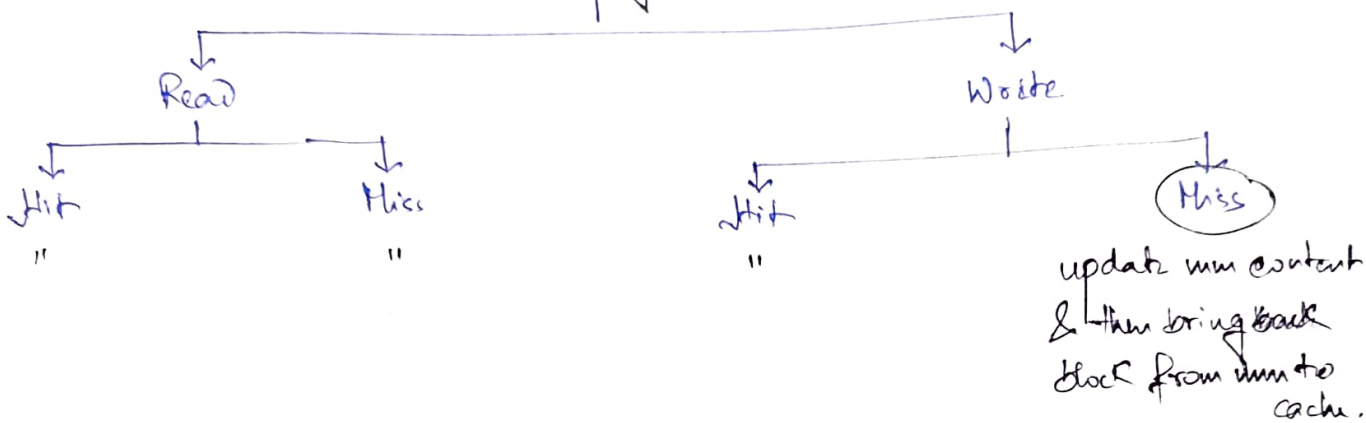
* Bus usage in Write Hit/miss :- Bus usage is come into picture when mm is accessed not cm.

(*) Only 1 data sent to mm for write in write through cache.
(*) In write through cache the block is replaced from cache directly
(*) In write back cache, the dirty blocks are only written back to mm.

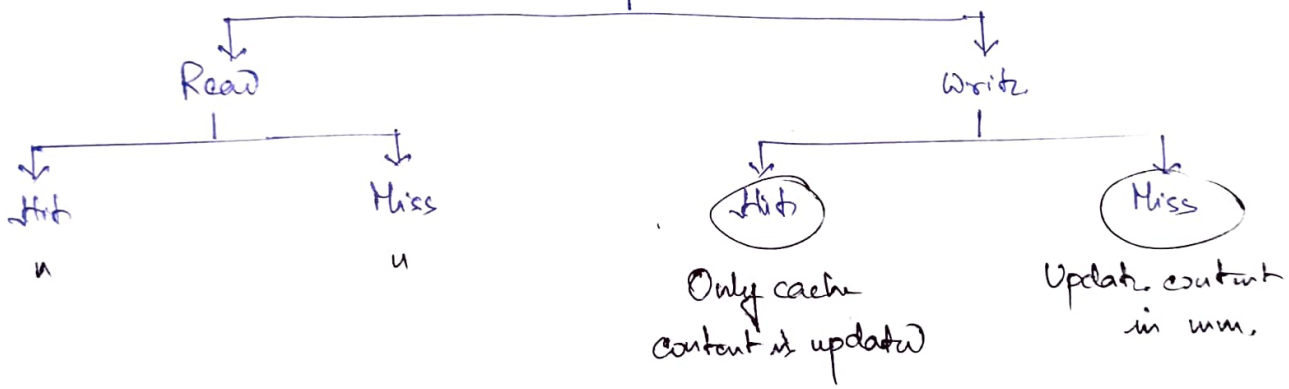
Write through with no write Allocate.



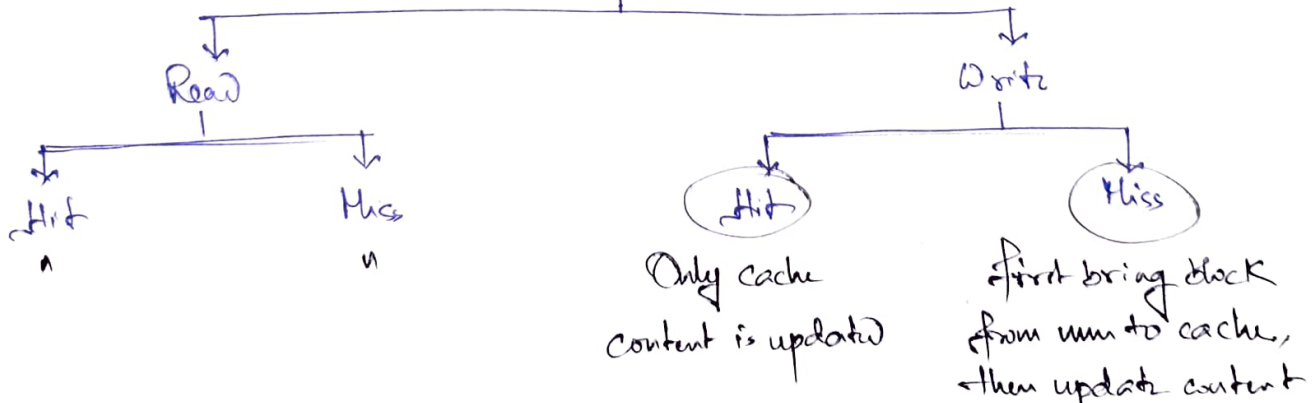
Write through with write allocate.



Write back with no-write allocate.



Write back with write allocate



Cache Mapping

→ CPU always generates mm address (even to access cache too)

→ Transformation of mm addr into cache memory addr is known as Cache mapping.

⇒ Mapping is applied on Blocks.

(i) Direct Mapping
(ii) Set-associative mapping.
(iii) Fully associative mapping.

* valid bit – 0 – invalid – no checking of tag
  – 1 – valid – checking of tag
* Dirty bit / modified bit – 0 – no update
  – 1 – update

⇒ Direct Mapping :

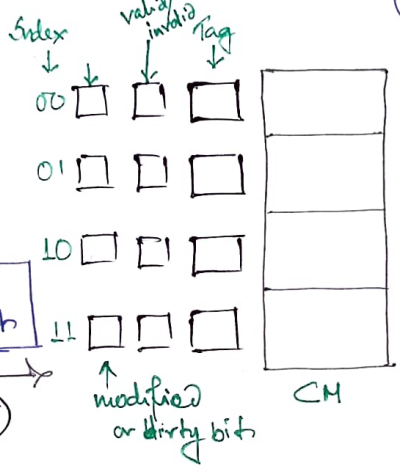
* CM block number = (MM block number) % number of blocks in cache.
* No of blocks in cache = Cache size (CS) / Block size (BS)
* If Block size = 2^K B, no of bits in Byte offset = K bits.
* MM Block No.

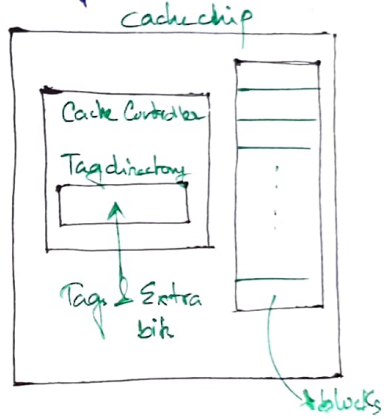
Tag	CM block No.

MM address

Tag	CM block no.	Byte offset

$\log_2$ (cache size)



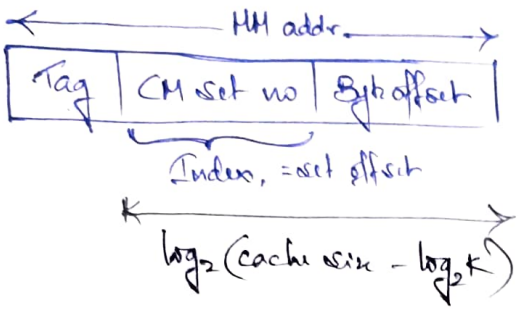


* Index in direct Mapping = CM block number
* Tag in direct Mapping = MM address – $\log_2$ (cache size)
* Tag identifies among all MM blocks which maps to one index, which one is present in cache.
* Byte offset is not used to check Hit/miss in any of the mappings.
* Meta-data size (or Tag-directory size) (all mappings) = no of blocks in cache * (Tag-bits + extra bits)
* For a given cache size, block size and MM size = Tag is same (for byte addressable and word addressable memory both)

2> Set Associative Mapping: [K-way set associative] → K-way means 1 set contains K blocks

* CM set no. = (mm block no) % (no of sets in cache)

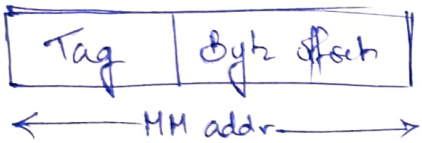
*



* Tag = mm addr − [log2(cache size) − log2(K)]

* No of indexes or no of sets = (no of blocks in cache) / (set-associativity)

3> Fully-Associative Mapping:



* Index in fully associative mapping = 0 bits

* Tag in fully associative mapping = mm block number

* For a given cache size, mm size & block size:

1> Size of tag & (tag-directory size)

Full assoc. > Set assoc. > Direct

2> No of bits in index

Direct > Set assoc. > Fully assoc.

Index	
Mapping	Index
Direct	CM block No.
Set-Asso	CM set No.
Fully	0 bit

* Hardware Implementation:

* Direct Mapping hardware:

1> Number of MUX for tag selection = Tag. bits

2> Size of MUX for tag selection = Number of blocks : 1

3> Number of Comparators = 1

4> Size of comparator = Tag. bits

* K-way set associative mapping :

1) Number of MUX for tag selection = K * tag bits
2) size of MUX for tag selection = Number of sets : 1
3) Number of comparators = K
4) size of comparator = Tag-bits.
5) OR-gate = 1 (K-input OR gate)

* Fully Associative mapping :

1) Number of comparators = Number of blocks in cache
2) size of comparator = Tag-bits
3) OR-gate = 1 (number of blocks-input OR gate)

* Hit latency time :

1) Direct Mapping = MUX delay + Comparator delay
2) Set associative Mapping = MUX delay + comparator delay + OR-gate delay
3) Fully associative Mapping = Comparator delay + OR-gate delay

Types of Cache Misses:

1) Cold or Compulsory miss: First time access of a block will always cause a miss.

To reduce Cold misses: Increase block size.

* Total cold miss = number of blocks in mm

2) Capacity Miss: If cache is full and hence miss occurs.
↳ and it is not the first time access of the block.

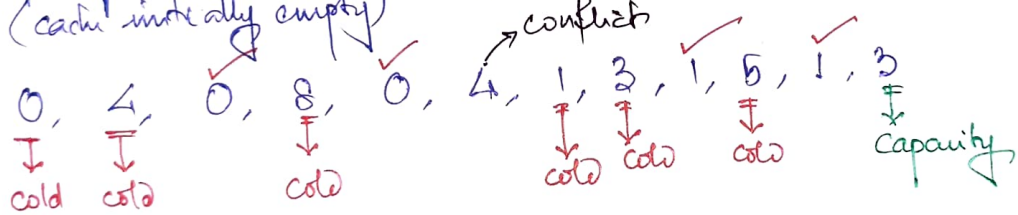
To reduce Capacity misses: Increase the capacity

3) Conflict Miss: If cache set is full and hence miss occurs due to tag mismatch.

To reduce conflict miss: Increase associativity

* No conflict miss in fully associative cache.

Ex: No of blocks in cache = 4
2-way set associative cache
LRU replacement policy
CPU requests for main memory blocks: 0, 4, 8, 0, 4, 1, 3, 1, 5, 1, 3
(cache initially empty)



00	0	~~4~~ ~~8~~ 4
01	1	~~3~~ ~~5~~ 3

cold miss = 6
conflict miss = 1
capacity miss = 1
8

* Ways of increasing Cache Performance:

1. Minimize Access Time — Smaller Cache
2. Maximize Hit Rate — larger Cache → {Multilevel Cache}
3. Minimize Miss penalty — Non-blocking cache, multi bank memory

Hit ratio = 4/12 = 0.33
Miss ratio = 8/12 = 0.67

Cache Block Replacement Policy :

* In direct mapping → no any replacement policy required.
* In set-associative mapping → replacement policy req.
  In fully-associative mapping → "

⇒ Block Replacement Algo : * replacement valid when : $N \geq 2$

1. FIFO :
2. optimal : Replace block which is not going to referred soon in future
3. LRU : Replace block which has not been referred since long.
   (Least Recently used)

Cache Miss Penalty : Time required to bring a missed block from main memory to cache

Assume, includes ⇒
- Cycles required to send addr the mem → 1 cycle
- Cycles required to access 1 mm cell → 10 cycles
- Cycles required to transfer 1 cell data to cache → 1 cycle

Cache Block Size	Main mem cell size	Miss penalty
4B	1B	1 + 10*4 + 1*4 ⇒ 45 cycles ⇒ 225ns
4B	2B	1 + 10*2 + 2*1 ⇒ 23 cycles ⇒ 115ns
4B	4B	1 + 10 + 1 ⇒ 12 cycles ⇒ 60ns

Assume, CPU operates on 200 MHz, then miss penalty time ?

$$\text{Cycle time} = \frac{1}{200\ MHz} = 5\ ns$$

* consider miss penalty cycle → 45 cycles ⟹ 225 ns

  In 225 ns, transfer = 4B

  1 ns → $\frac{4B}{225}$

  1 sec → $\frac{4B}{225 \times 10^{-9}\ sec} = \frac{4 \times 10^3 \times 10^6 B}{225} = 17.77\ MB/sec$

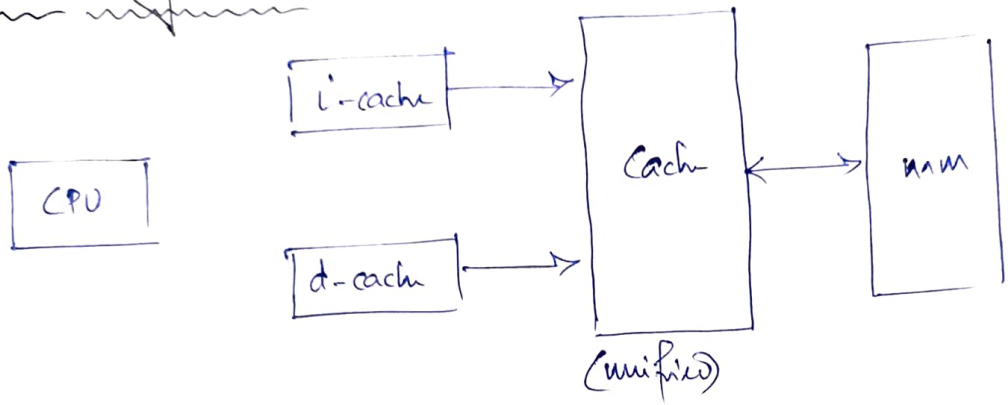
* Miss penalty increases when the block size increases.

- Multilevel Cache :

Simultaneous → $H_1 \times t_1 + (1 - H_1)\left[H_2 \times t_2 + (1 - H_2)\, t_{mm}\right]$

Hierarchical → $H_1 \times t_1 + (1 - H_1)\left(H_2 \times (t_1 + t_2) + (1 - H_2)(t_1 + t_2 + t_{mm})\right)$

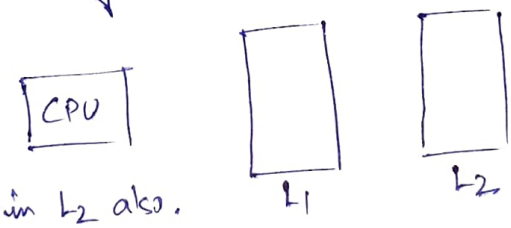
* Dual Cache Organization :



T_{avg} = ?

1) T_{avg} for inst (Assume d-cache is not present at all)
2) T_{avg} for data (Assume i-cache is not present at all)
3) T_{avg} = fract^{n} of inst^{n} × (T_{avg} inst) + fract^{n} of data × (T_{avg} data)

* Cache Inclusion Policy



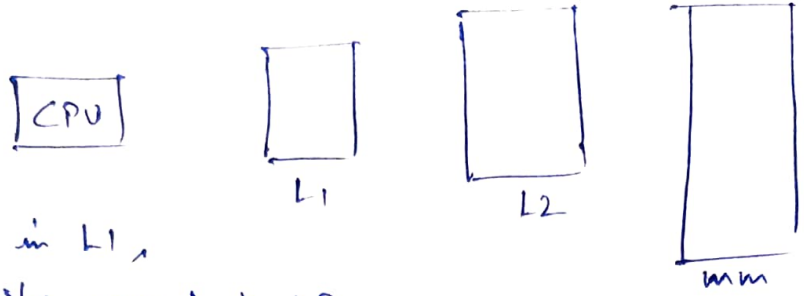
- All content of L_1 is present in L_2 also.
- It is not necessary to have same content in both the block present at level 1 and level 2 at any time
- We will use write back in most of the time, not need to update the block content every time the value changes in L_1 cache.

Read Operation Only

1. Hit in L1 : CPU gets direct content from L_1
2. Miss in L1 & Hit in L2 : Bring a block from L_2 to L_1, if a block evicted from L_1 then no any role of L2.
3. Miss in L1 & Miss in L2 : Bring the block into L2 and then L1

* If any block evicted from L2 then sent a block-invalidation to L_1 (make the valid bit in L1 of that block as 0)

* Exclusion:



Whatever present in L1,
not necessarily present in L2.

Read:

1) Hit in L1: CPU takes the content

2) Miss in L1, Hit in L2: Move a block from L2 to L1 and if the block is evicted from L1 then place it into L2.

3) Miss in L1, Miss in L2: Bring a block from mm to L1 and if the block is evicted from L1 then place it into L2.

* L2 is known as victim cache [All the replaced cache of L1 is placed here]

Pipeline

- **Parallel Processing:** Simultaneous data processing

  Types: Vector processing → Modular memory
  Array processing
  ** Pipeline Processing (Pipelining)

* Inst fetch related to control unit, and Inst execution related to ALU.

- **Flynn's Classification of Computers**
  1) SISD
  * 2) SIMD – pipeline processor
  3) MISD – Only theoritical
  4) MIMD – Multiple pipelines (Superscalar Computers)

{ S = Single; M = Multiple
I = Inst stream; D = Data stream

- **Pipeline Processing:**
  → Useful when some processing is applied over multiple i/p.
  → Tech. to decompose a seq. ops into sub ops.
  → Task: One Operation performed in all segment
  → Sub operations are performed in segments
  → Each segment can perform it's sub operation over different inputs

- **Pipeline Cycle time:**
  → That min time in which all segments can perform their respective sub-operation.
  → Time required to perform some operations in pipeline.
  ⇒ No of cycles required * cycle time
  → Performance in pipeline is given by speedup ratio.

$$\text{Speedup} = \frac{\text{non-pipeline time}}{\text{pipeline time}} * = \frac{n * t_n}{(K+n-1) * t_p}$$

[n value given in ques]

{ t_p = pipeline cycle time
t_n = Non-pipeline cycle time
K = No of stages in pipeline = # segment
n = No of inst^n

* When no of tasks increased i.e $n >> K$ [ignore K-1]

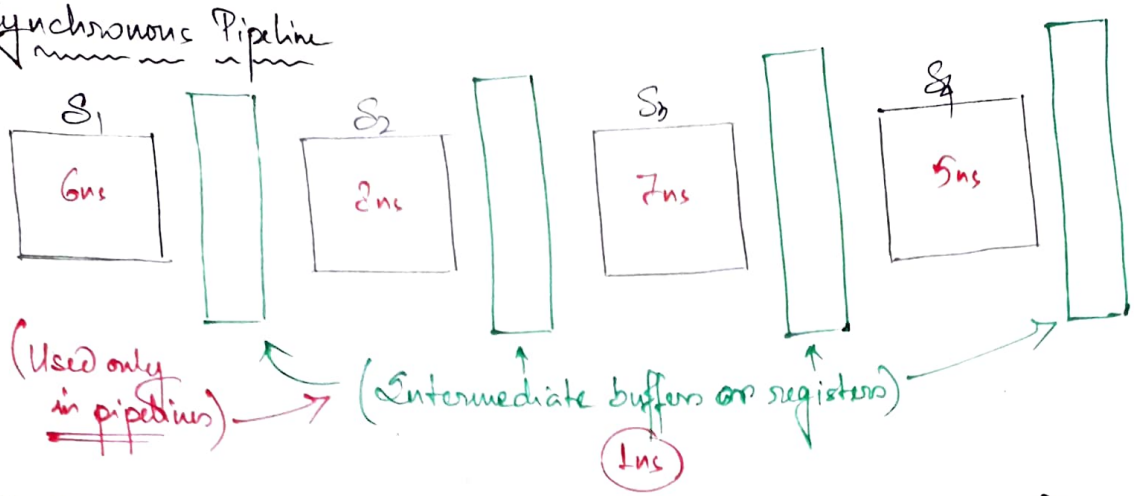
$$\text{Speedup}_{ideal} = \frac{t_n}{t_p}$$

{ max speedup, 1 cycle for each input }
[If n is not given in ques]

* Special Case: (Not always true)
⇒ Assume one task takes equal time in non-pipeline & pipeline system.
⇒ $t_n = K * t_p$
⇒ $S_{ideal} = K$ ← only in special case.

** $S \leq K$ ← Always.

- Synchronous Pipeline



* pipeline cycle time (t_p) = max (all segment delay) + Intermediate buffers or reg delay
  = max (6, 8, 7, 5) + 1ns = 8 + 1 = 9ns
* one task time in non-pipeline (t_n) = (Sum of all Reg delays) = 6 + 8 + 7 + 5 = 26ns

Latency (After how much time, we get the next o/p) new input is provided to system
- Non-pipelined: t_n
- pipelined: t_p

Throughput (No of tasks per unit time)
- Non-ideal case: $\frac{n}{(K+n-1)\,t_p}$
- Ideal case: $\frac{1}{t_p}$

* Try to divide operation into segments in such a way that each segment should take equal delay : to have speed up near maximum.

Observations :- In general case, no of cycles = $(K+n-1)$

no of cycles per task = $\frac{(K+n-1)}{n}$ = $CPI_{pipeline}$

Ideal case,

no of cycles = n

no of cycles per task = 1

Ignoring starting $(K-1)$ cycles to fill the pipe → $CPI_{ideal} = 1$ → In every single cycle, one o/p can be obtained from pipeline

$\therefore$ Throughput increase = $\left[\frac{\text{New throughput} - \text{old throughput}}{\text{old throughput}}\right] \times 100\%$

$\therefore$ $MIPS = \frac{\text{clock rate}}{CPI_{ideal} \times 10^6}$

Instruction Pipeline :

- pipeline processing applied on the instruction cycle → (fetch cycle) + (6 execution cycles)

Assume 5 seg. inst^n pipeline

- IF - Instruction fetch
- ID - Instruction Decode & Add. Cal^n
- OF - Operand fetch
- EX - Execution
- WB - Write Back

* Branch Detection : ID (Decod)
* Branch Evaluation : Ex (4^{th})

(4^{th}) → 3 stall cycles
(n^{th}) → (n-1) stall cycles

	1	2	3	4	5	6	7	8	9	10	11	12	13
I_1	IF	ID	OF	EX	WB								
I_2		IF	ID	OF	EX	WB							
I_3			IF	ID	OF	EX	WB						
I_4			~~IF~~	IF	ID (BRA)	OF	EX	WB					
I_5					IF_x	–	–						
I_6													
I_7													
I_8								IF	ID	OF	EX	WB	
I_9									IF	ID	OF	EX	WB

Branch Conditions = → Target

Stall Cycles

* Smooth pipeline = (K+n-1) = 5+6-1 = 10 cyc
* Extra stall = 3 cy.

Total = 13 cyc. //

* Key points :-

* Stall cycles ≠ Extra cycles needed in pipeline execution due to some hazard.

* If branch condition is evaluated after i^{th} stage of branch inst^n then the number of stalls because of branch inst^n = (i-1).

* In any case, whether the condition (Branch) is true or false, the branch inst^n will incur the stall cycles.

Pipelining

Q) Consider a program which contains 500 inst^n $I_1, I_2, I_3, \ldots I_{500}$. Further consider a 5-stage pipeline with stages as: Inst^n fetch, Decode, Operand fetch, Execution and Write back. The program contains 3 branch instructions, information of those given in a table below. The number of cycles required to execute this program in the given pipeline is ____ ?

Brach inst^n	Target Inst^n	Branch taken or not
I_9	I_{29}	Taken
I_{234}	I_{322}	Not Taken
I_{443}	I_{491}	Taken

In above ques Consider pipeline stage delays are 12ns, 15 ns, 13ns 17 ns and 14ns then,

(i) Total time req to execute the program?

(ii) Calculate avg. CPI ?

(iii) Calculate MIPS count ?

Sol^n:

No of inst^n $= (I_1 - I_9) + (I_{29} - I_{443}) + (I_{491} - I_{500})$

$= 9 + 415 + 10$ ($\leftarrow$ (443 − 29 + 1))

$= 434$

w/o hazards no of cycles $= (n + k - 1) = (5 + 434 - 1) = 438$ cycles

No of stalls $= (2) * 3 = 9$

(Taken or not taken both should be count.)

Total no of cycles $= 438 + 9$

$= 447$ cycles

(i) $447 * t_p = 447 * \max(12, 15, 13, 17, 14) = 447 \times 17 = 7599$ ns

(ii) 434 inst^n $\longrightarrow$ 447 cycles

1 inst^n $\longrightarrow$ $\frac{447}{434} = 1.029$ (Average CPI)

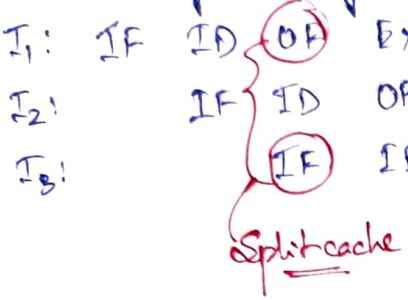
(iii) $MIPS = \frac{\text{clock rate}}{CPI_{avg} \times 10^6} = 57.16$ MIPS

• Pipeline Hazards :- Situations that prevent the next inst^n from being executions during its designated clock cycle.

1. Structural Hazard / Resource conflicts
2. Data Hazard / Data Dependency.
3. Control Hazard / Branch Difficulty.

* Structural Hazard / Resource conflicts :-

→ 2 different segments try to use same resource at same time.

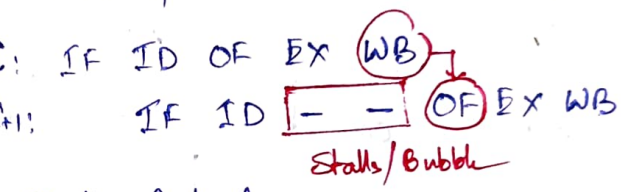


Sol^n: Use 2 cache (i) i-cache (ii) d-cache

* Data Dependency or Data Hazards :- Result of an inst^n is used as input in next.

i: $R_1 \leftarrow R_2 + R_3$

i+1: $R_5 \leftarrow R_1 * R_4$



A General pipeline h/w cannot understand or detect the Data dep.
↳ [Wrong result will be generated]

Sol^n of Data dependency

- s/w sol^n by compiler
  - Delayed load
    - Rearrange i/ps (insert independent inst^n)
    - insert no-operation inst^n b/w dependent inst^n
- h/w sol^n
  1. h/w Interlocking → providing stall cycles
  2. *** Operand forwarding or bypassing

i: $R_1 \leftarrow R_2 + R_3$

NOP } Added by compiler
NOP }

i+1: $R_5 \leftarrow R_1 * R_4$

(*) Compiler can also insert some other independent operation in b/w those inst^n.

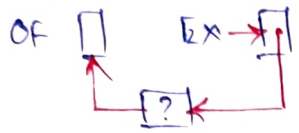
* Operand forwarding or Bypassing :-

Here pipeline is intelligent enough to detect dependency

i: $R_1 \leftarrow R_2 + R_3$: IF [] ID [] OF [] EX [] WB []

ii: $R_5 \leftarrow R_1 * R_4$: IF [] ID [] OF [] EX [] WB []

(*) No stall cycle [But costly sol^n]



decision to forward operand or not

⇒ Some h/w which detects

NOTE:

* For ALU to ALU data dependency, no stall cycle required of operand forwarding is used.

I_1: $R_1 \leftarrow M[add]$
I_2: $R_4 \leftarrow R_1 * R_5$

(or)

I_1: $R_1 \leftarrow R_2 * R_3$
I_2: $M[add] \leftarrow R_1$

⇐ Stalls are required.

* Control Hazards or Branch Difficulty :- Hazards bcz of branch inst^n

A general pipeline cannot have any solution for branch hazards

- s/w by compiler
  - Delayed branch
- h/w sol^n
  - → Branch prediction
  - → Providing stalls
  - → Prefetch target inst^n

⇒ Delayed Branch :-

Hardware cannot detect branch, so compiler has to provide sol^n [By adding NOPs]

i: Branch IF ID OF EX WB → 4th stage (3 stalls)

NOP
NOP
NOP
} hence, 3 stall / NOPs

i+1: Inst^n → IF ID OF EX WB

⇒ Branch Prediction :-

Pipeline can detect branch & predict the ~~stall~~ result.

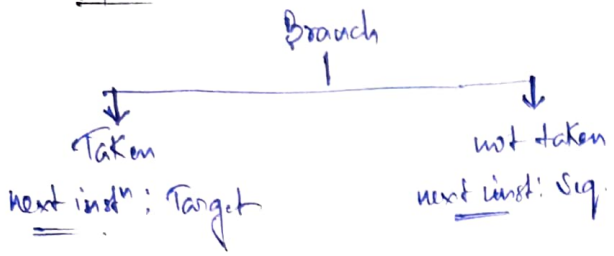
if prediction not correct, then roll back [CPU state needed to be saved]

→ costly h/w

(i) Static Prediction
(ii) Dynamic Prediction

Loop Buffer : Special Buffer which stores the small loop instⁿ and that is executed a no of times (as per req.)

⇒ Prefetch :



→ 2 pipelines (separate)
→ one pipeline is discovered when condition is known.
→ very costly (more h/w)
→ nested branch makes system even more complex.

* Hazard Classification (Data) :

Assume that, there are 2 instructions i & j. and i executes before j

(i) RAW (True dependency)
j tries to read a source before i with it → inconsistent
Sol" = delayed branch, operand forwarding

i: $R_1 \leftarrow R_2 + R_3$
i+1: $R_5 \leftarrow R_1 * R_4$

(ii) WAW : (write dependency) / false dep.
j tries to write a destination before i with it (final result differs)
Sol" = Register Renaming (static single assignment) H/w sol"

i: $R_1 \leftarrow R_1 + R_3$
i+1: $R_1 \leftarrow R_4 * R_5$

* (Race condition)

(iii) WAR : (Anti dep / false dep)
j writes a destination before i reads (i gets incorrect value)
Sol" = register Renaming.

i: $R_1 \leftarrow R_2 + R_3$
j: $R_2 \leftarrow R_4 * R_5$

NOTE:
(*) In case if we are not using operand forwarding, we are going to have the stall cycles due to RAW dep / true dep not due to WAW or WAR dep / false dep.
→ If u r not using op. forwarding try to draw diagram.

Q) Consider a pipelined processor with the following four stages:
IF: Instruction fetch; ID: Instruction decode & Operand fetch; Ex: Execution and WB: Write Back.

The IF, ID, and WB stages take one clock cycle each to complete the operation. The number of clock cycles for Ex stage depends on the inst^n. The ADD and SUB inst^n need 1 clock cycle and the MUL inst^n needs 3 clock cycle in the Ex stage. Operand forwarding is used in the pipelined processor. What is the number of clock cycles taken to complete the following sequence of inst^n?

ADD	R2, R1, R0	R2 ← R1 + R0
MUL	R4, R3, R2	R4 ← R3 * R2
SUB	R6, R5, R4	R6 ← R5 - R4

Try to work method 2 (w/o diagram) else draw diagram

In general, each stage takes one cycle for each inst^n

Given: IF – 1, ID – 1

	ADD	SUB	MUL
EX →	1	1	3
WB – 1	Extra – 0	0	2

In general, w/o hazard = K + n - 1 = 4 + 3 - 1 = 6
stall = 2
= 8 cycles

NOTE :-
All inst^n are ALU inst^n, hence no stall cycles due to data dependency, because operand forw. is used.

Q) GATE 2004

	S_1	S_2	S_3	S_4
I_1	2	1	1	1
I_2	1	3	2	2
I_3	2	1	1	3
I_4	1	2	2	2

for(i=1 to 2) {I1; I2; I3; I4;}

Algo. use LCS

	S_1	S_2	S_3	S_4
I_1	2	3	4	5
I_2	3	6	8	10
I_3	5	7	9	13
I_4	6	9	11	15
I_1	8	10	12	16
I_2	9	13	15	18
I_3	11	14	16	21
I_4	12	16	18	23

max + ( ) using given table

like LCS problem

Effects of Hazards: (CPI calculation in pipeline ← w/ hazard / w/o hazard)

⇒ w/o any hazard:

In general, $CPI = \frac{(k+n-1)}{n}$

In ideal, $CPI_{ideal} = 1$ $(\because n >> k)$

so ignore $(k-1)$

⇒ w/ hazard:

assuming 'x' no of cycles are needed extra because of hazard

In general, $CPI = \frac{k+n-1+x}{n}$

In ideal, $CPI = \frac{n+x}{n} = 1 + \frac{x}{n}$ → IDEAL

Ex.

Assume, $n = 100$ inst^n, $k = 4$

15 inst^n causing 3 stall cycles each.

∴ Total stall cycles $(x) = 15 * 3 = 45$ cycles.

$$CPI = \frac{k+n-1+x}{n} = \frac{4+100-1+45}{100} = \frac{148}{100} = 1.48$$

$$CPI_{ideal} = \frac{n+x}{n} = \frac{100+45}{100} = 1.45 = 1 + \frac{45}{100}$$

$$= 1 + \frac{15 * 3}{100}$$

$$= 1 + 0.15 * 3$$

$$CPI_{ideal} = 1 + \text{Stall freq.} * \text{Stall cycles}$$

(Stall freq. = Stall inst^n)

$CPI = 0.85 * 1 + 0.15 (1+3)$

$= 1.45$

$(1-f) * 1 + f(1+c)$

$(1-f) + f + fc$

$1 + fc$

* Efficiency and throughput of pipeline:

In ideal cond^n, max speed up = S_{max}

Achieved speed up = S

∴ $\text{efficiency} = \frac{S}{S_{max}} = \eta$

$S = \frac{t_n}{t_p}$ (w/o hazard)

$S = \frac{t_n}{CPI * t_p}$ (w/ hazard)

Throughput → work done / unit time

No of operation (ideal) = 1 in 1 cycle

Time (1 operation) = 1 cycle time (t_p)

∴ $t_p \rightarrow 1$

1 unit → $1/t_p$ = Thr.

Q) An instruction pipeline has five stages where each stage takes 3 ns and inst" are all five stages. Branch inst" are not overlapped, i.e. the instruction after the branch is not fetched till the branch instruction is completed. Under ideal conditions

a) Calculate the avg. inst" execution time assuming that 10% of all inst" executed are branch inst". Ignore the fact that some branch inst" may be conditional.

$K = 5$

$t_p = 3ns$

{ Stall bcz of branch inst" = 4 } $(k-1)$

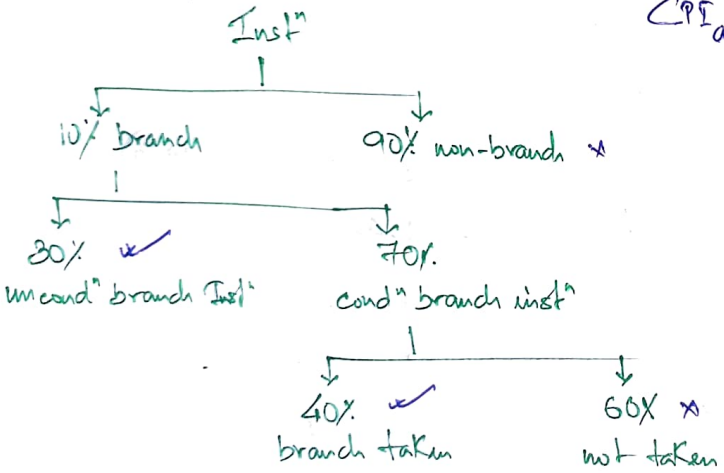
$CPI_{ideal} = 1 + p_c = 1 + 0.1 * 4$

$= 1.4$

extra { $S = \frac{15}{1.4 \times 3} = 3.57$

avg. inst" time $= 1.4 * 8ns = 4.2ns$ (Ans)

b) If a branch inst" is a conditional inst", the branch need not be taken when the condition is false. If the branch is not taken, the following instructions can be overlapped. When 70% of all branch inst" are conditional branch inst", and 40% of the conditional branch are such that the branch is taken, calculate the average inst" execution time?



$CPI_{avg} = 0.9 \times 1 + 0.1 \times 0.3 \times (1+4)$

$+ 0.1 \times 0.7 \times 0.4 \times (1+4)$

$+ 0.1 \times 0.7 \times 0.6 \times 1$

$= 1.232$

avg inst" exe time

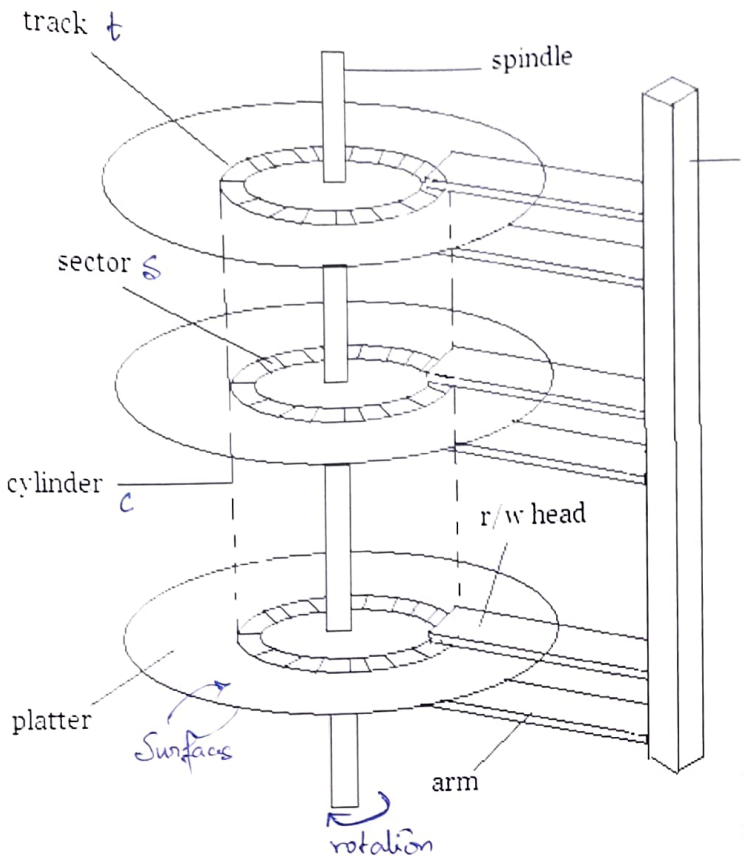
$= 1.232 * 3ns$

$= 3.696ns$ (Ans)

extra { $S = \frac{t_n}{CPI \times t_p} = \frac{15}{1.232 \times 3}$

$= 4.058$

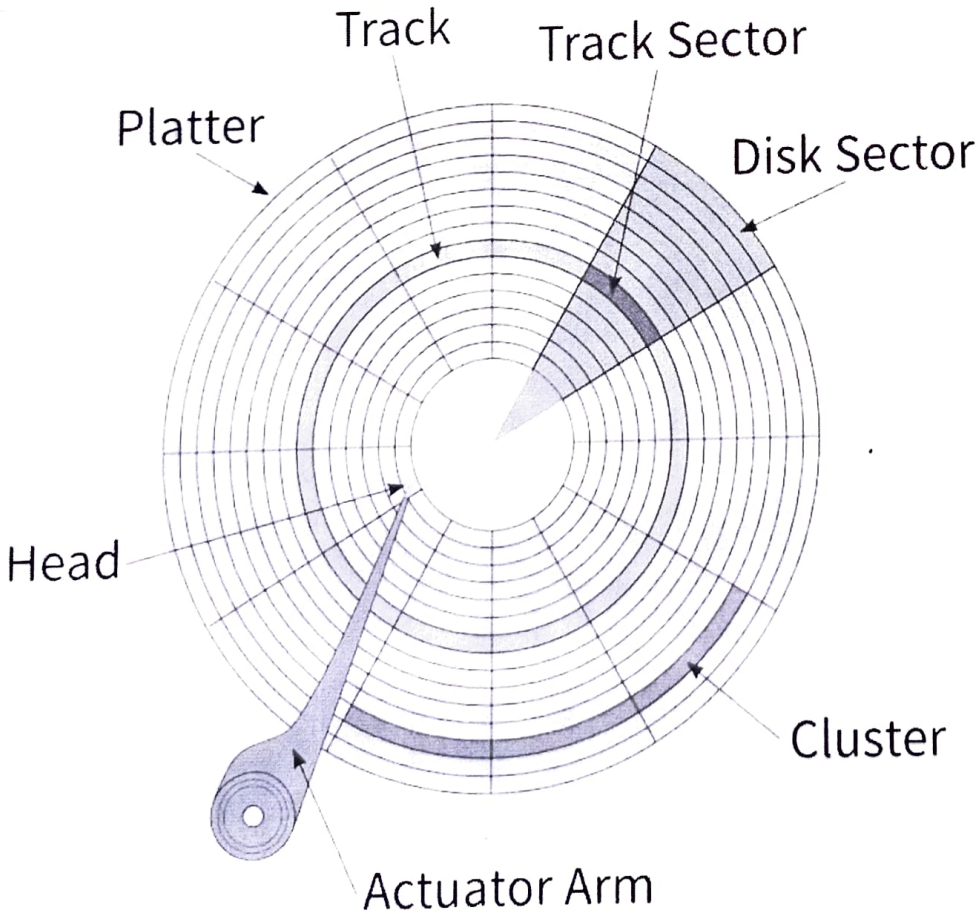
Magnetic Disk



* Addressable unit of disk → Sectors.

* No of Sectors in Disk =
2 * no of platters * No of tracks per surface * No of sectors per track

Top view of the Disk:



* Sector :-
Sector is the smallest unit of disk which can be read/written at once.

Sector Capacity

↓ (Default disk)

Sectors having constant capacity

(or)

variable storage density

* Constant Angular velocity

Not in syllabus

Sectors having variable capacity

or

Constant storage density

* constant linear velocity

* Disk Capacity:
  1) Surface capacity: Sum of capacity of
  2) Disk capacity: Surface cap. * no of surfaces

* Disk Capacity = 2 * No of platters * no of tracks per surface * No of sectors per track * one sector capacity

→ access time for 1 sector of disk

* 1 disk access time = Seek time + Avg rotational delay or latency + 1 sector transfer time + Extra delay.

* Seek time = Time required to position the arm over the desired track.
* Rotational latency = Time required to rotate desired sector under R/W head.
* Transfer time = Time required to read or write 1 sector.

Rotational Delay

If current and target sector given.

⇒ $\frac{1 \text{ Disk rotation time}}{\text{No of sector per track}}$ * No of Sectors to rotate.

If current & target sector is not given.

⇒ Avg. rotational latency

⇒ $\frac{1 \text{ disk Rotation time}}{2}$

Note: In one disk rotation, one track can be transferred.

Hence,

$$1 \text{ Sector transfer time} = \frac{1 \text{ Disk rotation time}}{\text{No of sectors per track}}$$

Multiple Sector Access time

All sectors stored on same track, consecutively or Sequentially.

Consider n sectors to be transferred;

= Seek time + Rotational latency + n * (1 sector transfer time)

All sectors are stored Randomly.

= n * [Seek time + Rotational latency + 1 sector transfer time]

* Cylinder:= Collection of tracks of same radius from all the surface from a cylinder.

⇒ Data stored and accessed cylinder-wise : To save seek time.

* No of cylinders in disk = No of tracks per surface

* Disk Addressing:=

- 3 platters,
- 2 surfaces per platter
- 4 tracks per surface,
- 10 sectors per track.

↳ 60 sectors in a cylinder

Address ⟶ <c, h, s> → sector no.
(c: cylinder no., h: surface no.)

n_c = no. of surfaces per cylinder

n_t = no of sectors per track

$$\text{Sector seq. no} = c * n_c + h * n_t + s.$$

$$c = \text{Sector seq. no} / n_c.$$

$$h = (\text{Sector seq. no} \,\%\, n_c) / n_t$$

$$s = (\text{Sector seq. no} \,\%\, n_c) \,\%\, n_t$$

Q> A hard disk has 63 sectors per track, 10 platters each with 2 (20 surfaces) recording surfaces and 1000 cylinders. The address of a sector is given as a triple $\langle c, h, s \rangle$ where c is cylinder number, h is surface number and s is sector number. Thus, the 0th sector is address as $\langle 0,0,0 \rangle$, the 1st sector as $\langle 0,0,1 \rangle$ and so on.

(a) The address $\langle 400, 16, 29 \rangle$ corresponding sector number is:

n_t = no of sectors per track = 63.

n_c = no of sectors per cylinder = 63 * 20 = 1260.

no of sectors covered in 400 cylinders (0 - 399) = 400 * 1260 = 504000

no of sectors covered in 16 surfaces (0 - 15) = 16 * 63 = 1008

+ 29

505037

same Qus and same data continued...

(b) The address of the 1039th sector is ⟶

$$c = \left\lfloor 1039 / 1260 \right\rfloor = 0$$

$$h = \left\lfloor (1039 \% 1260) / 63 \right\rfloor = 16$$

$$s = \left[(1039 \% 1260) \% 63 \right] = 31$$