INTRODUCTION

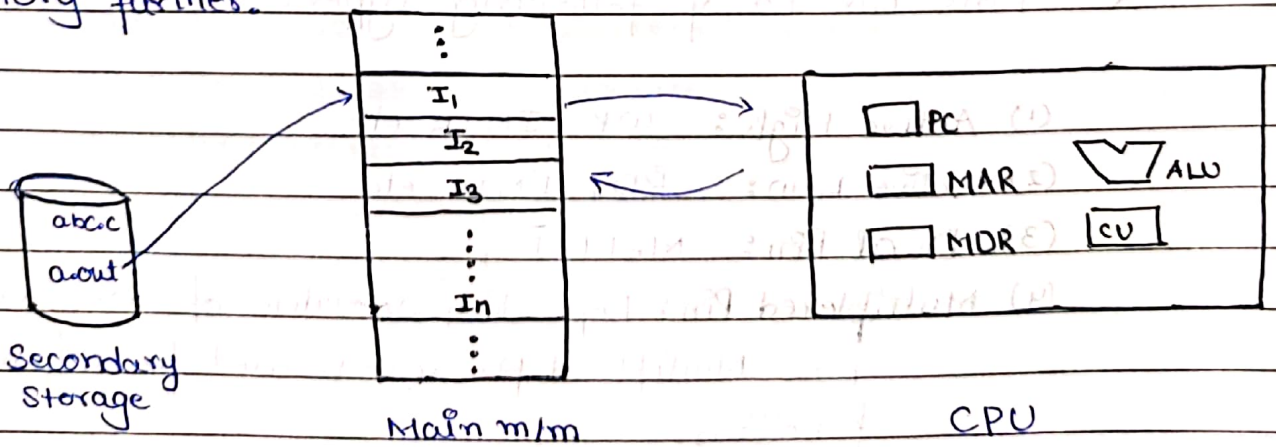
Computer Architecture	Computer Organization
⇓	⇓
• Programer's Perspective • Abstract View • Programer knows that there's CPU, Memory, I/O & ways to interact with it say, using instructions.	• Designer's perspective • Hardware View • Talks about, how to design low latency m/m, may use cache or not, etc, what instructions to provide, etc.

* Computer Organization

Story started as follows:

(1) C Program, written by programmer
(2) Compiler & linker converted to binary executable.
(3) Loader loaded the program into memory.

Now CPU has access to instructions. COA continues the story further.



* Program: Sequence of instructions + Data

* • Memory is divided into equal sized ~~cells called~~ partition called cells & each cell has unique address.

• Each cell allows only two operations: Read or Write.

• CPU generates Memory Request to access data or instruction from memory.

Memory Read: <Address>, $\overline{RD}$
Memory Write: <Address>, WR, <value>

• We see two types of signals here: $\overline{RD}$ & WR. $\overline{RD}$ is active low & WR is active high.

• Pins like these are how two components can communicate with each other.

* Pins can be of following types:

(1) Active High: WR, INTR, etc.
(2) Active Low: $\overline{RD}$, $\overline{INTR}$, etc.
(3) Dual Pin: MEM/$\overline{IO}$,
(4) Multiplexed Pin: Depending on value of some other pins Multiplexed pins can be used for different functions.

Ex: AD_0 .... AD_{15}

Can carry out 2 or more funcs.

* n-bits Processor : Tells that :
  (1) ~~Size~~ No. of Data lines : ~~n-bits~~ n
  (2) ~~No.~~ Databus size : n-bits
  (3) Word Size : n-bits
  (4) ALU : n-bits
  (5) Accumulator : n-bits
  (6) Memory Data Register : n-bits
  (7) General Purpose Registers : n-bits.

Basically anything to do with data by the processor is n-bits.

* Addressability of Memory :
  (1) Byte Address : Each cell is of 1 Byte. OR address is given to each byte.

  (2) Word Addressable : Each cell is of 1 word. OR address is given to each word.

Ex. M/M is 8 GB. No. of address bits for
  (1) Byte Addressable : 33 bits.
  (2) Word Addressable (w = 4B)

$$\therefore \#\text{cells} = \frac{8GB}{4B} = 2G$$

$$\therefore 31 \text{ bits.}$$

* Memory Chip Configuration

m x n

- m → #cells in the m/m
- n → size of each cell/word.

Ex. 64K x 16

∴ # cells = 64K

⇒ size of address bus = $\log_2 64K$

= 16

width of address bus = 16 bits

- If unit of cell size isn't mentioned, take as bits.

* Components of Computer System

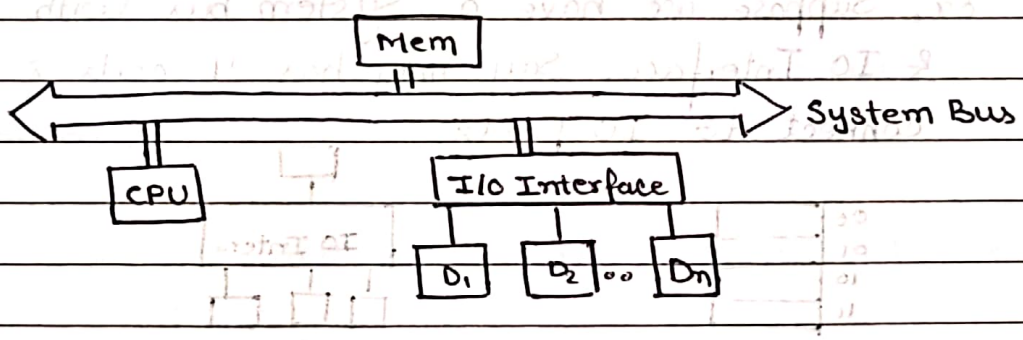
Whenever a program is run, it would use / require three components:

(1) CPU: Master component. Would execute the program.

(2) Memory: Slave component. Slow. Stores program & data

(3) I/O: Slave component. Slowest.

When we say fast / slow this is based on clock cycle time. Faster component has lower cycle time.



* . System Bus

Consists of three set of lines:

(1) Address Lines (AL): Used to carry address from CPU to m/m or I/O. Unidirectional.

Ex. 8085 has 16 bit Address Bus:

So, max addressable m/m = 2^{16} Cells

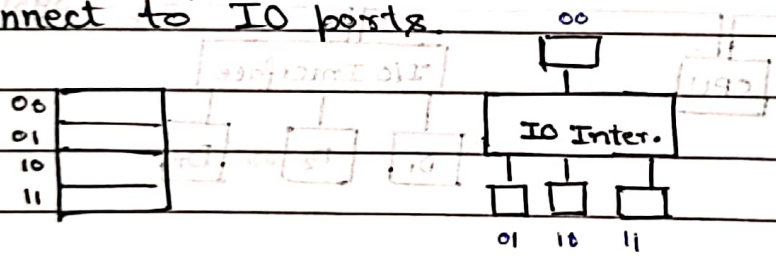
Ex. Mem is $2^{40} \times 8$

So, no. address lines = 40.

(2) Data Lines [DL]: Used to carry data from CPU to M/m, M/m to CPU, CPU to I/O & IO/CPU.

(3) Control Lines [CL]: Used to carry control signals. Control signals tell what type of operation is being performed.
Memory never issues control signal.

Ex. Suppose we have a system bus with CPU, M/m & IO Interface. Say m/m has 4 cells & 4 I/O devices connect to IO ports.



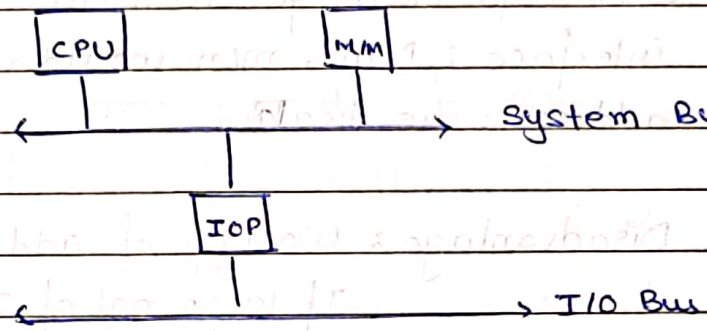
Now if CPU issues, say a Memory read Req:
01, $\overline{RD}$
This would go on system bus & hence both to M/m & I/O interface.
How is one to know if it is meant for IO read or M/M read ?

Since all three components share the same AL, DL & CL, this problem is happening.

* There are three solutions to the problem:

(1) Separate I/O Processor

It's a separate processor which only performs I/O operation. DMA is an example of IOP.



∴ A separate bus for IO. All IO operations go through IOP.

We can use same addresses & control signal, but different buses.

(2) Isolated I/O (IO Mapped IO)

We have separate read & write operation for IO & M/M.

IO: $\overline{IORD}$ & IOWR

M/M: $\overline{MRD}$ & MWR

So we use same system bus & same addresses but different control signals.

This is generally achived using an additional signal

M/$\overline{IO}$	$\overline{RD}$	WR	Operation	Assembly Inst Ex.
0	0	0	IO RD	IN R_1, 01
0	1	1	IO WR	OUT 11, R_1
1	0	0	MM RD	LOAD R_1, 01
1	1	1	MM WR	STORE 11, R_2

(3) Memory Mapped IO:

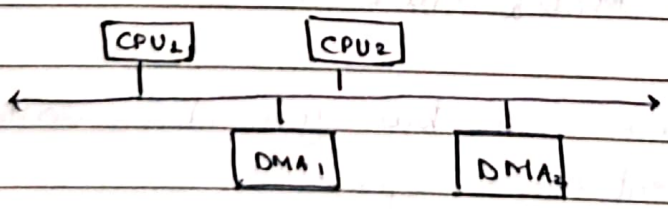
We keep the same system bus, use same signal & use some addresses specifically for IO device addressing, and disable those addresses in M/m.

When req. meant for IO device goes to both IO interface & M/m, m/m would reject it if the addresses are disabled

Disadvantage: Wastage of address space in m/m. If large no. of IO devices, then large m/m wastage.

Ex. M/M has 8 bit addresses,
then m/m size = 2^8 cells = 256 cells.
Say we decide that any address of form xyz1ab0c is reserved for IO port addressing
∴ We disabled 2^6 = 64 cells in m/m.

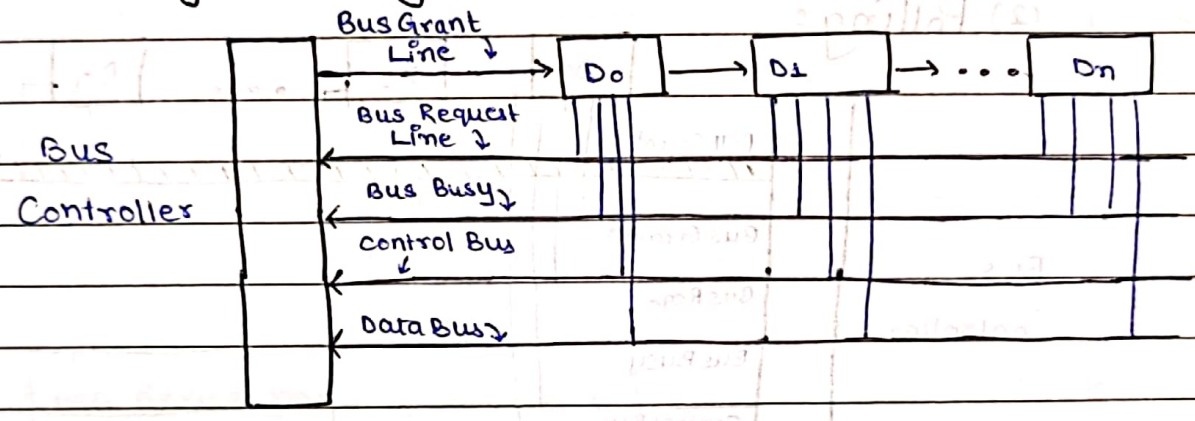
Ex. Say we have such a system bus



If multiple masters are trying to access System Bus for various operations at same time, then we need a method to choose the next master of system bus.

*. Bus Arbitration: The process of deciding on the next master of System Bus is called as Bus Arbitration.
There are three methods of bus arbitration

(1) Daisy Chaining:



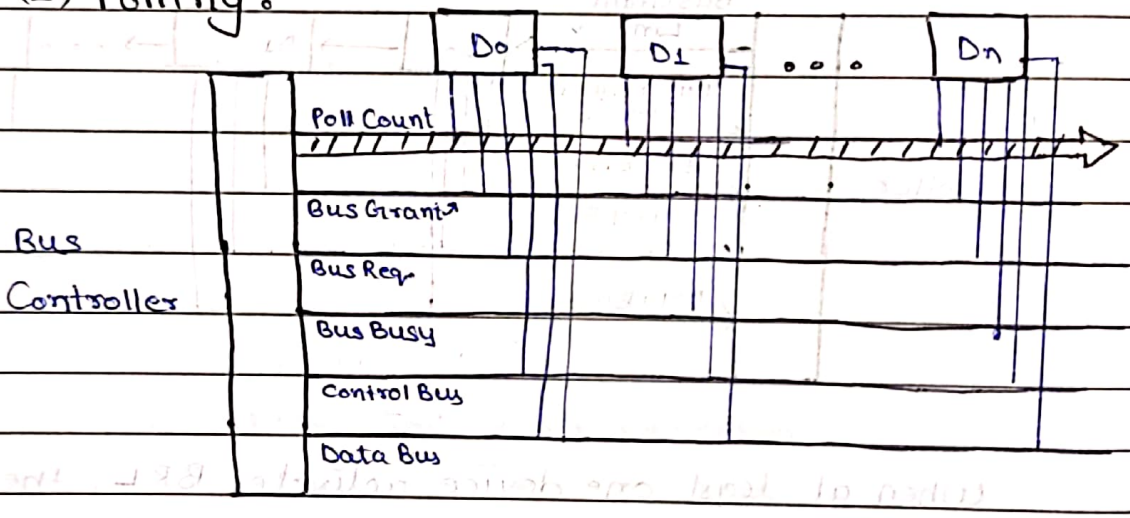
When at least one device activates BRL, the Bus Controller activates bus grant line of D_0. If D_0 made the request, it activates Bus Busy signal so that Bus controller & other devices know that it is accessing the Control & Data buses. Bus Controller grants access to D_0.
If D_0 didn't make request it just activates Bus Grant Line of D_1 & the process repeats.

Note: Devices closer to Bus Controller get higher priority.

Disadvantages:
(1) If one node fails whole system fails. Single point of failure
(2) Higher unnecessory delay for devices further from Bus Controller.
(3) Device priority is static & higher for closer devices to Bus Controller.

The task of Bus Controller also keeps shifting from one device to other. Sometime CPU_1 can act as BC, or sometime CPU_5, sometime DMA_{10}. The Control Unit of the current BC decides which Master device would become BC next.

(2) Polling:



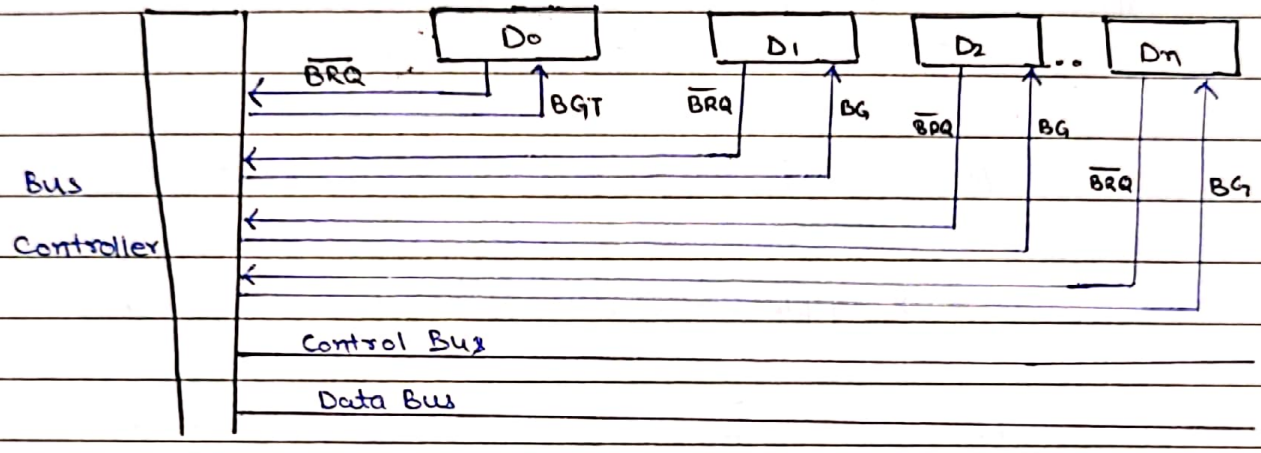
Poll Count is a K-lines signal. $K \geq \lceil \log n \rceil$ for n devices.

When one or more device activates Bus Request Line, the Bus controller generates Poll Count (or device index) in random order & queries if device generated the request. The devices are indexed based on dynamic priority which can change over time.

The device which did activate request line activate Bus Busy signal. Bus controller activates Bus Grant signal & grant access to Control & Data bus to that device.

So all devices have fair chance & no single point of failure.

(3) Independent Bus Req. & Grant:



Each device has separate Bus Request & Bus Grant lines.
Independently they request access to system buses.
BC has a static priority for the devices based on which it chooses the next device to grant access.
BC activates the selected devices Bus Grant Line & allows access to Control Bus & Data Bus.

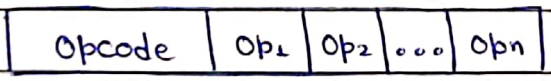
Instructions & Addressing Mode

* **Instructions :** Inside a machine, Instruction is nothing but sequence of bits which is processed by CPU.

Each Instruction is made up of some operation & operands.
Say we want to add values in register R_1 & R_3.

$R_1 + R_3$	≡	ADD R_1, R_3	≡	1010 0111
Human Readble form		Assembly Code		Corresponding Bits

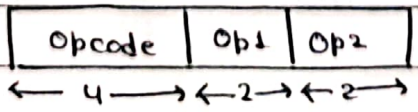
The mandatory operational code or opcode.
Then there are 0 or more operands.



Ex. Say opcode field in a particular CPU is 3 bits.
then total $2^3 = 8$ operations can be supported & we can encode whichever operation to whichever opcode. One possible assignment is

ADD : 000	LOAD : 010	MUL : 100	INCR : 110
SUB : 001	STORE : 011	DIV : 101	DECR : 111

Ex.



So, $2^4 = 16$ operations
$2^{4+2+2} = 256$ instructions possible
With each operation, $2^{2+2} = 16$ instr. are possible.

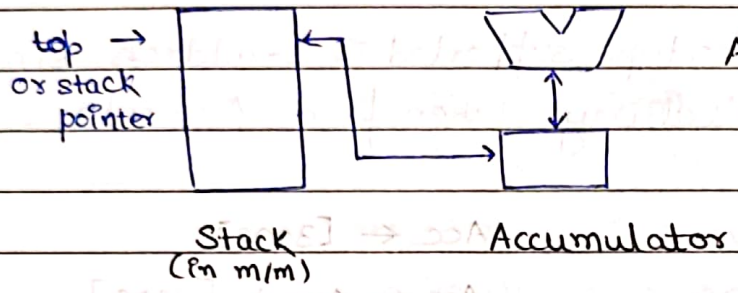
Ex. Instruction size = 20 bits M/M Size = 256 Bytes
2 addr instructions. # operations?
(Assuming 256 x 8 m/m)
Addr size = $\lceil \log_2 256 \rceil = 8$.
∴ # opcode bits = 20 - 8 - 8
$\Rightarrow 4$
∴ # opcodes = $2^4 = 16$

Ex. Inst Size = 32 bits #operations = 256. Only 3 addr inst. Size of m/m?

Rem. bits = $32 - \lceil \log_2 256 \rceil = 24$
per addr = 24/3 = 8
∴ m/m size = 2^8 = 256 B (or cells)

* Instruction Format (Based on m/c Type)

- Stack Organization:



At each cycle the top of stack is moved to accumulator, then operation is performed by ALU & moved to accum. Then pushed to stack top again.

(1) PUSH Acc → Stack Top
(2) POP Stack Top → Acc.
(3) Operations such as ADD, INCR, etc.

All Stack org. Inst^{n}s are 0-address.

Ex. POP — pops two elements from Top of Stack
POP — adds them & pushes result back to
ADD — stack.
PUSH

- Accumulator Organization:

Only one register is available: Accumulator.
One operand is available in Accum. always & the other

operand is in memory.
After the operation completes the result is stored in the accumulator.
This is also called LOAD & STORE machine.

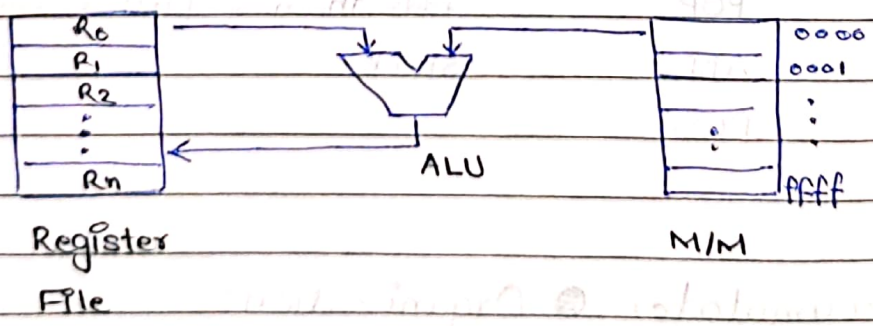
The ~~only~~ operation type supported is 1-address, since other operand is always taken from Accumulator.

Ex. LOAD 3000 Acc ← [3000]
ADD 2000 Acc ← Acc + [2000]
STORE 3002 3002 ← Acc

(Zero addr. operations are also possible).

• General Register Organization:

(1) Register to M/M organization
Small set of registers. One operand is from register and another from the memory. Result would be stored in the register.



This type of machine supports upto 2-addr. instruction.

Ex. LOAD	R_0, [2000]	$R_0 \leftarrow$ [2000]
STORE	[3000], R_1	3000 $\leftarrow R_1$
ADD	R_2, [2000]	$R_2 \leftarrow R_2 +$ [2000]

(2) Register to Register Orgⁿ

- Both operands are register locations and the result is also stored in register.
- Processor supports more no of registers.
- Since #registers is high, operands can be prefetched as well.
- It supports upto 3-address instructions.

Ex. LOAD	R_1, 3000	$R_1 \leftarrow$ [3000]
LOAD	R_2, 2000	$R_2 \leftarrow$ [2000]
ADD	R_3, R_1, R_2	$R_3 \leftarrow R_1 + R_2$
STORE	3002, R_3	3002 $\leftarrow R_3$

- Fastest of all orgⁿ since operands are in registers.

* Instruction Format (Based on number of operands)

- Zero Addr: | opcode |
- One Addr: | opcode | Op_1 |
- Two Addr: | opcode | Op_1 | Op_2 |
- Three Addr.: | opcode | Op_1 | Op_2 | Op_3 |

Operand can be m/m addr., register addr., or constants.

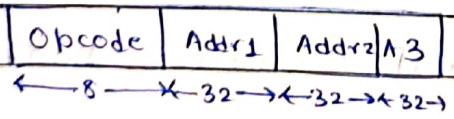
Ex. Instⁿ Size: 20 bits. 64 regs. MM = 256 B.
If one addr is reg. addr, other is m/m addr. No. of 2addropcodes supported?

m/m addr size = $\log_2 256 = 8$
reg. addr size = $\log_2 64 = 6$.

∴ Opcode bits = 20 - 8 - 6 = 6.
∴ # opcodes (2addr) = $2^6 = 64$.

* There are two types of machines:
(1) Same Length Instⁿ
The size of instruction is fixed.

Ex. Say there's a machine with upto 3-addr instructions, with Mem. addr of 32 bits, and 256 operations. then addr would be like



and size of Instⁿ is 104 bits.
Even if we have a 0 addr instⁿ out of the 256 operations, say CMA or INCRA etc. we'd still have to use 104 bits to store it in the memory.

Disadvantage: Memory wastage.

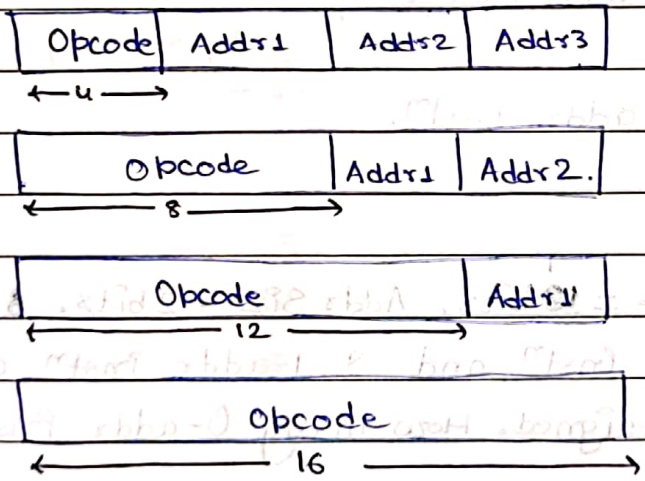
(2) Variable Length Instructions :

We may have some 1-addr, some 2-addr, some 0-addr instructions that we want our machine to perform. Using variable length inst^n would create tradeoff b/w opcode bits and address bits.

Expanding Opcode Technique

This basically means, we are making the opcode field length variable & not the inst^n length.

Ex: Say we have 16 bit instn & 4 bit addr, then we can vary opcode field size as per no. of addresses:



Ex. Possible Design.

0000	xxxx	yyyy	zzzz	15 3-add inst^n
0001	xxxx	yyyy	zzzz	
⋮				
1110	xxxx	yyyy	zzzz.	
1111	0000	xxxx	yyyy	13 2 addr inst^n
⋮				
1111	1100	xxxx	yyyy	
1111	1101	0000	xxxx	$(3\times16)-1 = 47$ 1 addr inst^n
⋮				
1111	1111	1110	xxxx	
1111	1111	1111	0000	10 0-addr inst^n.
⋮				
1111	1111	1111	1010	

The CPU would be designed to recognize the following:

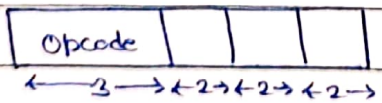
(1) If first 4 bit are in range (0000 to 1110), then interpret as 3 addr Instⁿ.

(2) If first 4bit in range (1111 to 1111) & the next 4 bit in range 0000 to 1100, then interpret as 2-addr instⁿ.

(3) If first 4 bit in range (1111 to 1111), next four bit in range (1101 to 1111), and next 4 bits in range then 1-addr instⁿ.

(4) Else 0-addr instⁿ.

Ex. Instⁿ Size = 9 bits. Addr Size = 2 bits. 2 3-addr instⁿ, 20 2-addr Instⁿ, and 8 1-addr Instⁿ are required to be designed. How many 0-addr Instⁿ are possible?



possible 3addr instⁿ = $2^3 = 8$

Req —11— = 2

Remaining = 6.

possible 2 addr instⁿ = $6 \times 2^2 = 24$

Req —11— = 20

Remaining = 4

possible 1 addr instⁿ = $4 \times 2^2 = 16$

Req. ⟶ ⟶ = 16 − 8 = 8.

possible 0 addr instⁿ = $8 \times 2^2 = 32$.

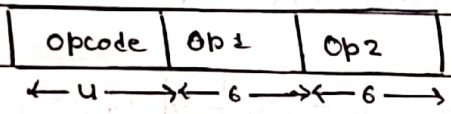
∴ 32

Ex. Is following config possible:

14 2-addr instⁿ

127 1-addr instⁿ

60 0-addr instⁿ



2 addr possible = $2^4 = 16$

Used = 14

Remain = 2

1 addr instn possible = $2 \times 2^6 = 128$

Used = 127

Remain = 1

0 addr possible = $1 \times 2^6 = 64$

Remain = 64 − 60

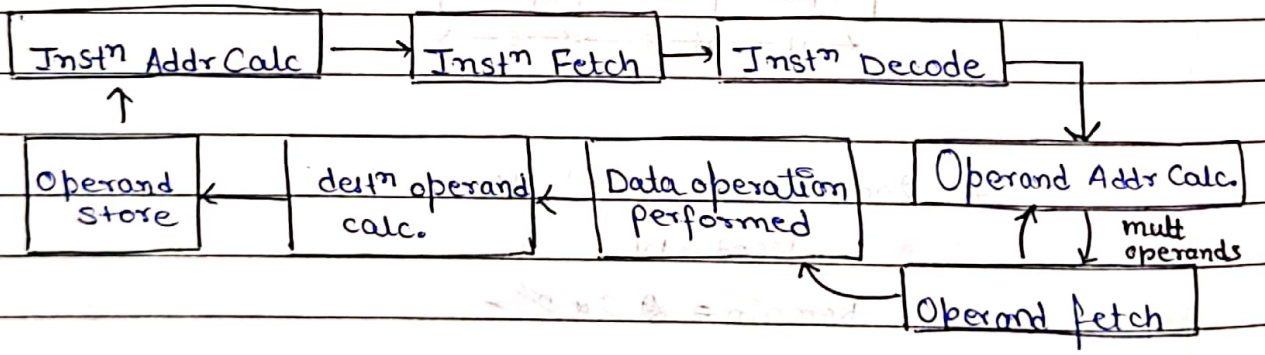
⇒ 4.

∴ Yes possible.

* Addressing Modes

Instruction Cycle has two phases : . Fetch Cycle
. Execution Cycle.

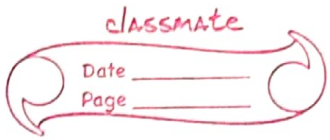
The first step during execution would be to calculate the address of inst^n, then we can go ahead & fetch inst^n from m/m.
Once inst^n is in Inst^n register, we can decode it.
After decoding the operation, we require operands.
This requires an iterative cycle of operand addr. calculation & operand fetch for each operand.



So the addressing mode dictates where the operand would be found.
An operand can be found in four places:
(1) Opcode : INCA (Incr. Acc), CLRC (Clear Carry), etc.
(2) Instruction : ADDA #0FFF (add accum.), etc.
(3) Registers : ADD R_1, R_2
(4) Memory : STOREA [3000]

- Any inst^n may use combination of multiple addr. modes.
- Some bits in the inst^n format are reserved for specifying addressing mode, per operand.

MOV: Used to move data b/w registers. It copies.
LOAD: Used to read from m/m.
STORE: Used to store data to m/m.



• Implied / Implicit Addressing Mode.

Operand is specified within the opcode. No effective address calculation. 0-addr instructions.
Ex. INCRA, CLRC, CLRA, etc.

• Immediate Addressing Mode:

Operand is available inside the instruction.
Sometimes specified using I at end of opcode.
Ex. ADD + I ⇒ ~~ADDI~~ ADI

Ex. ADI 1000 adds value 1000 to Accum. Value.

Sometimes immediate values are prefixed with #.
Ex. ~~ADDA~~ ADA #1000 adds value 1000 to Accum. value.
Ex. MVI R_1, 1000

• Register Addressing Mode.

Operand is available inside some register and the address of that register is specified in the Insⁿ.

Eff. Address = Reg. Name / Reg. Address

Ex. MOV R_1, R_2 $R_1 \leftarrow [R_2]$

• Direct (Absolute) Addressing Mode

Operand is stored in some memory location & the address of the operand is provided inside the Instⁿ.

Eff. Addr. = Mem. Addr in Instⁿ

Ex. ADD 2000, 3000 EA = 2000, EA = 3000

[2000] ← [2000] + [3000]

∴ 2 m/m access during operand fetch & 1 m/m access during write back.

• Indirect Addressing Mode

Used in case of pointers in High Level language. Operand is stored in some m/m loc. x & the address x is stored in another m/m location y. The Instⁿ contains the value x.

Eff. Addr = [x] = y.

Requires two memory accesses.

Ex. ADA @1000 & say [1000] = 5000.

then EA = [1000] = 5000

Acc ← [Acc] + [[1000]]

i.e. Acc ← [Acc] + [5000]

• Register Indirect Addressing Mode

The operand is stored in some m/m loc. x, the addr. x is stored in some register R_n & the inst^n contains the register name/address.

Eff. Address = $[R_n]$

Ex. LOAD R_1, @R_1 & $[R_1] = 1001$

EA = $[R_1] = 1000$

$R_1 \leftarrow [[R_1]]$

$R_1 \leftarrow [1001]$

One Register Access & one m/m access.

• Indexed Addressing Mode

Used for Array access. Can't use array construct without this AM.

The Base Address of the array is stored in a special register : Index Register. In the inst^n, the offset is provided.

EA = $[IR]$ + offset (offset → base reg. ↓)

Ex. LOAD 16(R_n)

EA = $[R_n] + 16$

Acc $\leftarrow [[R_n] + 16]$

• Program Counter Addressing Mode

The base addr is taken from Program Counter.

In the inst^n, the offset only is provided.

(aka. PC Relative)

(aka. Relative)

At the time the instⁿ is decoded, the Program Counter is already moved to the next instⁿ location.

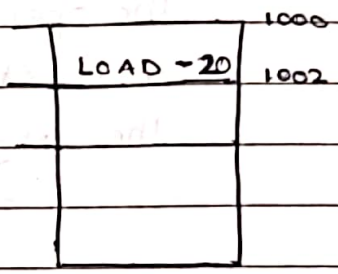
Ex. LOAD -20

would result in:

EA = [PC] - 20

= 1002 - 20

⇒ 982



Eff. Address = [Program Counter] + offset

- Base Index Addressing Mode

Eff. Address = [Base Reg.] + [Index Register]

Ex. LOAD [Bx][Ri]

EA = [Bx] + [Ri]

This requires two register access to calculate the effective address, and one m/m access to get operand.

- Base-Index w/ Displacement AM

Eff. Address = [Base Reg] + [Index Reg] + Displ.

Ex. LOAD 20[Bx][Ri]

EA = [Bx] + [Ri] + 20.

Also req. 2 register access & one m/m access.

- Register Auto Incr./Decr. AM

(1) POST

EA = [Reg. Name / Addr]

Reg ← [Reg] ± Inst Size.

(2) PRE

EA = [Reg] ± Inst. Size

Reg ← [Reg] ± Inst. Size.

Ex. LOAD +(R2) Inst size = 2B.

EA = [R2] + 2

R2 ← [R2] + 2.

Ex. STORE (R1)- Inst size = 4

EA = [R1]

R1 ← [R1] - 4.

So, one register access, one arithmatic operation & one m/m access.

- Base Addressing Mode.

EA = [BR] + offset

Imp This & PC relative modes are used to write Position Independent Code.

Position Ind. Code → think PC relative, then Base Addr. Mode

* Instruction Cycle

There are four cycles, two of which are optional:

(1) Fetch Cycle: The instruction is fetched from m/m based on PC value.

(2) Indirect Cycle [optional]: Required when using some form of indirection & the effective address has to be brought into the Instⁿ, such as in Indirect AM, Indexed AM, PC Relative AM, etc.

(3) Execution Cycle: Instruction is decoded, operands are fetched, operation is executed & result is written back.

(4) Interrupt Cycle [optional]: Executed when there's an interrupt to the CPU. There's an INTR line of CPU, which is activated by IO & m/m. After each instruction, CPU checks if this line is high, it runs this cycle.

(1) Acknowledges the interrupt. $\overline{\text{INTA}}$
(2) Saves state
(3) Jump to ISR.
(4) Run ISR
(5) Jump back & resume state.

• Fetch Cycle

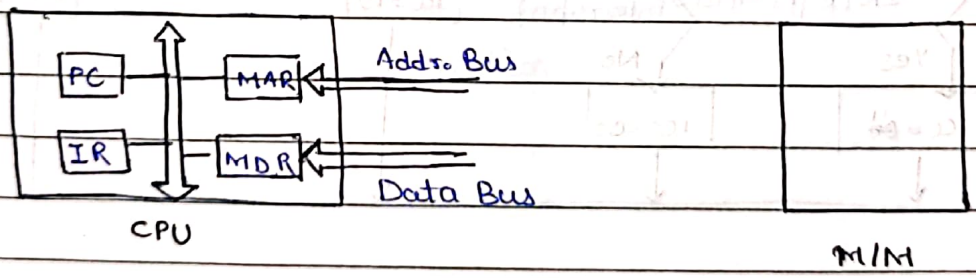
Instⁿ are stored in the m/m. CPU can't act of m/m directly. so it needs to be fetched to CPU.

Program Counter: Stores the address of the next ~~reg~~ instⁿ to be executed.

Instruction Register: Stores the instⁿ being executed.

Memory Address Register: Used to communicate address b/w ~~en~~ system bus & internal bus

Memory Data Register (Mem. Buffer Reg): Used to communicate data b/w system bus & internal bus.



(1) Transfer addr. of next instⁿ from PC to MAR. MAR floats this addr on system bus & m/m read signal is issued. PC → MAR, $\overline{RD}$

(2) The value from the m/m location is loaded into the MBR. At the same time (in the same CPU cycle) the value of PC can be incremented.

MBR ← Mem[MAR], PC ← PC + 1

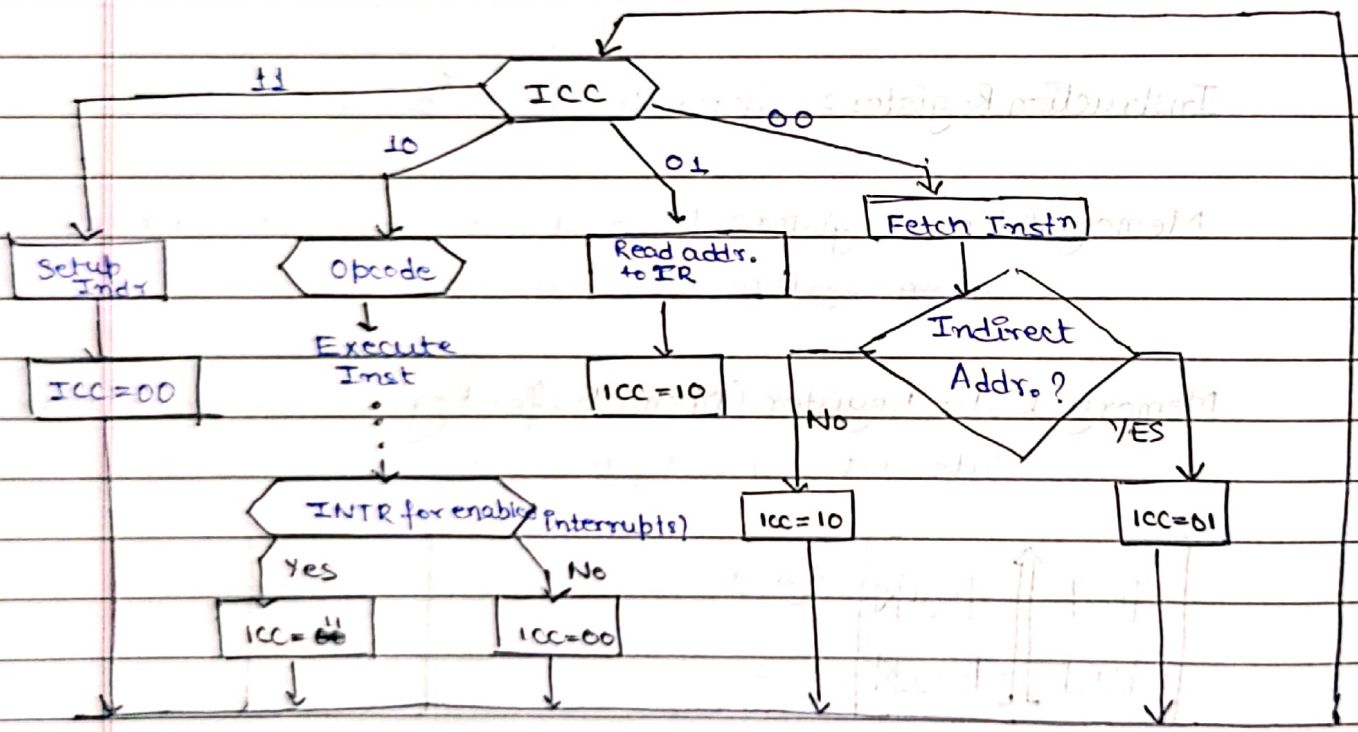
(3) The value from MBR is transfered to IR.

IR ← MBR

- How does CPU Know which cycle is executing & which control signals to generate?

CPU has Instruction Cycle Code which has four values:

00 → fetch
01 → Indirect
10 → execute
11 → interrupt.



- Indired Cycle: If one or more operand uses indirect addressing, then this cycle is executed. It checks the addr. mode of operands incrementally & if indirect, brings the effective addr. in the instn

MAR ← Addr. part of IR
MBR ← Mem[MAR]
Addr. part of IR ← MBR.

Ex. LOAD @ 1000

2052	1000
20	2052

MAR ← 1000

MBR ← [1000]

IR[Mem Addr] ← MBR

LOAD 2052.

Note: The m/m cycles to get eff. address are included in Execution Cycle only unless stated otherwise.

- Execution Cycle: The following steps can take place during this cycle.
  (1) Instruction Decode
  (2) Operand Fetch
  (3) ALU operation
  (4) Write Back.

Ex. ADD R_1, X

t_1: MAR ← X, $\overline{RD}$

t_2: MBR ← Mem [MAR] — No write Back.

t_3: R_1 ← [R_1] + [MBR]

Inst^n Decode happens by decoding the opcode using a n: 2^n decoder, based on which sequence of microinst^n is generated.

Ex. JMP 20(R_1)

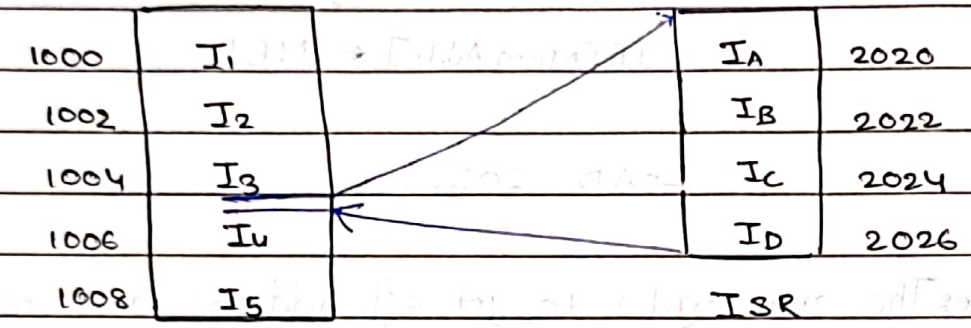
t_1: MAR ← [R_1] + 20, $\overline{RD}$ — No ALU op^n or WB.

t_2: MBR ← Mem[MAR]

t_3: PC ← MBR

• Interrupt Cycle

Ex. Say the prog. $\langle I_1, I_2, I_3 \ldots \rangle$ is executing.



• Say ~~...~~ I_3 is executing when an ~~enabled~~ interrupt comes, i.e., CPU's INTR line is activated.
At this point PC = 1006.

• Once CPU completes executing I_3 it checks if an enabled interrupt has been activated.

- First the program state is stored/saved in m/m.
- Then PC is loaded with base addr. of ISR.
- Then ISR is run.
- Once finished the PC value is restored.

t_1: MBR ← PC
t_2: MAR ← Save_addr, WR
t_3: PC ← ISR BA

* Interrupts

Interrupts can be classified on various basis.

- Vectored & Non-Vectored Interrupt

The interrupts whose ISR is located (stored) in a fixed memory location are called Vectored interrupts.

Non-Vectored interrupt comes or supplies the base address of the ISR, i.e. the devices supplies ISR addr. along with INTR signal while raising interrupt.

NVInterrupt allows for the device to supply different ISR addresses.

- Software & Hardware Interrupt

Hardware interrupts come through hardware lines connected to the CPU.

Software interrupts are raised by the program, such as Divide by 0, or Segmentation fault, etc.

- Maskable & Unmaskable Interrupt

Maskable interrupts mean that they can be disabled and enabled.
Non-maskabe interrupts can't be disabled.

- Interrupt Service Routines are part of OS & stored in Kernel space of m/m.

- Non-maskable interrupts have higher priority compared to maskable interrupts.

- Interrupt Handling Procedure:

(1) Save the PC value, flags & register values on the m/m stack.
(2) Load PC with beginning address of ISR.
(3) Finish ISR when RET inst. is executed.
(4) Return to the point of execution by restoring PC, flag & register state.

* Instruction Set

Set of all the instructions made available to the programmer by the designer & performable by the CPU.

- Data Transfer Instⁿs

  Used to transfer data (copy)

  (1) from Reg to Reg
  (2) from Reg to M/m
  (3) from Mem to Reg.
  (4) from Mem to Mem.
  (5) ~~from~~ Immediate to Mem.
  (6) Immediate to Reg.

(a) MOV Rd, Rs — Move
$R_d \leftarrow [R_s]$

(b) MVI Rd, Imm — Move Immediate.
$R_d \leftarrow Imm$

(c) LDA Addr — Load to Accumulator
$Acc \leftarrow [Addr]$

(d) STA Addr — Store Acc. value to m/m addr
$Addr \leftarrow [Acc]$

(e) XCHG R1, R2 — Exchange values stored in regs.

- Arithmetic Instⁿ:

  Affects all flags: Sign, Carry, Aux Carry, Zero, Parity, Overflow

(a) Add R, M — $R \leftarrow [R] + [Mem]$

(b) Add R — $Acc \leftarrow [Acc] + [R]$

(c) Add M — $Acc \leftarrow [Acc] + [Mem]$

(d) ADC R,M Add with Carry
[R] ← [R] + [M] + [C]

(e) ADI Imm.
Acc ← [Acc] + Imm.

(f) ACI Imm.
Acc ← [Acc] + Imm + [C]

(g) SUB R, M
SUB R
SUB M

(h) SBB R,M Subtract w/ Borrow
(i) SUI Imm
(j) SBI Imm.

(K) INR R Increment
INR M
(l) DCR R Decrement
DCR M

- Logical Instⁿ : (AND, OR, XOR, ROTATE, COMPARE, COMPLIMENT, SHIFT)

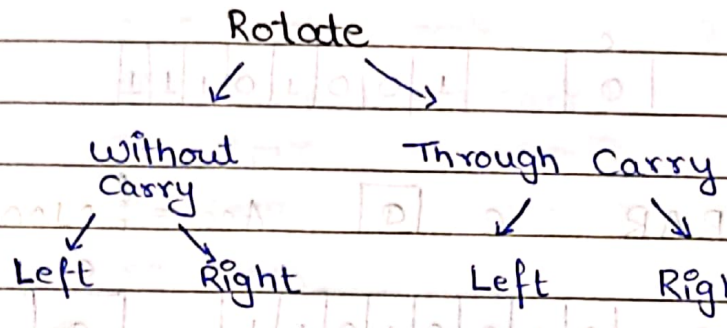
Affects Sign, Zero & Parity flags.

(a) ANA R/M AND accumulator
(b) ANI Imm. AND acc. with imm. value.

(c) ORA R/M (e) XRA R/M

(d) ORI Imm. (f) XRI Imm.

(g)



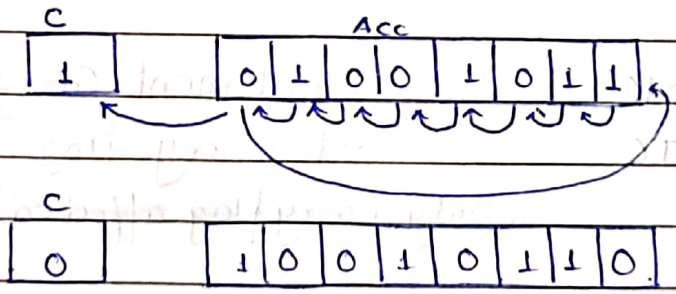
RLC: Rotate Acc. Left w/o Carry

RRC: Rotate Acc. Right w/o Carry.

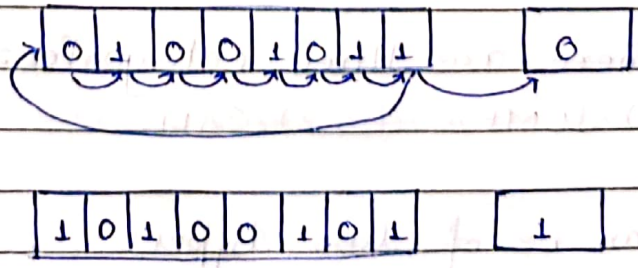
RAL: Rotate Acc. Left with Carry

RAR: Rotate Acc. Right with Carry.

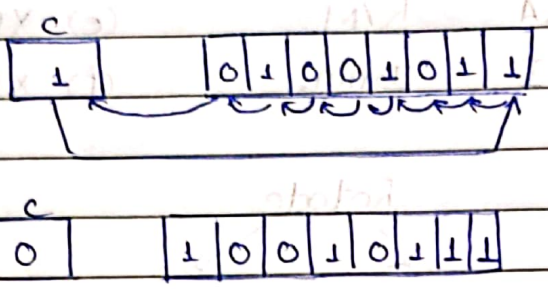
Ex. RLC C = 1 Acc = 01001011.



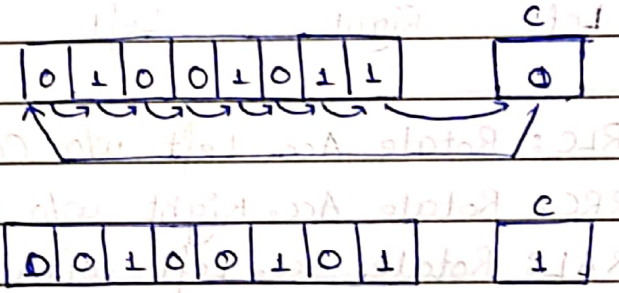
Ex. RRC C = 0 Acc = 01001011.



Ex. RAL C = 1 Acc = 0100 1011.



Ex. RAR C = 0 Acc = 0100 1011.



(h) CMA Complement Accumulator
(No flags affected).

(i) CMC Complement Carry Flag.
STC Set Carry flag
(Only Carry flag affected).

• Branching Instⁿ :

There are three categories:
(1) JUMP (2) CALL (3) RET.

Can be of two types:
(1) Conditional
(2) Unconditional.

(A) Jump

(a) JMP addr (Unconditional Jump)

(b) J X addr

X here can be flag checks, like

JC: Jump if carry

JNC: Jump if not carry

JZ, JNZ

JP, JNP;

JS, JNS

(Sometimes BUN: Branch Uncond.)

(B) Call

(a) CALL addr

(unconditional)

When this instn executes, it first stores the "return address" on the program stack & when RET instn executes, it pops the "return addr" & sets PC to it.

(b) CX addr

X here can be flag checks.

Prog. Stack = Call Stack = Runtime Stack.

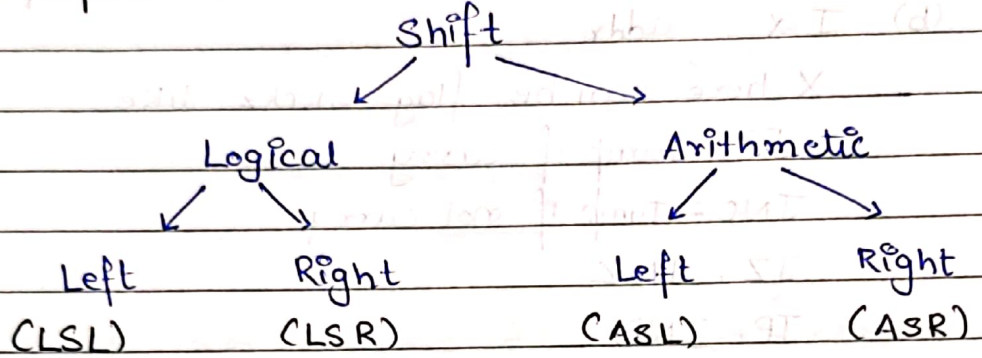
(C) Return

(a) RET

Pops the "return addr" from stack & sets PC to this value.

(b) RX

- Shift Instⁿ



In Arth. Shift the sign bit mustn't change.

Ex. A = | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

(a) ASL

Not Allowed. Sign bit changes.

(b) ASR

| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

Control Unit

- Every Program is a sequence of instructions. Each inst^n has two cycles (in general), fetch & execute. Both the cycle require some microinst^n to run.

Say, fetch cycle:

t_1: MAR ← PC, $\overline{RD}$

t_2: MBR ← Mem [MAR], PC ← PC + 1

t_3: IR ← MBR.

For execution cycle, the microoperations depends on the op^n.

- CPU majorly consists of these components:
  (1) ALU
  (2) Register
  (3) Internal Data Path
  (4) External Data Path
  (5) Control Unit.

- Control Unit is responsible for generation of control signals which make the microinst^n happen. It is also responsible to maintain the sequence of execution.

Ex. For fetch cycle, equivalent control signals would generate in sequence:

t_1: PCout, MARin, $\overline{MEMRD}$

t_2: MBRin, PC-Incr

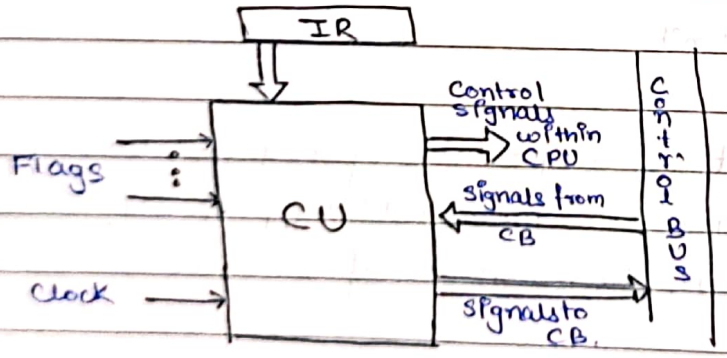
t_3: IRin, MBRout

- Control Unit keeps track of current cycle of Instⁿ based on Instruction Cycle Code register. Based on the value of ICC, the microoperations are run, by CU generating control signals in sequence.

- Microoperations fall into one of following categories:
  (1) Transfer data from one reg to another.
  (2) Transfer b/w internal reg. to external interface.
  (3) Perform ALU operation using registers for inp & output.

- CU performs two basic tasks:
  (1) **Sequencing:** The CU causes the processor to step through a series of micro-operations in the proper sequence, based on program being executed.

  (2) **Execution:** The CU causes each μopⁿ to be performed.



From the IR opcode field, the decoder decodes which operation is to be performed & respectively CU generates signal.

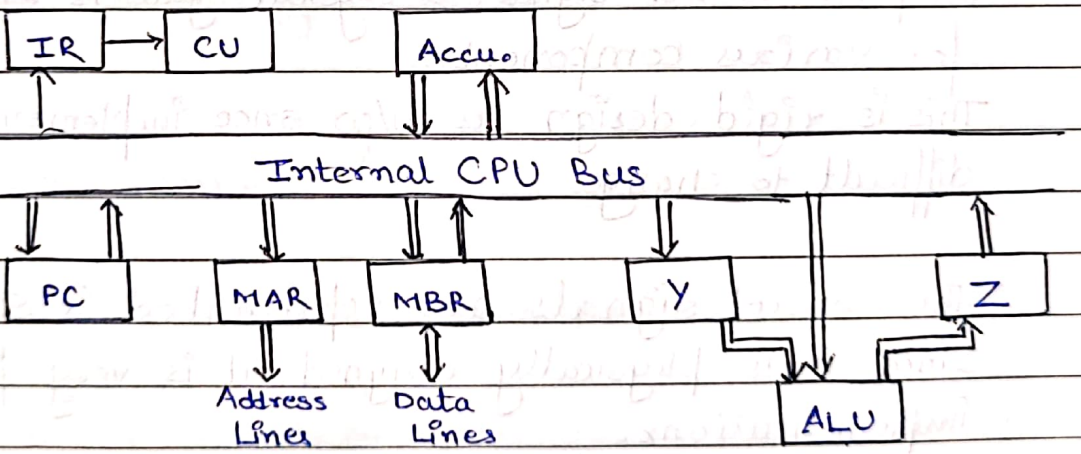
Flags allows for check values for Instⁿs like JNZ, CZ, etc.

Control bus supplies signals such as INTR, HOLD, etc. & receives control signals for mem & IO such as $\overline{RD}$, IN, OUT, etc.

Control signals within CPU allow for data transfer & ALU functions.

Clock simply generates pulses based on which microinsts are run.

* Processor Organization



- A buffer (temp) register Z is required to store result of ALU, otherwise if ALU result is directly allowed to float on Internal bus, then ALU would again take it as input.

Ex: ADA 2002H

t_1: PCout, MARin, Yin, $\overline{RD}$, INCR, Zin

t_2: MBR(ext)in, Zout, PCin

t_3: MBR(int)out, IRin

t_4: IRout, MARin, $\overline{RD}$

t_5: MBR(ext)in, Accout, Yin

t_6: MBR(int)out, ADD, Zin

t_7: Zout, Accin.

* Control Unit is generally implemented in one of the two ways:
  (1) Hardwired
  (2) Microprogrammed.

* Hardwired Implementation

Was used in earlier systems. Suitable for RISC archs.
Control Signals are directly implemented by h/w circuits. So, for a control signal, a physical gate is enabled disabled for various components.
This is rigid design. as h/w once implemented is very difficult to change.

The control signals are represented is SOP form.
Since it is physically designed, it is very fastest of all implementations.

Ex. Desing a hypothetical CPU with 3 instⁿ & 4 control signals using Hardwired CU design.

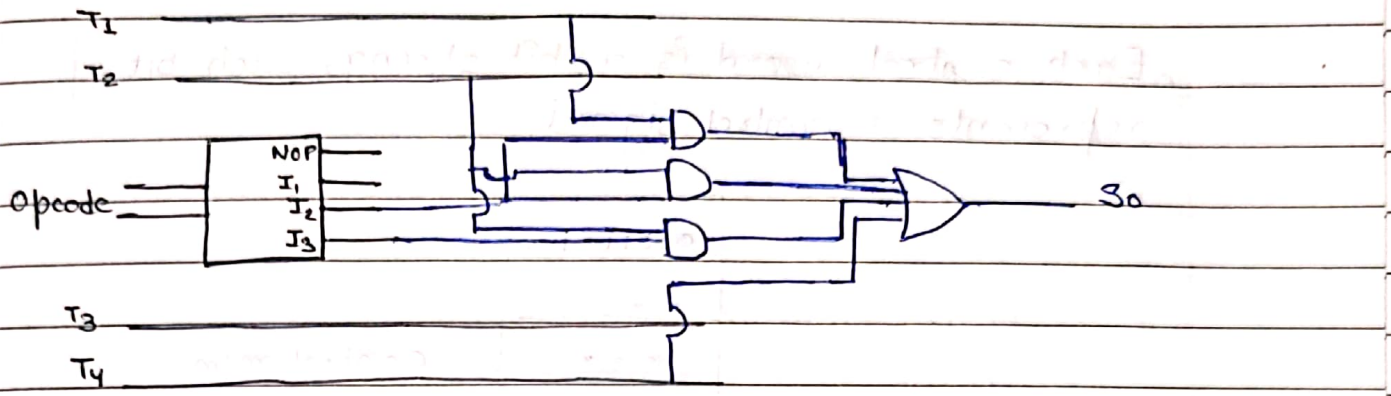
	T_1	T_2	T_3	T_4
I_1	S_1, S_3	S_1, S_2	S_1, S_3	S_0, S_1
I_2	S_0, S_3	S_0, S_2	S_1, S_3	S_0, S_2
I_3	S_3, S_2	S_0, S_1	S_1, S_3	S_0, S_3

$S_0 = T_1 I_2 + T_2 (I_2 + I_3') + T_4$

$S_1 = T_1 I_1 + T_2 (I_1 + I_3) + T_3 + T_4 I_1$

$S_2 = T_1 I_3 + T_2 (I_1 + I_2) + T_4 I_2$

$S_3 = T_1 + T_3 + T_4 I_3$.



Generally a counter is used to generate generate control steps; $T_1, T_2, T_3, \ldots$ Here, since there are four steps, a mod-4 counter can be used, then output of the counter & output of opcode decoder can be used to realise control Signal SOPs.

* Microprogrammed CU

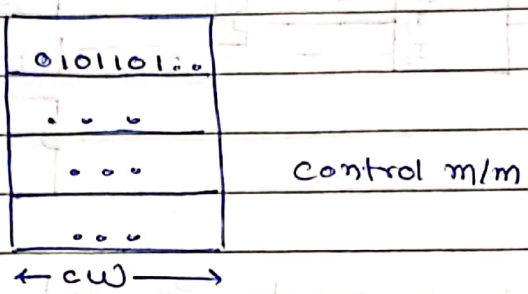
In Microprogrammed CU, the logic of the control unit is specified by a microprogram
A microprogram consists of a sequence of inst^{n}s in a microprogramming language. These are simple inst^{n}s that specify micro-operations.

Suitable for CISC architectures.

Microprogram resides somewhere b/w Hardware & S/W. Firmwares are written in microprogramming language.

There is a control memory which stores Control Words. We have Control Address Reg & Control Buffer Reg. & µp. Program Counter.

Each control word is a bit string, each bit of which represents a control signal.



- Control Word: Control Word consists of various fields:
  (1) Control Signals
  (2) Flags
  (3) Branch Condition
  (4) Next Address

CS	flag	BC	NA

- Horizontal μprog. CU: In this the control signals in CW are in decoded form, i.e. for each control signal there is one bit indicating active/inactive in the CW.

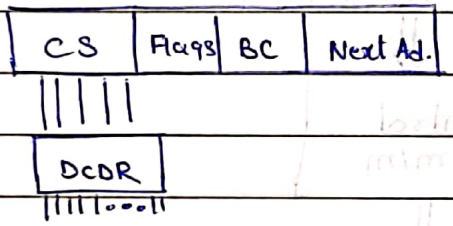
  Ex. 64 Control Signals.
  64 bit for control signals in CW.

This uses a lot of space for each CW since 1bit/control signal.

- Vertical μprog. CU: In this the control signals are stored in the Controlword in encoded form.

  Ex. 64 Control Signals
  ∴ $\lceil \log_2 64 \rceil = 6$ bits only
  for control signals in the CU.

Since the Control signals are stored in encoded form, a decoder is also required to decode all the control signals.

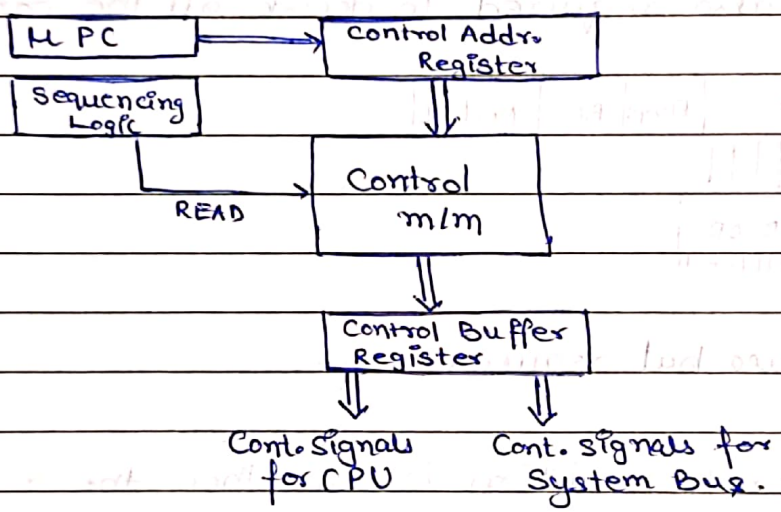


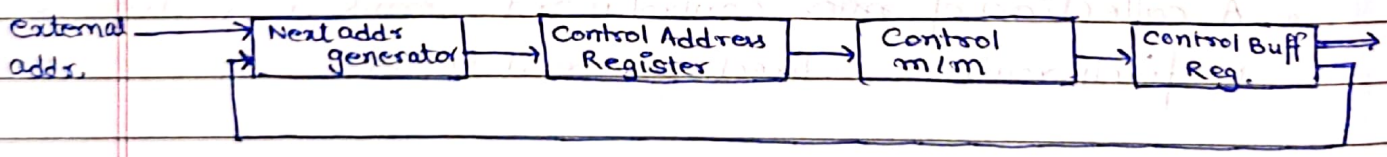
This saves space but requires time.

- If some branch condition is true, then the next address field is used to fetch the next control word from the control m/m. It is transfered to μPC.
- A collection of control words forms a routine, such as fetch routine, indirect cycle routine, etc.
- Each microinstⁿ / CW is associated with a single CPU clock cycle, say a Instⁿ takes 4 clock cyles, it means there are total 4 control words associated with the Instⁿ.
- Since Horizontal μprog CU doesn't require decoding it is faster than Vertical μprog. CU.

Control Memory

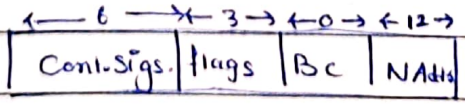
The set of microinstⁿs is stored in Control memory.
The control address reg. contains the address of next μinstⁿ to be read.
When μinstⁿ is read from CM, it is transfered to CBR.
CBR is then connected with control lines (or decoder).
Reading CW from CM is same as executing μinstⁿ.





Ex. A vert. μp. CU supports 256 inst. System uses 8 flags & contains 48 control signal. Each instⁿ req. 16 microoperations. Size of CM?

$\#'\mu ops = 256 \times 8 = 2^{12}$



$\therefore 21 \times 2^{12} = 21 \times 2^{9}\ B = 10752\ B.$

Cache Memory Design

- Cache m/m resides b/w CPU & main m/m. It is made up of static RAM.

- Main m/m can be accessed using physical address. Cache m/m can be on-chip (with CPU) or off chip. On chip Cache can be accessed by virtual addr. generated by the CPU. For off chip cache first a Memory Management Unit converts virtual address to physical address.

* Why Cache Memory

When a process is executed, it performs only two operations: fetching from main m/m & execute on CPU. Majority of the time is spent on fetching from m/m. If we could improve the memory access time, then we can improve overall execution time.

It is observed that not all parts (locations) associated with a process are accessed uniformly. Some are accessed more often than others. This is called Locality of Reference.

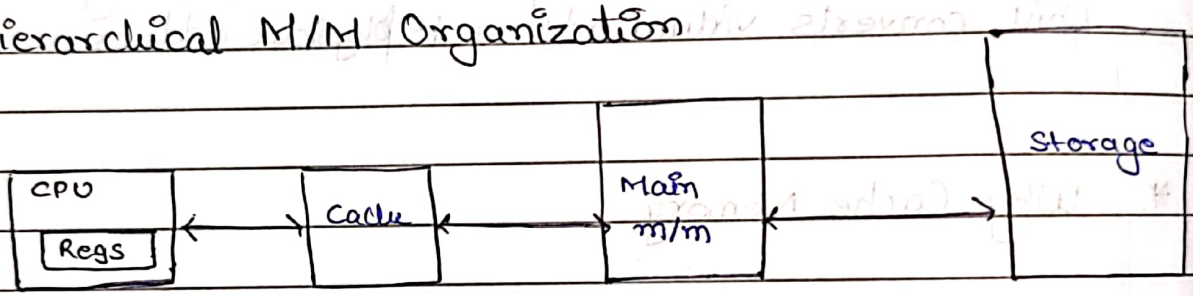
- Locality of Space (Spatial Locality): If CPU access location X at time t_k, then words near to X, such as X-2, X+4 would be accessed in near future, with high probability.

- Locality of Time (Temporal Locality): If CPU accesses loc. X at time t_k, then the same loc. would be accessed in near future with high probability.

Ex. for (i=0; i<10; ++i)
    a[i]++;

Here 'i' shows temporal locality.
'a[i]' shows spatial locality.

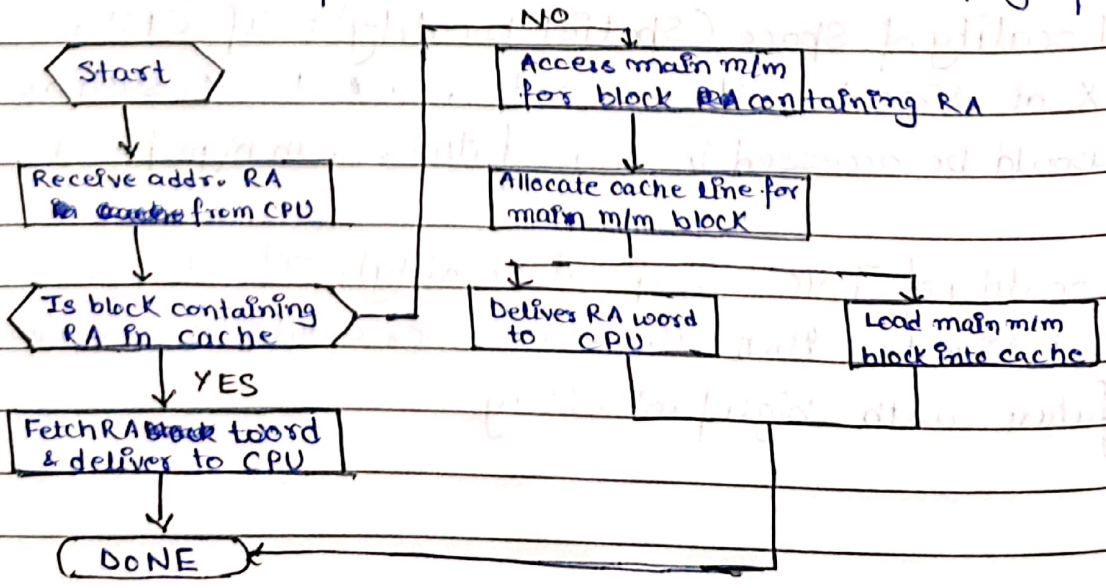
* Hierarchical M/M Organization



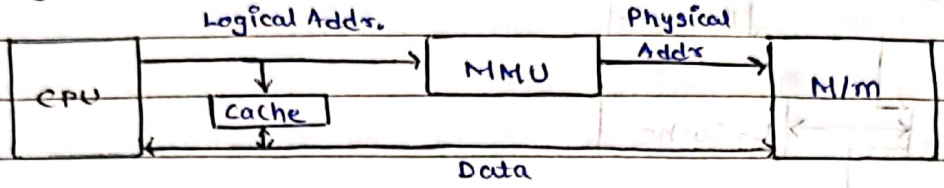
- Speed of access in descending order: Regs. > Cache > Main m/m > Secondary Storage.

- Generally memory in a system is organized in hierarchical fashion.

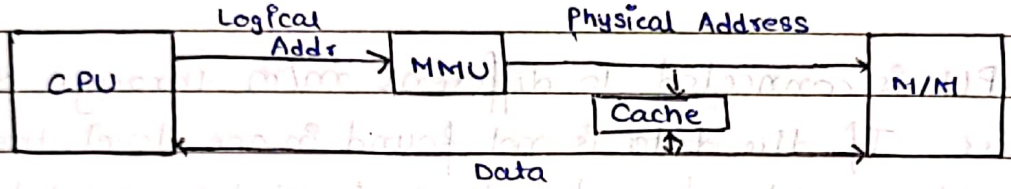
- If data/instn isn't found in cache, its called Cache miss, if not found in main m/m its called page fault.



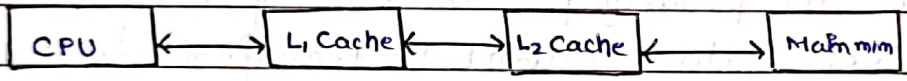
- Logical Cache: Can access cache m/m using Virtual address



- Physical Cache: Can access cache using physical address.



* - Hierarchical Cache Orgⁿ



Block/word travels only to its next neighbour for each m/m.

(Hit Ratio: Probo of finding a addr in m/m X / of m/m X)

(T_x: Access time of mem. X)

Case 1: L_1 Hit:

Access Time = $H_{L_1} \times T_{L_1}$

Case 2: L_1 Miss, L_2 Hit

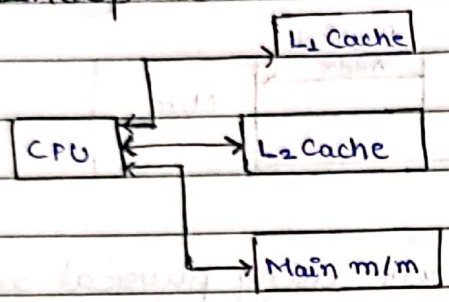
Access Time = $(1-H_{L_1}) \times H_{L_2} \times (T_{L_1} + T_{L_2})$

Case 3: L_1 Miss, L_2 Miss

Access Time = $(1-H_{L_1})(1-H_{L_2}) \times H_M (T_{m/m} + T_{L_1} + T_{L_2})$

Avg Access Time = $H_{L_1} \times T_{L_1} + (1-H_{L_1}) \times H_{L_2} \times (T_{L_2} + T_{L_1})$
$+ (1-H_{L_1})(1-H_{L_2})(T_{m/m} + T_{L_2} + T_{L_1})$

- Independent Cache Org^n:



CPU is connected to different m/m through different buses. If the data is not found in one level, then it would never be found in that level, because data is never copied to that level from higher level, directly transferred to CPU.
There is no penalty for level(i) if data isn't found in that level. Data wouldn't be transferred from level (i+1) to (i), only to CPU.

C1: L_1 Hit

Access Time = $H_{L_1} \times T_{L_1}$

C2: L_1 Miss, L_2 Hit

Access Time = $(1 - H_{L_1}) \times H_{L_2} \times T_{L_2}$

C3: L_1 Miss, L_2 Miss, M/m hit

Access Time = $(1 - H_{L_1})(1 - H_{L_2}) T_{m/m}$

$$AT_{avg} = H_{L_1} \times T_{L_1} + (1 - H_{L_1}) H_{L_2} \times T_{L_2} + (1 - H_{L_1})(1 - H_{L_2}) H_{m/m} \times T_{m/m}$$

* Cache Structure

Cache m/m is divided into equal sized units called Blocks / Lines. At a time b/w cache & m/m the least size of transfer is one block (one word at a time)

Ex. 16B cache. 4B cache line.

00	00 01 10 11
01	00 01 10 11
10	00 01 10 11
11	00 01 10 11

$\#\ lines = \frac{16B}{4B} = 4$

∴ 4 lines, each of 4B.

* Cache Mapping

Initially the cache is empty (or lines are invalid) when CPU requests a word from m/m, it is bought into the cache as well.

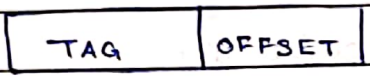
Cache Mapping techniques answer the question, which block of m/m will be mapped to which cache line.

• Fully Associative

CPU generates an address to access main m/m. Fully Associative Cache will divide the address into two parts: Tag & Offset. Offset is the location of (index) of word within a cache block.

In fully associative cache, any m/m block can be put in any available (empty) cache line. Each cache line additionally

stores tag bits. Each cache line has a comparator attached. When CPU sends address to Cache to check for hit/miss, Cache takes the tag part of the address and compares it with tag part of all the cache lines. If any comparator returns true, then cache hit.



Address fmt for Fully-Assoc.

OFFSET size: No. of bits req. to index a words in block.
⇒ $\lceil \log_2 (\text{block size in words}) \rceil$

TAG size: All bits other than offset. / No. of memory blocks
⇒ Addr len - offset size.

All tags are compared in parallel.
Tag Storage size = #tag bits x #cache lines.

one mux selects the block ↑

If a tag matches, then all the words from that block go to Multiplexer, with the offset bits of addr. acting as select line.

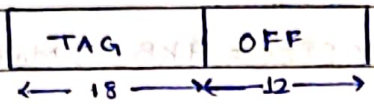
comparators = #cache lines
bits in comparator = size of tag field
multiplexer = 1

Ex. CM = 256 KB 4KB Blocksize 1GB m/m.

$$\#\ \text{lines} = \frac{256\ KB}{4\ KB} = 2^6 = 64$$

words per block = $\frac{4K}{\rightarrow 2^{12}}$

$$\#\ \text{blocks} = \frac{2^{30}\ B}{2^{12}\ B} = 2^{18}$$

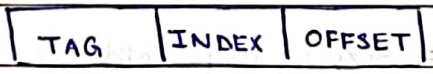


comparators = 64
size of comparator = 18 bits
tag dir. size = 64 x 18 bits
→ 144 B

• Direct Mapping

Rule: Cache line no. for m/m block = m/m block no. % no. of cache lines

Mem. address is divided into three parts:



INDEX: Cache line index.
TAG: Index of m/m blocks within the cache line
OFFSET: Word offset.

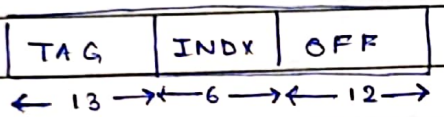
The Direct mapped cache stores m/m blocks based on modulo of m/m block no. by no. of cache lines.
When addr. from CPU comes, cache directly jumps to the specified cache line using INDEX. TAG represents, out of all the m/m blocks which can possibly exist in the particular cache line, which one is currently residing there. OFFSET is word offset within the cache block.

Ex. CM = 256 KB Block Size = 4KB Main m/m = 2GB.

$$\#\text{blocks} = \frac{2GB}{4KB} = \frac{2^{31}B}{2^{12}B} = 2^{19}$$

$$\#\text{cache lines} = \frac{256KB}{4KB} = 2^6$$

tag dir size = $2^6 \times 13 = 104$ B



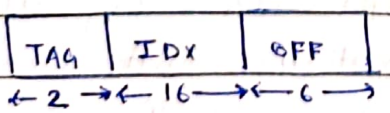
One multiplexer required to fetch the tag bits of particular cache line with INDEX bits as select line.
One comparator req. to compare TAG bits.

multiplexers = 1
comparator = 1
size of comparator = size of tag field.
Tag dir. size = # cache lines x size of tag field.
After tag matches of the specified index in add. all words of that index go to multiplexer with offset bits acting as select lines.

Ex. CM = 4MB BlockSize = 64 B
Mm addr = 0x AB12CF. Find Cache line no of this word addr?

Offset bits = 6

$$\#\text{cache lines} = \frac{2^{22}B}{2^6 B} = 2^{16}$$



∴ Block No. of cache = 10 1011 0001 0010 11

Set Associative Mapping

A combination of Direct & Fully Assoc. mapping.
We take the cache lines & divide into set of K-lines.
K-way Set Associate means that each set is made up of K lines.

For a m/m block the Set Associative Cache finds the set index by: m/m block no. % no. of sets in cache.
Within the designated set the block can reside in any of the "K" cache lines. Each cache line consists of a tag bits indicating that out of all possible m/m blocks which could be mapped to the particular cache set, which ones are residing in the set currently, specifically in that block.

When CPU generates & sends address to the cache, it is interpreted in three parts:

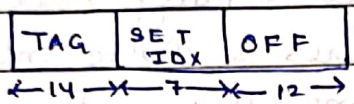
TAG	SET INDEX	OFFSET

A set is selected from the cache based on set index. All the blocks within the set (K) send their tag to K different comparators to compare with Tag field. If matched with any of the K blocks, its a hit & offset is used to select a particular word from the block.

comparator = K size of comparator = size of tag field.
Tag Dir. Size = # cache lines x size of tag field
mux = 2 (one to select block, another to select word).

Ex. CM = 4MB Block size = 4KB Main m/m = 8GB

8-way set Assoc.



cache lines = 2^{10}

m/m blocks = 2^{21}

sets = $\frac{2^{10}}{2^3} = 2^7$

comps = 8 Size comp = 14b.

Tag dir size = $2^{10} \times 14 = 1792$ B

mux = 2

Mux1 : $2^3 : 1$ (Block selector)

Mux2 : $2^{12} : 1$ (Word Selector)

- Along with storing tag info, cache lines also need to store a few extra bits, such as Valid / Invalid, Dirty, etc.

* Cache Miss

When CPU demands a particular word from cache & the containing block is not in the cache, its called a Cache Miss & the cache has to incur a miss penalty. There are three types of cache misses:

- **Compulsory Miss:** The very first access to a block can't be in the cache, so the block must be bought into the cache. Compulsory misses are those that happen even if you have infinite cache.

- **Capacity Miss:** If a block is being discared from cache due to the cache being full and later retreived, it is known as capacity miss.
  Fully Associative cache would have high capacity miss.

- **Conflict Miss:** If the mapping strategy isn't fully associative, then conflict misses would happen (when its not compulsory & cache has space) when a block is discarded due to conflict with another block & later retreived.
  Ex. In direct map, even if cache has empty lines a block maps to a particular line, and if its occupied then the occupying block would be discarded, causing conflict miss.

* Cache Latency: The time required to access the first word of the block.

* Cache Read.

When CPU tries to read a word from the cache:

(1) Block is available in cache: $T_c \times H_c$

(2) Block is not available in cache: The block would be transferred from m/m:

(a) Read Allocate: The block is parallely transferred to both CPU & cache. $[(1-H_c)(T_{mm}+T_c)]$

(b) No Read Allocate: The block would directly go to CPU only. $[(1-H_c)T_{mm}]$

Cache Write

When CPU is tring to write to a particular address:

(1) Block is available in cache: $[T_c \times H_c]$

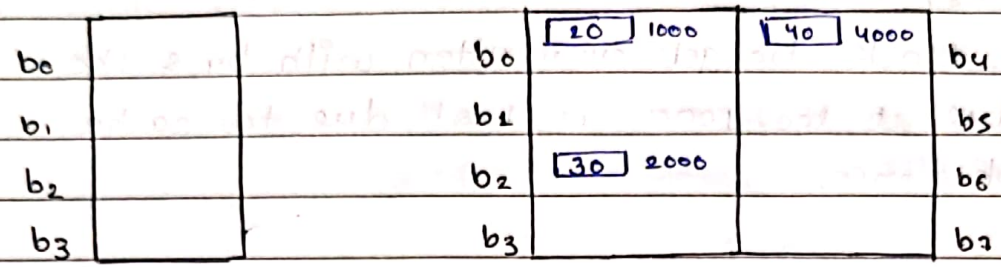
(2) Block isn't available in cache:

(a) Write Allocate: The block is transferred from m/m to cache. $[(1-H_c)(T_c+T_{mm})]$

Ex. Say CPU wants to add two nos.

```
LOAD  r0, 1000
LOAD  r1, 2000
ADD   r2, r1, r0
STORE 1000, r2
LOAD  r3, 4000
```

Say 4 block Direct Cache & 8 block main m/m.



DM Cache Main m/m

LOAD r0, 1000 : Read Allocate.

b0, mm → b0, cache → r0, CPU

LOAD r1, 2000 : Read Allocate.

b2, mm → b2, cache → r1, CPU

ADD r2, r1, r0 :

$r_2 \leftarrow [r_1] + [r_0]$

STORE 1000, r2 :

r2, CPU → b0, cache

At this point block in cache contains value 50 for addr 1000 & main m/m contains value 20.

This is Cache Coherence problem.

- Cache Coherency: Same address in two different memories contains different values.

~~How when do~~

LOAD R3, 4000 :

The block b0 gets overwritten with b4 & the value at loc. 1000 is lost due to cache coherence.

- Write Allocate : A block to be written, on a cache miss is bought from m/m to cache, followed by a write hit. Write Alloc. acts as Read Allocate.

- No Write Allocate : Write miss don't affect the cache. Block is directly modified in main m/m.

- Write Through : The information is written both to the block in cache & in main m/m.

- Write Back : The information is written only to the cache. The modified cache block is written to main m/m only when it is replaced.

- Dirty Bit : A bit is associated with each cache block, which indicates, while in cache if the block has been modified or not. During replacement only the blocks which have been modified are written back.

- With Write Through by default No write allocate is used.

$T_{avg(r)} = H_c T_c + (1 - H_c)(T_{mm} + T_c)$

$T_{avg(w)} = H_c (\max(T_c, T_{mm})) + (1 - H_c)(\max(T_c, T_{mm}))$

$\Rightarrow H_c (\max(T_c, T_{mm})) + (1 - H_c) T_{mm}$

(Heierarchical)

$T_{avg(r)} = H_c \times T_c + (1 - H_c) T_{mm}$ (Parallal)

$T_{avg(w)} = T_{mm}$ (Parallel)

- With Write Back by default Write Allocate is used

(Write Back, Write Alloc)

$T_{read} = T_{write} = H_c \times T_c + (1 - H_c)[\%\text{ of dirty bits} \times (T_{mm} + T_{mm} + T_c) + \%\text{ of not dirty} \times (T_{mm} + T_c)]$

For the dirty blocks, first the block is written to main m/m, then new block is accessed from m/m & then written to cache.

Ex. Consider a two level m/m hierarchy. L_1 (cache) has AT = 10 ns & main m/m has a AT = 20 ns. Writing & updating into L_1 & m/m takes 20 ns & 30 ns resp. $H_{L_1} = 20\%$. Avg writing time in case of Write Through?

$T_{avg(w)} = H_c \times (\max(T_{mm}, T_c) + (1 - H_c) \times T_{MM}$

$= 0.8 \times 30 + 0.2 \times 30$

$\Rightarrow 30$

- In Hierarchical access in case of a read miss, the block is first read from m/m & put into cache (read allocate) & then Instⁿ is restarted & this time its a read hit & word is read from cache.

  ∴ Tmm + Tc

  If no read allocate, then Tmm.

- For write if its a hit both cache & m/m are updated simultaneously in Write Through, & on miss, only m/m is updated.

Ex. A cache line is 64 B. The main m/m has latency of 32 ns & bandwidth of 1 GB/s. The time req. to fetch entire cache line from the main m/m is?

(1 GB/s =

Time req = Time to access first word
+ Time to transfer all subsequent words

$\Rightarrow 32 * 10^{-9}$ sec + $\frac{64}{10^9 \text{ B/s}}$

$\Rightarrow$ 32 ns + 64 ns

$\Rightarrow$ 96 ns.

* Virtual Cache

- Till now what we've seen is cache being accessed through Physical m/m address: Physical Tag Physical Index.

CPU sends arrr. to MMU to convert virtual addr. to Physical addr. either using TLB or page table. & then the addr is sent to cache.

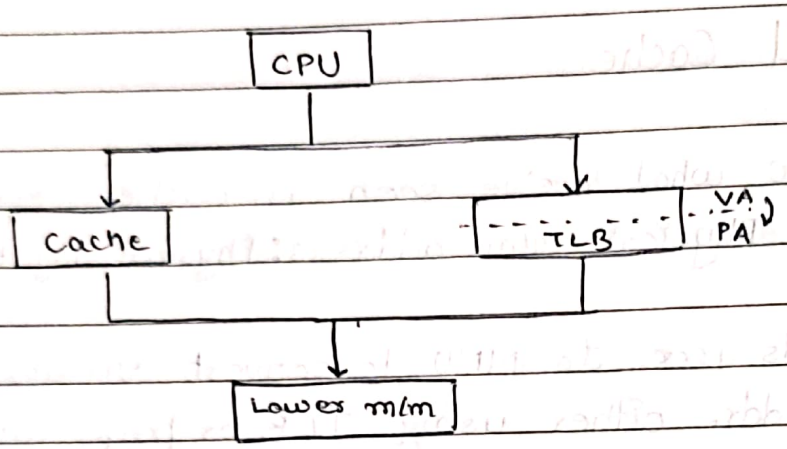
- Virtual cache allows us to simultaneously search the cache & translate Vaddr to Paddr. in MMU. Virtual addr. can use: Virtual Tag, Virtual Index.

CPU sends V.A. to cache, which interprets Tag & Index fields from it, and to MMU to translate to P.A.

- Tag can be Physical & Virtual.
  Index can be Physical & Virtual.

Based on this two types of Cache designs are used
(1) Virtual Index Physically Tagged
(2) Physically Index Virtual Tagged [NOT used]

- Problem with VA: VTVI
(1) In case of context switch, cache has to be flushed.
(2) Diff. VA referring to same PA: Synonym problem.
(3) Diff. PA referring to same VA: Homonym problem.



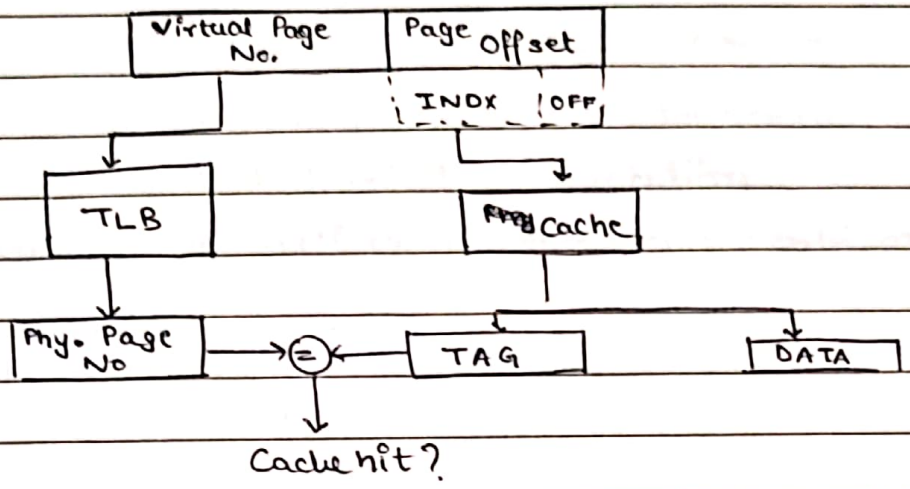
- Synonym Problem: Two virtual ~~pages~~ addr. refer to same physical location. If both are in cache, then writing to one line is invisible to other. So if other line is read, it will read wrong info.

- Homonyn Problem: A single VA, for two different process can refer to different Physical loc. If CPU generates such an address, then cache won't know which location's data to bring from m/m.
  Sol^n: (A) Store PID identifier along with VA.
  (B) Flush on Context Switch.
  (C) Physical Tags

- Virtual Index Physical Tag

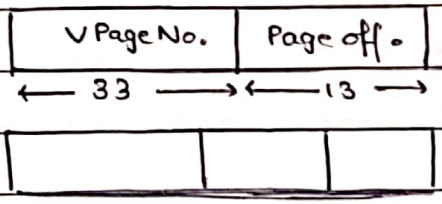
Cache block size can't exceed page size, so as to avoid synonym problem.

Page size x Assoc. $\geq$ Cache size



In case, $C > P \times A$, then bits from Virt. Page No. field would be used & would cause Synonym problem.

Ex. VA = 46 bits Page size = 2^{13}B. 1 MB cache with 16 way Assoc.



$C = 1\,MB = 2^{20}\,B \qquad A = 16 = 2^4 \qquad P = 2^{13}$

$\therefore \quad 2^{20} > 2^4 \times 2^{13}$

$20 > 17.$

∴ 3 bits from the Virtual Page No. field would be used for set index. Hence 2^3 virtual pages would map to same set.

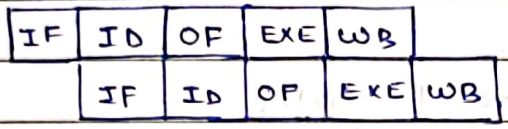
Hence 8 page colors req. to avoid Synonym.

For each color cache would return 1 physical addr.

Pipelining

Pipelining is an implementation technique where multiple instⁿs are overlapped in their execution. It takes advantage of parallelism of actions needed to execute an instⁿ.

Ex.



* Cycle Time: The timeperiod b/w two rising edges of a clock.
  Frequency of Clock: $\frac{1}{\text{Cycle Time}}$.

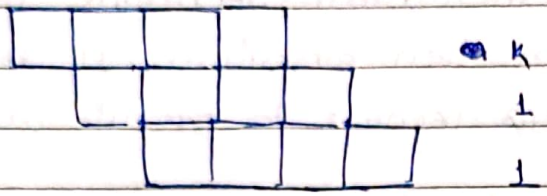
* There are various stages to execute an instⁿ. Each stage requires / interacts with different ~~component~~ independent component.

  Ex: IF : Mem
  ID : Instⁿ Decoder / CU
  OF : Mem / Register
  Op. : ALU
  WB : Mem. / Register.

The cycle time of CPU is set large enough to make sure each stage can be executed in almost 1 clock cycle.

Pipelining takes benifit of independence of these components, to start executing further instⁿs before finish of the first without hampering the order.

Ex. Say K-step pipeline for n inst^n.



∴ for first inst^n ⇒ K cycles.
for remaining (n-1) inst^n = 1

∴ #cycles = K + (n-1)
Say cycle time is tp
∴ tp x (K+n-1)

Say. K = 5

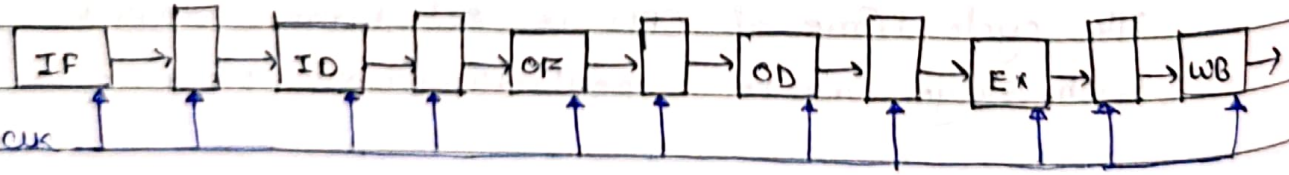
n = 10 : CPI = (10+5-1)/10 = 1.4

n = 20 : CPI = (20+5-1)/20 = 1.2

n = 100 : CPI = (100+5-1)/100 = 1.04

∴ If we keep increasing inst^n CPI ≃ 1.

* Pipeline Design



All the stages are followed by buffers to store the output of the stage. Each buffer has a common clock & hence is sync.

∴ The clock period would depend on the slowest stage delay + buffer delay.

∴ t_p = max (stage delays) + buffer delay.

Ex. N = 1000 K = 5 t_p = 2ns t_{np} = 2ns

$\#cycles_P$ = 1000 − 1 + 5 = 1004 ∴ Exec. $Time_P$ = 2008 ns

$\#cycles_{NP}$ = N × K = 1000 × 5 = 5000. ∴ Exec $Time_{NP}$ = 10000 ns

- Speed Up = $\frac{\text{Performance of Pipeline}}{\text{Performance of Non PL.}}$

$$\Rightarrow \frac{1/ET_P}{1/ET_{NP}}$$

$$\Rightarrow \frac{ET_{NP}}{ET_P}$$

$$\Rightarrow \frac{n \times t_{pNP}}{(n+K-1)\,t_p} = \frac{n\, t_{NP}}{(n+K-1) \times t_p}$$

- Ideal Condition : $n >> K-1$.

- $Speedup_{ideal} = \frac{n\, t_{NP}}{n\, t_p} = \frac{t_{NP}}{t_p}$

- Theoretically, $t_{np} = K * t_p$

$$\text{Speedup}_{max} = \frac{t_{ne}}{t_p} = \frac{k \times t_p}{t_p} = k$$

- $\text{Efficiency } (\eta) = \frac{\text{Speedup}}{\text{No. stages}} = \frac{S}{K}$

- $\text{Throughput} = \frac{\#\text{tasks}}{\text{time taken}} = \frac{n}{(n+k-1)t_p}$

$$\text{Throughput}_{ideal} = \frac{n}{nt_p} = \frac{1}{t_p}$$

* Types of Pipelining

- Uniform Delay Pipeline: Every stage of the pipeline executes in same cycle time (t_p), i.e. each stage of pipeline takes exactly one cycle & no less.

$$T_p = t_p + \text{buffer delay}$$

- Non-Uniform Delay Pipeline: Stages have different stage delays, and hence, $t_p = \max(\text{stage delays})$.

&

$$T_p = \max(\text{stage delay}) + \text{buffer delay}$$

Ex. Consider an instⁿ pipeline with speedup factor of 10 & operating with 80% efficiency.

$$S = \frac{1/ET_P}{1/ET_{NP}}$$

$$\Rightarrow \frac{K \times t_p}{(n-K+1)t_p}$$

$$\eta = \frac{S}{K} = \frac{10}{K} = 0.8$$

$$\Rightarrow K = \frac{10}{\frac{8}{10}} = \frac{100}{8}$$

$$\simeq 13$$

Ex. Proc. A

$K = 8$

uniform delay

$N = 100$

$t_p = 20$ ns

Proc. B.

$K = 5$

Non-uniform delay.

$N = 100$.

Stage Delays ⇒ 2ns, 3ns, 1ns, 2ns, 2ns

Find speed up if A proc. is used instead of B?

Proc. A

$ET = (107) \times 2$ ns

Proc. B.

$ET = (104) \times 3$.

$$\text{Speedup} = \frac{1/ET_A}{1/ET_B}$$

$$\Rightarrow \frac{104 \times 3}{107 \times 2}$$

$$\Rightarrow 1.45$$

* RISC vs CISC

RISC	CISC
(1) Fewer instⁿ, fewer addressing modes.	(1) More instⁿ & more addressing modes.
(2) Simple instⁿs.	(2) Complex instⁿs.
(3) Fixed len instⁿ.	(3) Variable length instⁿ.
(4) 1 cycle per instⁿ.	(4) Multiple cycles per instⁿ
(5) Easy to implement using hardwired CU.	(5) Difficult with hardwired CU.
(6) Register to Register ALU opⁿs only.	(6) Reg to Mem & Reg to Reg ALU opⁿ.
(7) Suitable for pipeling	(7) Unsuitable.

- In RISC systems, the opⁿs can only be performed b/w registers, so if an opⁿ needs some data from some m/m loc., it first needs to be loaded into some register, then opⁿ is performed, then the result is stored back into register. The result may be moved from reg. to m/m.

Ex. LOAD R_1, 2010
ADD R_1, R_2
STORE R_1, 2010.

* RISC Pipeline

- The RISC instⁿ set consists of the following instⁿs.
  (1) LOAD & Store
  (2) ALU Instⁿs
  (3) Branch InStⁿs.

- RISC Pipeline Stages
  (1) Instⁿ fetch cycle : Fetch instⁿ & update PC.
  (2) Instⁿ decode & register fetch cycle : decode the instⁿ & fetch from registers parallely. All registers can be read from in parallel.
  (3) Execution / Effective Addr. Calc. : ~~If ALU operation, then~~ ALU operates on the operands from previous cycle depending on the instⁿ type:
    (a) Memory Reference : The ALU adds base & offset register to form effective address.
    (b) Register-Register ALU instⁿ : The ALU performs the opⁿ specified by the opcode in the instⁿ on values read from registers.
    (c) Register-Immediate ALU Instⁿ : The ALU performs operation specified by the ALU opcode, on register & immidiate value.
  (4) Memory Access : If the instⁿ is a load, the mem. does a read using the effective address computed in the previous cycle. If it is store, then the m/m writes the data from the register read in previous cycle using the eff. addr.
  (5) Write Back :
    (a) Register to Register ALU instⁿ or LOAD instⁿ : Write the result into the register file.

* Basic Performance Issues

- For fewer inst^{n}s, pipelining ~~reduces~~ increases the execution time.
- Imbalance of stage delays results in reduced performance since the clock can run no faster than the slowest stage.

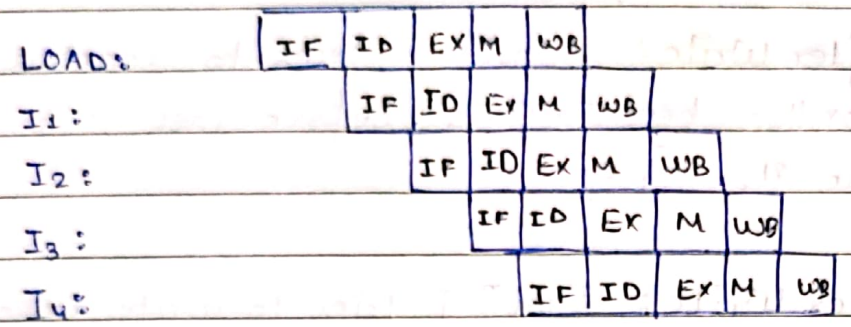
* Pipeline Hazards:

- Anything causing stall cycle is called Hazard.
- Stall Cycles: Waiting cycle in pipeline in which no work is done, i.e. cycle at end of which no inst^n completes. or cycles in which no new inst^n enters the pipeline.
- Structural Hazards: Hazards arising due to structural or resource conflicts.
- Data Hazards: When an inst^n depends on the result of previous inst^n in a way that it affects pipelined execution.
- Control Hazards: Arise from the pipelining of branch & other inst^{n}s which change PC content.
- The CPI gets hampered due to stall cycles:

  $CPI = CPI_{ideal} + \#\text{stall cycles}$.

  $\therefore ET_p = (n - 1 + K + \text{Stalls}) \times t_p$.

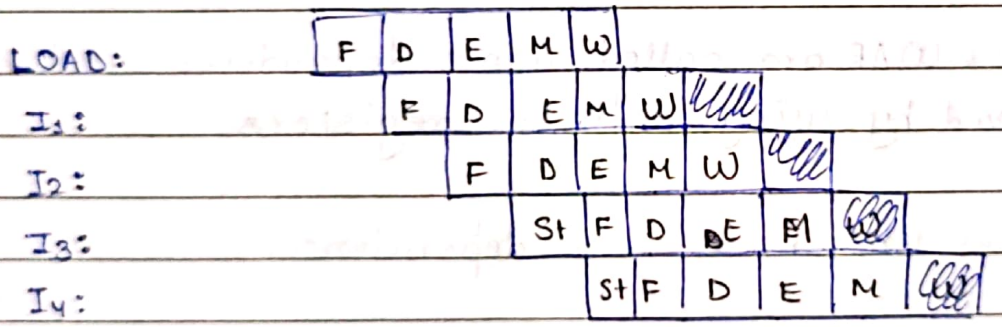
Ex. [Structural Hazard]



∴ 9 cycles.

In cycle 4, both LOAD & I3 access memory simultaneously hence this is Structural Hazard.

∴ We'd have to stall.



∴ inst^{n} takes 10 cycles.

- Sol^{n} to this structural hazard is to keep separate inst^{n} & data memory.

Ex. [Data Hazard]

ADD R_1, R_2, R_3

SUB R_4, R_1, R_5

AND R_6, R_1, R_7

OR R_8, R_1, R_9

XOR R_{10}, R_1, R_{11}

Types of Data Dependence

(1) Read After Write: Instⁿ j tries to read some value written by Instⁿ i, before instⁿ i is able to write it. (i < j)

(2) Write after Write: Instⁿ j tries to write to some reg. before i^{th} inst, which instⁿ i also writes. (i < j)

(3) Write after Read: Instⁿ j tries to write to some reg. before i^{th} inst, which instⁿ i has to read from (i < j).

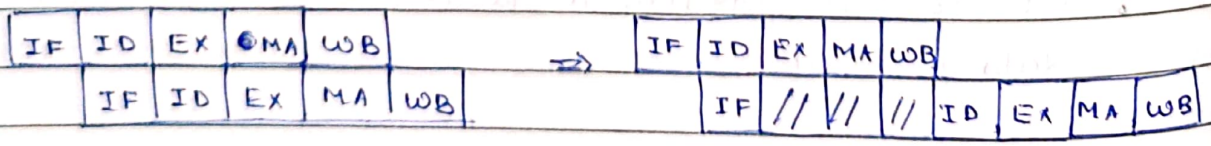
WAW & WAR are called name dependence, and can be removed by using different registers.

RAW is known as true dependence.

Ex. [RAW]

ADD R_1, R_2, R_3

SUB R_5, R_1, R_4



Ex. [WAW]

ADD R_1, R_2, R_3

SUB R_1, R_4, R_5

IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB.

Since both update the value of R_1 in WB, so order maintained.

Ex. [WAW]

Say m/m takes 3 cycles to read

```
LOAD  R1, 0x3FFE2
ADD   R1, R2, R3
```

IF	ID	EX	M	M	M	WB
	IF	ID	EX	M	WB	

⇒

IF	ID	EX	M	M	M	WB
	IF	ID	EX	M	//	//	WB

This creates problem, as newer value is overwritten.

Ex. [WAR]

```
ADD  R1, R2, R3
SUB  R2, R5, R6.
```

IF	ID	EX	M	WB	
	IF	ID	EX	M	WB

∴ No hazard.

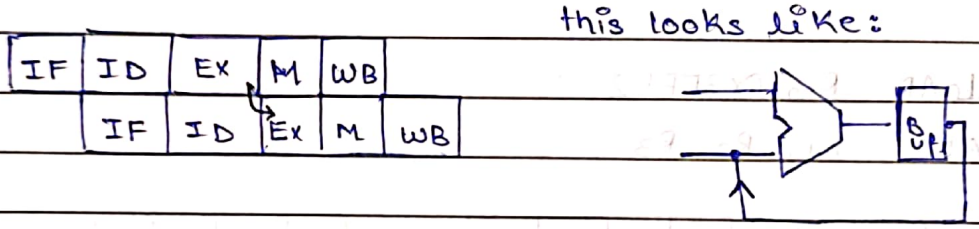
- Only RAW dependency is sure to cause hazard.
- WAR is called anti-dependency as it rarely causes stall.
- Register renaming can solve both WAR & WAW dependencies.

• Operand Forwarding / Data Forwarding / Bypassing

Result of previous operations can directly be transferred from ALU buffer to ALU directly for next operation.

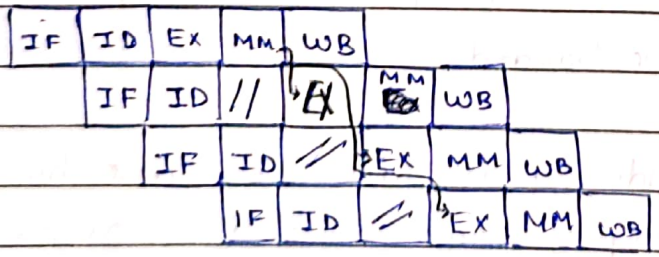
Ex. ADD R_1, R_2, R_3
SUB R_4, R_1, R_5

this looks like:



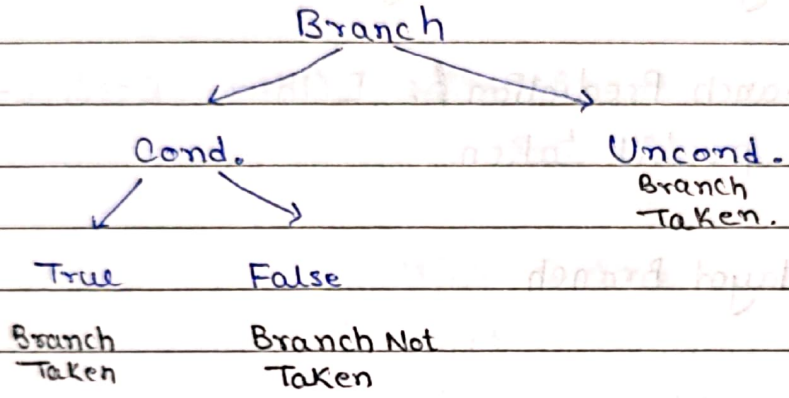
This helps to prevent stalls.

Ex. LOAD R_1, 10(R_2)
SUB R_4, R_1, R_5
AND R_6, R_1, R_7
OR R_8, R_1, R_9



Bypassing can't handle all RAW hazards. There can still be stalls.

- Control Hazards: Cause huge performance loss. When a branch is taken it may/maynot take PC to current value + 1.

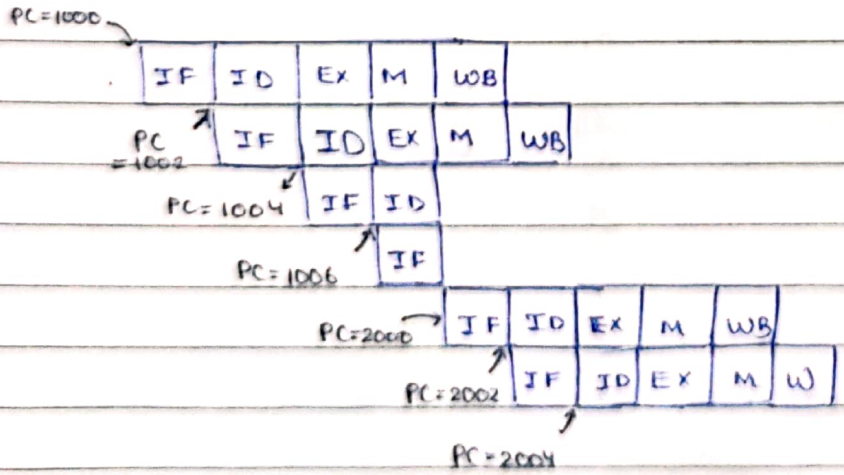


Ex. 1000: ADD R_1, R_2 2000: ADD R_2, R_4

1002: JMP 2000 2002: SUB R_3, R_4

1004: SUB R_2, R_4

1006: MUL R_3, R_4



Instⁿ at loc. 1002 calculates the target of JMP in EX cycle, then takes a jump, till this happen two instⁿ at 1004 & 1006 have already been fetched. Once branch is taken the results of these 2 instⁿs need to be flushed from buffers before the next instⁿ from 2000 can be executed.

- Branch Penalty : The penalty incured due to fetching wrong instⁿ following branch.

- Reducing Branch Penalty :

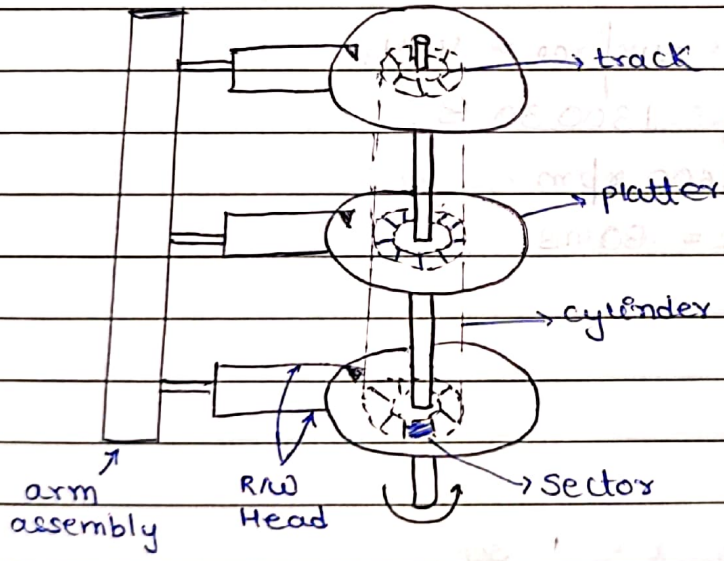
  (1) Branch Prediction : Either predict-not taken or predict taken

  (2) Delayed Branch

Disk Structure

Main m/m is small & volatile & hence we need Disk Storage to sustain our data.

- Sector: The smallest unit of data that can be read or written to on a disk.
- Track: Made up of multiple sectors.
- Platter: Collection of tracks.
- Cylinder: Collection of single track across platters.



- Platters can be single sided or double sided.

- Rotation Time: Time to one complete rotation.

Rotational Latency: Half of Rotational time. (Avg Latency)

$$\frac{RT}{2}$$

- Seek Time: Time to align R/W head over the desired track.

- Sector Transfer Time: $\frac{1 \text{ Rotation Time}}{\text{No. of sectors on the track.}}$

- Disk Access Time = Rotational Latency + Seek Time + 1 sector access time + any additional delay (if given)

Ex. Num tracks per surface = 404.
Track Capacity = 130030 B.
Disk Speed = 3600 rpm.
Avg seek time = 30 ms.

(1) Avg latency.
$\Rightarrow \frac{1}{2}$ RT.

$RT = \frac{3600}{60} = \frac{1}{x} \Rightarrow x = \frac{1}{60}$ sec.

$\therefore AL = \frac{1}{120}$ sec. $= \frac{25}{3}$ ms

(2) Storage Capacity = # surfaces x 404 x 130030 B
$\simeq$ 50 MB x # surface.

(3) Data Transfer Rate.

1 Rot → 1 track

⇒ $\frac{50\ ms}{3}$ ⇒ 130030 B.

⇒ 1 sec ⇒ 130030 × 60 B

∴ Data transfer rate = 7618.94 KB/s

- Disk Transfer Time: Time to transfer ↑ one sector (or block)

- Avg disk access time = Avg Seek Time + Avg Rot. Latency + Disk Transfer time + Controller Delay + Queing delay

* Disk Scheduling

Goal of Disk Scheduling is to minimize head movement, since it affects seek time.

So given a queue of track ids we want to schedule the seeks in a way that it minimizes head movement.

- FCFS

Serve the track req. as per the arrival time.

- Shortest Seek Time First

From the current head position serve the track request which is nearest.

- SCAN (Elevator)

From its current pos. the R/W heads goes to the one ↑ end of surface servicing all the req. on its path. Then it reverses its direction & it goes to the other end, again servicing the requests in its path.

- C-SCAN.

From its current pos. the R/W head goes to the one end of the surface servicing all the req. on its path. Then it reverses its direction & goes to the other end

without servicing any req. From the other end it reverses its direction & starts servicing.

- LOOK

Same as SCAN but in one direction it only goes to last request & not the surface end.

- CLOOK

Same as CSCAN but in a direction it only goes to last request & not the surface end.

FLOATING POINT REPR

- This is used to store very large or very small number.

Ex. Say we have the following hex no.

$(897000000000)_H$

If we were to store this, it would take $12 \times 4 = 48$ bits. We can come up with a different representation of the same number.

$$+ 8.97 \times 16^{11}$$

So we can just store the values +, 8.97, 11 into a smaller sized register & later interpret it back.

- This representation is known as Mantisa - Base - Exponent Representation.

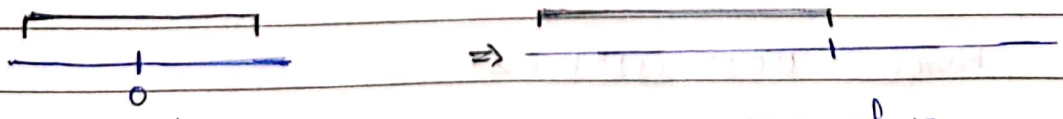
$$\pm K.M \times B^{\pm E}$$

This representation stores the following info:

Sign: 1 bit for +/v (0 for +ve, 1 for -ve).

Mantisa: Value after decimal point

Biased Exponent: Since our exponent can also be +ve or neg., we shift our exponent value by a bias towards pos. side.



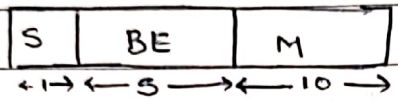
Original range of E

New range of E

$\therefore E' = E + \text{Bias}$

The bias value depends on the length of bias field in our representation.

Ex.



In the above repr. BE is of 5 bits meaning it can store values in range:

$$0 \text{ to } 2^5 - 1$$

So, since E' is of range 0 to $2^5 - 1$,
E would be in range $-(2^4 - 1)$ to (2^4)

$$\Rightarrow -15 \text{ to } 16$$

∴ For n bit Biased Exponent field, range of E Exponents is

$$-(2^{n-1} - 1) \text{ to } +2^{n-1}$$

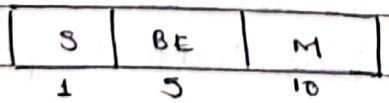
and the bias value is $(2^{n-1} - 1)$.

- Normalized Form: For a binary number, XYZ.ABC, the normalized form is X.YZABC, i.e only 1 bit to the left of decimal.

Ex. 1001.0101
$\Rightarrow 1.0010101 \times 2^3$

Ex. 0.0010101
$\Rightarrow 1.0101 \times 2^{-3}$

Ex. Format:



Num: 10010.11101×2^4

$\Rightarrow 1.0010111101 \times 2^8$

$\therefore M = 0010111010$ (10 bits)

$S = 0$ (+ve) (1 bit)

$E = 8 \quad E' = 8 + (2^{n-1} - 1)$

$\Rightarrow 8 + (16 - 1)$

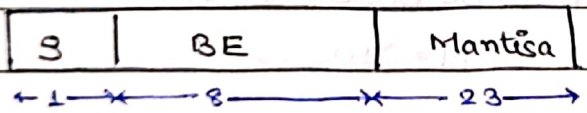
$\Rightarrow 23$

$\Rightarrow 10111_B = (5 \text{ bit})$

0	10111	0010111010

$= 27BA$

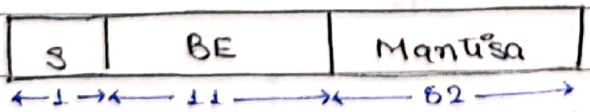
* IEEE-754 Single Precision Format (32-bits)



$\therefore$ Bias $= 2^7 - 1 = 127$

& Exponent Range $= -(2^7 - 1)$ to $+(2^7) = -127$ to 128

* IEEE-754 Double Precision Format (64-bits)



$\therefore$ Bias $= 2^{11-1} - 1 = 2^{10} - 1 = 1023$

& Exponent Range $= -(2^{10} - 1)$ to $+(2^{10}) = -1023$ to 1024

* Conversion To Value: Given S, BE & M.

$B = BE - Bias$ $\quad$ Value $= (-1)^S \times 1.M \times 2^B$

Ex. $R_1 = 0x42200000$ $R_2 = 0xC1200000$

$R_3 = \frac{R_1}{R_2}$ in Single Prec. format.

$R_1 \Rightarrow$ 0 100 0010 0010 0000 0000...

$\therefore S = 0$ $BE = 132$ $M = 0100...$

$E = 132 - 127$

$\Rightarrow 5$

$\therefore$ Value $= +1.01000.. \times 2^5$

$\Rightarrow +(101000)_2$

$\Rightarrow (+40)_{10}$

$R_2 \Rightarrow$ 1 100 0001 0010 0000 0000 0000...

$\therefore S = 1$ $BE = 130$ $M = 01000..$

$E = 3$

$\therefore$ Value $= -(1.0100) \times 2^3$

$\Rightarrow -(1010)_2$

$\Rightarrow (-10)_{10}$

$\therefore R_3 = \frac{R_1}{R_2} = \frac{+40}{-10} = -4.$

$R_3 \Rightarrow$ $-(100.0)$

$\Rightarrow -(1.00) \times 2^2$

$\therefore S = 1$ $E = 2$ $BE = 129$ $M = 000...$

1 100 0000 1 0000...

$\Rightarrow (C0800000)_H$

Ex. $(3E6D0000)_H$. Single Prec. Format.

$0b11\ 1110\ 0110\ 1101\ 0000\ 0000\ldots$

$S=0 \quad BE=124 \quad M=1101\ 1010\ 0000\ldots$

$E = 124 - 127 = -3.$

$+ (1.11011010) \times 2^{-3}$

$\Rightarrow +(0.001\ 11011010)_2$

$\Rightarrow$

* Special Values

Although the BE field in (say) single prec. format allows for values in range: -127 to 128 by biasing them (with 127) to 0 to 255.

0 is 0000 0000 & 255 is 11111111

These two values are used for special purposes, so the range usable is (-126 to 127).

S	BE	M	Remark
+/-	All 0s	All 0s	± 0
+/-	All 0s	Not All 0s	Denormalized Nums
+/-	All 1s	All 0s	± Infinity
+/-	All 1s	Not all 0s	NAN

* Implicit Normalization: 1. Mantisa format. 1 is implicit.
Explicit Normalization: 0.1 Mantisa. 1 is explicitly included in the mantisa.