* COMPUTER ORGANIZATION & ARCHITECTURE *

* Computer Architecture :-

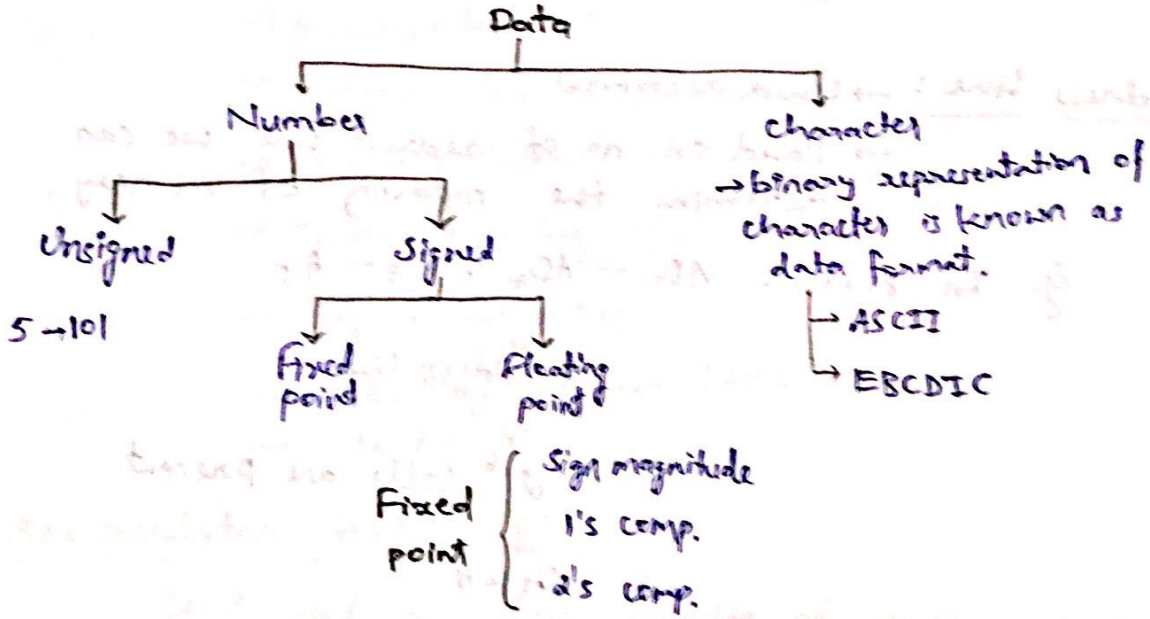
Conceptual design and fundamental operational structure of a computer is known as computer architecture.

→ CPU design, Instructions, Addressing mode, Data format

* Computer Organization :-

It deals with physical devices and their interconnection with a perspective of improving the performance.

→ I/O organization
→ Memory organization
→ performance improvement
  ↳ pipelining



* Component of computer

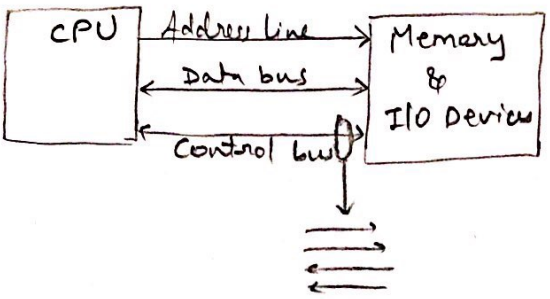
→ CPU ↳ control unit
  ↳ ALU

→ Memory ↳ Primary or Main memory
  ↳ Secondary Memory or Auxiliary

→ I/O ↳ Input
  ↳ Output

→ Other component → System buses
→ CPU register

System Bus : Collection of communication line within the computer system.



Address line : → Unidirectional
→ Based on no. of address line we can determine the capacity of memory.

Eg. In 8085 : $AD_0 - AD_7$, $A_8 - A_{15}$
⇓
16 Address line
⇓
2^{16} cells are present
⇓
64 KB

Data line : This lines are used to carry the binary sequence of between CPU memory and input output.
→ This lines are bidirectional.
→ Based on no. of data line we can determine word length of processor.

Eg. 8085 : $AD_0 - AD_7$, $A_8 - A_{15}$
↓
8-bit processor

Controll Lines : This lines are used to carry the control signals and timing signals.

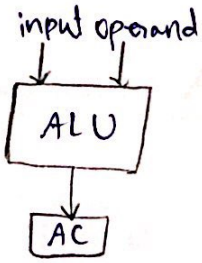
→ Control signals used to indicate the type of operation and timing signals are used to synchronise memory, and Input output operation with the processor clock.

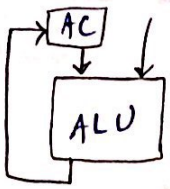
* Registers : Small memories within the CPU.

- → General purpose registers → Used to store general content like intermediate result.
- → Special purpose registers
  - → Accumulator (AC)
  - → Program counter (PC)
  - → stack pointer (SP)
  - → Instruction Reg. (IR)
  - → Flag or status Reg.
  - → Addressing Register (AR)
  - → Data Reg.

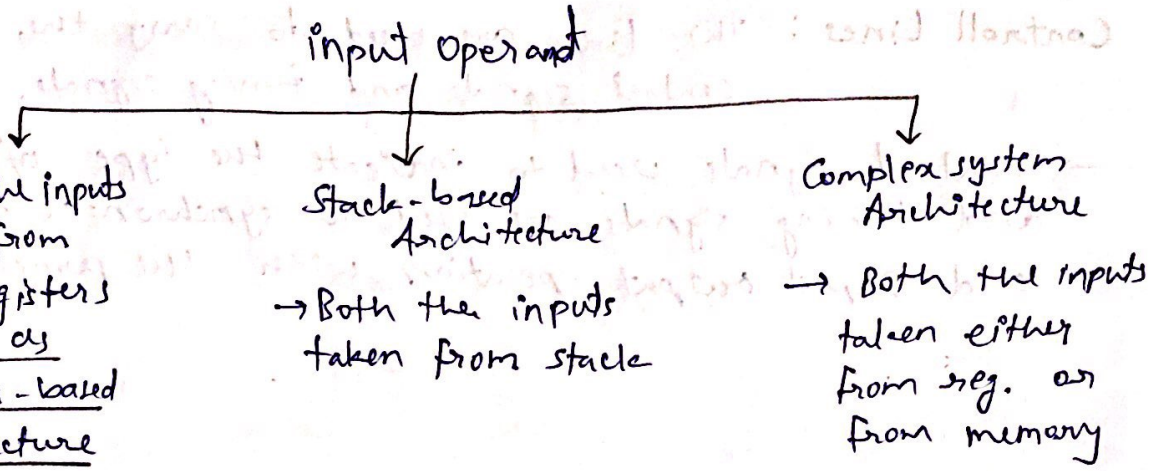
* Accumulator (AC)

It is used to store results of ALU and sometimes one of the ~~or~~ input operant for the ALU also.





Accumulator based Architecture

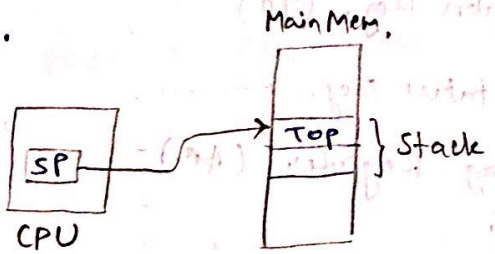


Program Counter (PC)

It is used to store the address of the next instruction to be executed

Stack Pointer Reg.

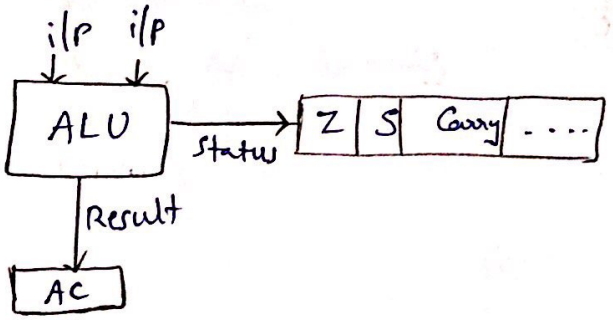
It is used to store address of top of processstack.



Instruction Register

It is used to store the current instructions which is being executed on CPU.

Flag or status Reg.



→ It is used to store the status of ALU result.

→ These flags are used to evaluate conditions.

eg. if (a>b) → a-b

Program status Word

- → only status register
- → AC + status register

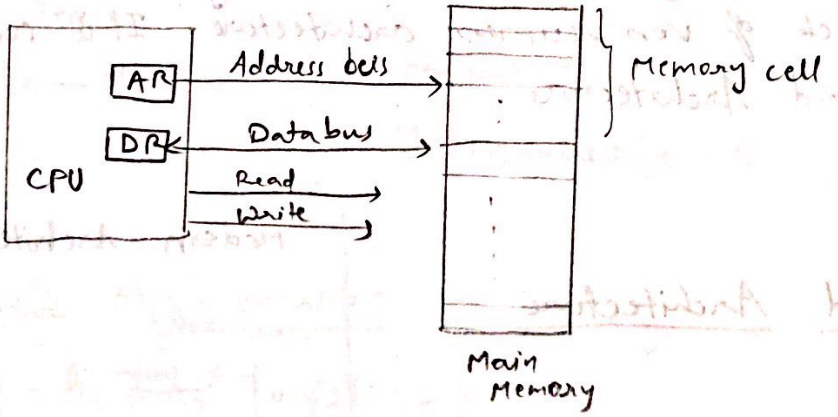
} depends on implementation

Address Register (AR) (Memory Address Register)

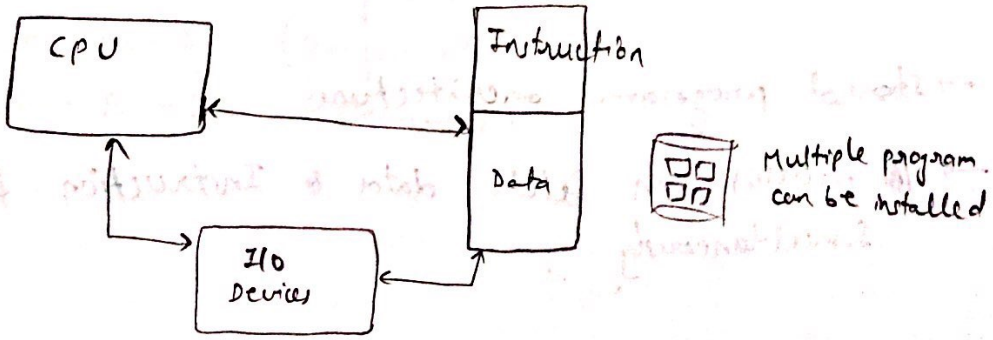
It is used to send address to memory.

Data Register (Memory Data Register) (Mem. Buffer Reg.)

It is used to send data to memory (Mem. write) and to receive data from memory (Mem. Read).



Von Neumann Architecture



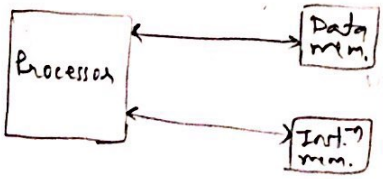
→ Afore von neuman architecture, there were fixed program computers, their function is very specific and they could not be programmed.

→ Stored Program Computers : These can be programmed to carry out many different tasks, applications are stored on them. This is introduced by Von Neumann.

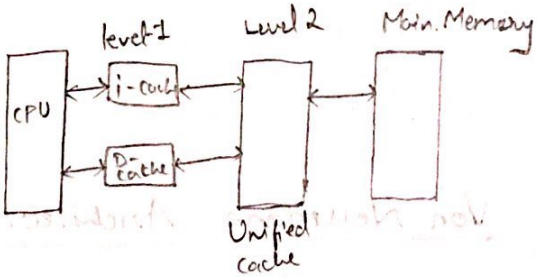
→ In Von Neumann architecture, programs and data are stored in a separate storage unit called memories and are treated the same.

→ Processor can not fetch instruction & data both simultaneously from the memory this is known as bottleneck of von neumann architecture. It is solved by Harvard Architecture

Harvard Architecture



Modern Architecture



→ stored program architecture

→ processor can fetch data & Instruction from memory simultaneously

Micro Operations : The operation performed on data stored in the registers.

→ CPU performance all the operation by performing one or multiple micro operations.

→ Atomic operation of processor.

Types of Micro Operation

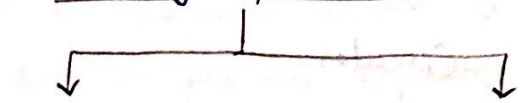
(i) Data Transfer / Register Transfer

(a) Register to Register

Eg. $R_1 \leftarrow R_2$

$R_2 \leftarrow R_1$

(b) Memory operations



Read

Eg. $R_1 \leftarrow M[Address]$

write

Memory ← CPU

Eg. $M[Address] \leftarrow R$

(ii) Arithmetic Microoperation

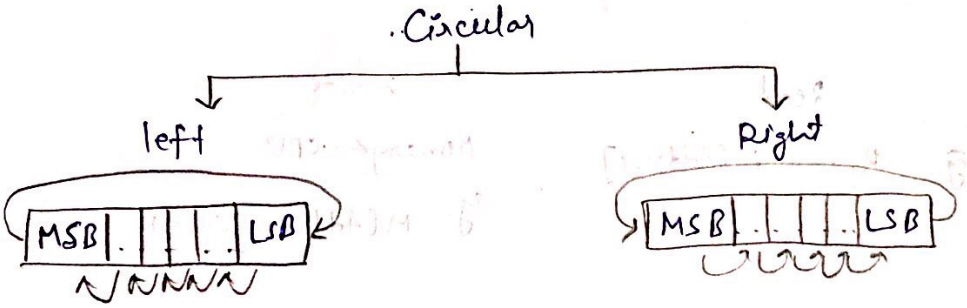
$R_1 \leftarrow R_2 + R_3$	(Addition)
$R_1 \leftarrow R_2 - R_3$	(Subtraction)
$R_1 \leftarrow \overline{R_2}$	(1's complement)
$R_1 \leftarrow \overline{R_2} + 1$	(2's complement)
$R_1 \leftarrow R_1 + 1$	(Increment)
$R_1 \leftarrow R_1 - 1$	(Decrement)

(iii) Logical Microoperation

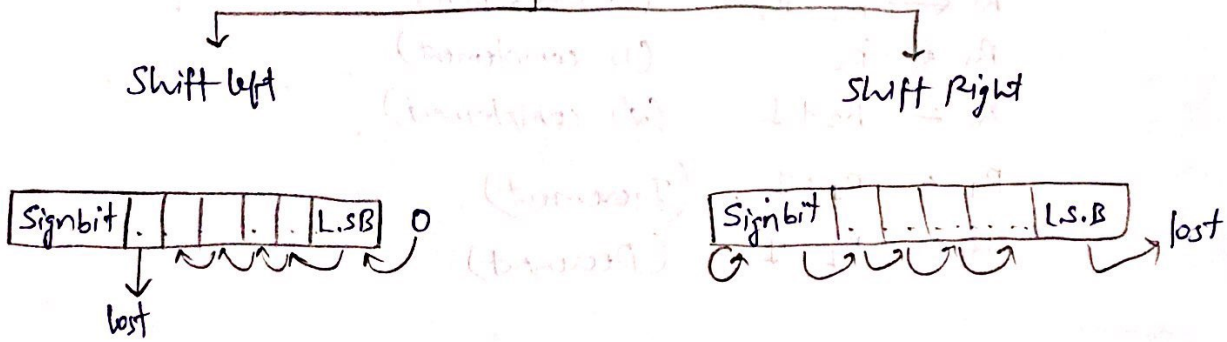
$R_1 \leftarrow R_2 \wedge R_3$	logical AND
$R_1 \leftarrow R_2 \vee R_3$	logical OR
$R_1 \leftarrow R_2 \oplus R_3$	logical XOR
$R_1 \leftarrow \overline{R_2 \oplus R_3}$	logical XNOR

(iv) Shift Microoperation

Logical → all bits are moving left or right along with sign bit

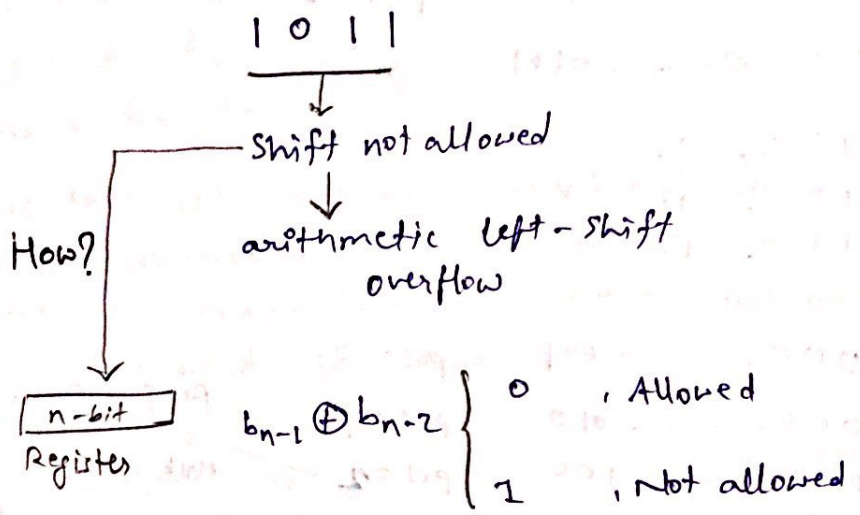


Arithmetic → All the bits are moving left or right except sign bit



→ perform only when CPU ensure that sign bit will not change

Eg. Arithmetic Shift Left



Q- Consider a new instruction named Branch on Bit set the instruction "bbs" reg, pos, label" jumps to label if bit in position pos of register operand reg is 1. A register is 32-bits wide & bits are numbered 0 to 31, bit in position 0 being the LSB. Consider the following immulation of this instruction on process that does not have bbs implemented.

temp ← reg & mask

Branch to label if temp is non-zero.

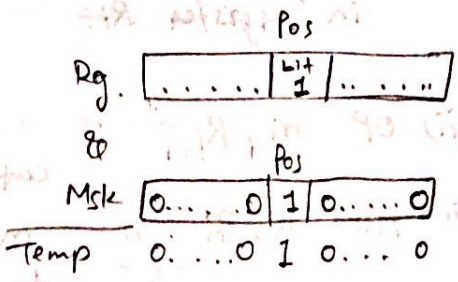
Variable temp is a temporary reg. for the correct immulation the variable mask must be generated by

(A) mask ← 0x1 <<pos

(B) mask ← 0x FFFFFFFF >> pos

(C) mask ← pos

(D) mask ← 0xF

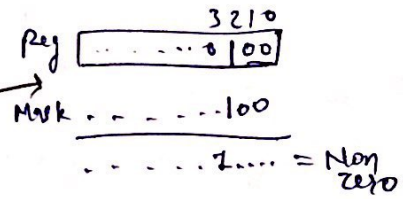


(D) Mask = 00....111 pos = 5 ✗

(C) Mask = 0.....0101 ✗

(B) 1111....11
0111...11 ✗
0011...11

(A) 000.... 01 pos = 0
000.....010 pos = 1
000.....100 pos = 2



Pg. 42

Q-3

M[3000] = 10
R3 = 2000 2001
2000 : 110
2001 : 110
2002 : 100
2010 : 100

R1 = 109
R2 = 110 119

M =

M[3000] = 10
R3 = 2000 2001
2000 = 110
2001 = 109
2002 = 100
2003 = 100
2010 = 100

R1 = 1
R2 = 100 100 109

Q- In a simplified computer the instructions are

(i) OP Rj, Ri performs Rj op Ri and stores the result in register Ri.

(ii) OP M, Ri : val. OP Ri & stores result in Ri (content of M)

(iii) MOV m, Ri : moves the content of memory location m to register Ri.

(iv) MOV Ri, m : moves Ri to m.

The computer has only two registers and OP is either add or sub. Consider the following basic block.

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - t_2$

$t_4 = t_1 - t_3$

Assume that all operands are initialy in memory. the final value of computation should be in the memory. what is the minimum no. of move instruction in the code generated for basic blocks.

→

```
MOV  M[1000], R1
ADD  M[2000], R1
MOV  M[3000], R2
ADD  M[4000], R2
SUB  M[5000], R2
SUB  R1, R2
MOV  R2, M[6000]
```

```
MOV  M[1000], R1
ADD  M[2000], R1
MOV  M[3000], R2
ADD  M[4000], R2
SUB  M[5000], R2
SUB  R1, R2
MOV  R2, M
```

minimum no. of moves inst^n = 3

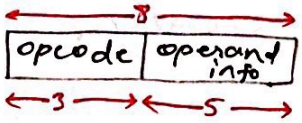
* Instruction

A group of bits which instructs computer to perform some operation.

Eg. Add a, b
(Add: opcode)

Instruction

Example: For a CPU, instruction is of 8-bits.



Max. operation possible = $2^3 = 8$

or

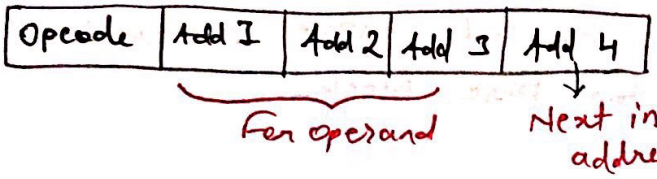
Max. type of instructions supported by CPU

```
000  - ADD
001  - SUB
010  - MOV
011  - :
100  - :
101  - :
110  - :
111  - :
```

⇒ Types of instruction

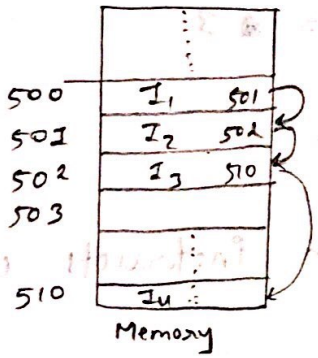
4- address instruction

Maximum four addresses can be specified within an instruction.



Such instructions are used in those computer which do not have program counter.

→ All processor support the program counter register as a mendatory register, therefore this format is not used.

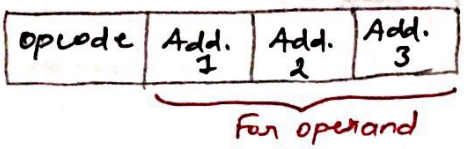


(i) Larger sized instructions, hence more memory is required to store the program.

(ii) If program is relocated then the fourth address part of each instruction should be updated and updation of instruction is costly operation.

3-address Instruction

Maximum three addresses can be specified within an instruction.



→ Those computer which are using 3-add. instruction have to use program counter.

Example : Add r_0, r_1, r_2

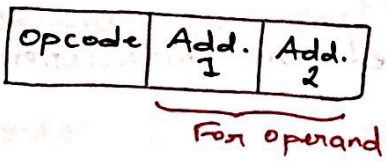
→ This 3-add. instruction format is used in modern computer system.

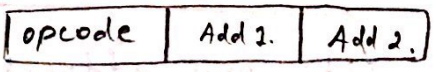
→ Smaller size compared to 4-add. inst.^{n}, hence required less memory to store program.

→ If relocation is required then no need to update instructions.

2-address Instruction

Max. Two addresses can be specified in instruction.





Source	Destination
reg.	reg.
reg.	Mem.
Mem.	reg.

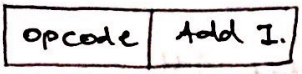
→ One of the operand is use as source & destination both.

(i) the value of common operand is overwritten by the result of operation

(ii) More no of instruction required for program as compared to 3-add. inst^n.

1- address Instruction

Max. one address can be possible within instruction



→ Accumulator is used as 2nd operand implicitly.

Example : ADD R_1

↓

$AC \leftarrow AC + R_1$

1) Load M[5000]
2) STORE M[4000]

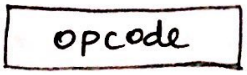
☹

(i) More number of instructions for program as compare to 2-address instruction.

→ 1-add. Instruction is used in Accumulator based Arch.

0- address instruction

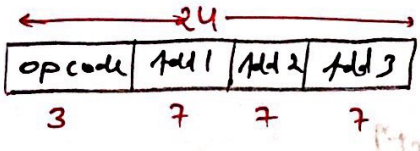
No any address specified in instruction.



→ 0- address instruction is used in stack based arch.

Q- Digital computer has 3 Byte instruction, 7 bit address. Min & Max 3-address inst^n. supported by this system.

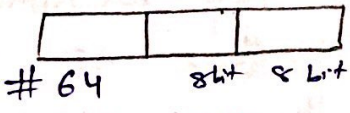
→



Max. $= 2^3 = 8$

Min. $= 1$

Q- Computer supports 64 2-add. inst^n in which each add. is of 8-bit. Size of an inst^n is



$\therefore 2^6$

$\therefore 6 + 8 + 8 = 22$ bit $\cong$ 3 Byte

Q- In above Question if we assume that inst^n are stored in memory in Byte align fasion, the ammount of memory required to store the program stude- with contains 200 inst^n is

→

~~$22 \times 200 = \frac{4400}{8} = 550$ Byte~~

For 1st inst^n 22 bit $\cong$ 3 Byte

$\therefore$ 200 inst^n $\longrightarrow 200 \times 3$

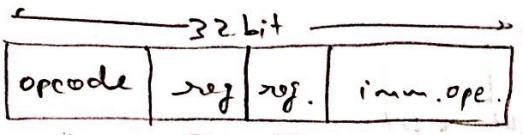
$= 600$ Byte

Instruction Set Architecture

Collection of all the instruction supported by CPU is known as inst.^{n} set. Arch. of that CPU.

Q-17 40 distinct inst^n.
24 gen. reg.
32-bit inst^n.
opcode, 2 reg. operand and an immediate operand.
No. of bits available for immediate operand field is.

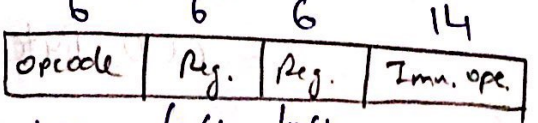
→



32-16 = 16 bit

#inst^n 40 reg.=24 reg=24
$\log_2 40$ $\log 24$ $\log 24$
= 6 + 5 + 5
= 16

Q-45 32-bit inst^n. with 1 word long inst^n.
- 64-registers each of 32 bit long.
45 inst^n have immediate operand in addition to two register operands.
Assume immediate operand is an unsigned integer max. value of the immediate operand is.

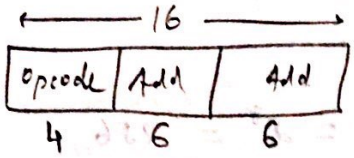


log 45 log 64 log 64
= 6 = 6 = 6

∴ Max. value of Immediate operand is $2^{14} = 16384 - 1 = 16383$

Q-16 CPU used expanding op-code. If has 16 bit inst^n with 6-bit add. It support one add & 2 add. inst^n only. If there are 'n' two ~~app~~ add. inst^n the max. no. of one add. inst^n is

→

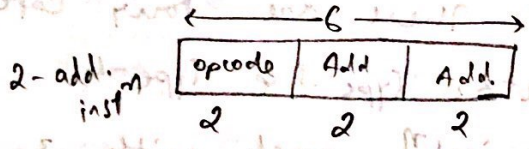


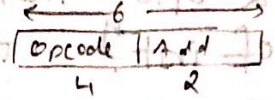
2 add. inst^n = 2^4

~~No add~~ → # 1 add. inst^n = $(2^4 - n) \times 2^6$

Q- A digital computer has 6-bit instructions and 2-bit Add. It supports 2-add. & 1-add. inst^s both. If there 3 2-add. inst. then Min & max. how many 1-add. inst^n it can support.

→





1-add. inst^n

2-add inst^n = $2^2 = 4$

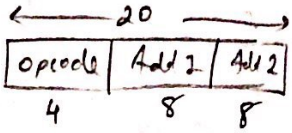
~~# 1-add~~ # 1-add. inst^n = $(4-3) \times 2^2$

= 4

∴ Min = 1

∴ Max = 4

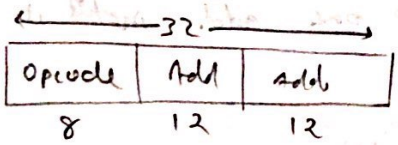
Q- consider a computer which have 20-bit inst^n & 8-bit add. if there are 12 2-add. inst^n then max. how many 1-add. inst^n possible.



2-add $2^4 = 16$

1-add = 16 - 12 = 4×2^8 ∴ Max = 1024

Q.- Digital computes has 32-bit inst^n & 12-bit add. If there are 254 2-add. inst^n & 8000 1-add. inst^n then Max. how many 0-add. inst^n. possible



2-add. inst^n $= 2^8 = 256$

1-add inst^n $= 256 - 254$

$= 2 \times 2^{12} = 2^{13}$

0-add inst^n $=$ ~~16384 - 8000~~ $8192 - 8000$

$=$ ~~384×2^{12}~~

$= 192 \times 2^{12}$

Q Processor has 16 integer registers ($R_0, R_1 \ldots R_{15}$) & 64 floating point registers ($F_0, F_1 \ldots F_{63}$). It uses 2-Byte inst^n format. There are four categories of inst^n. Type-1, Type-2, Type-3 & Type-4.

Type-1 consists of 4 inst^{n}s. each with 3 ~~reg~~ integer register operands. (3Rs)

Type-2 category consists of 8 inst^{n}s each with 2-floating point register operand (2Fs)

Type-3 consists of 14 inst.^{n}s each with one integer reg. operand and 1-floating point reg. operand (1R + 1F)

Type-4 consists of N inst.^{n}s each with a floating point reg. operand (1F).

The max value of N is.

Type 1 $= 4 \times 2^4 \times 2^4 \times 2^4 = 2^{14}$

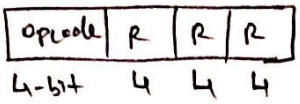
Type 2 $= 8 \times 2^6 \times 2^4 = 2^{15}$

Type 3 $= 14 \times 2^4 \times 2^6 = 7 \times 2^{11}$

→

Total instruction = 2^{16}

Type 1 : (3 Rs)



#inst. = 4

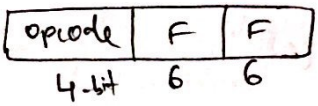
∴ ~~4 * 4 * 4 * 4~~

Max = $2^4 = 16$

Used = 4

Unused = 12

Type 2 : (2 Fs)



~~Unused = 4~~ unused

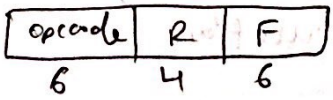
Max = $12 * 2^0$

~~Used~~ = 12

Used = 8

Unused = 4

Type 3 : (1R + 1F)



Unused = 4

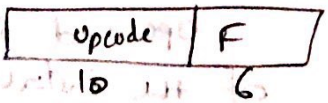
Max = $4 * 2^2$

= 16

Used = 14

Unused = 2

Type 4 : (1F)



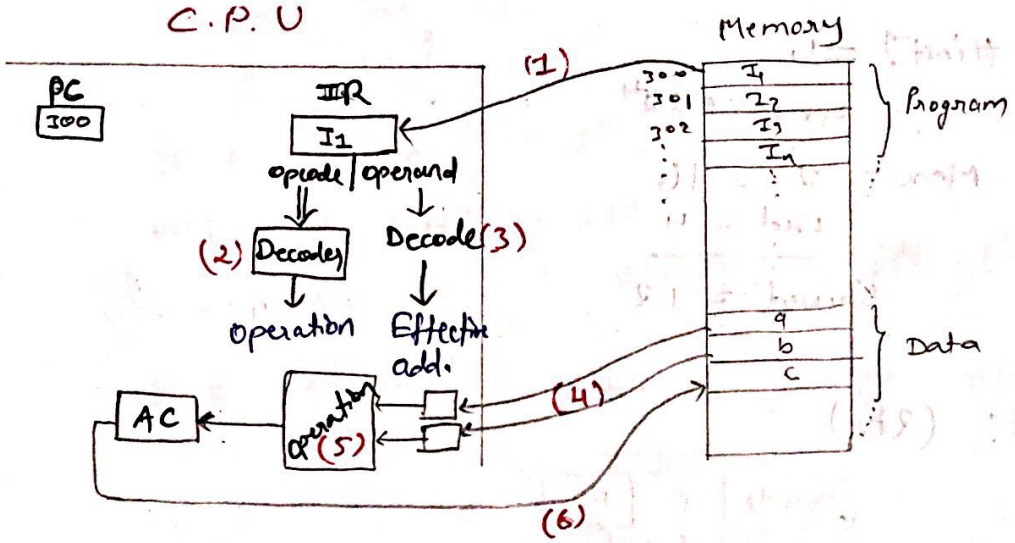
Unused = 2

Max = 2×2^4

= 32

* Instruction Cycle

Six phases are required to execute the instruction.



(1) **Instruction Fetch**: Using the program counter value the next instruction is copied from memory to Instruction Register. During the IF only, the value of PC is incremented by the size of instruction.

(2) **Instruction Decode**: CPU decodes the opcode part of instruction to obtained the instruction.

(3) **Effective Address Calculation**: CPU decodes the operand information part of the instruction to obtained the effective address.

Effective Address : Address of operand in computation type instruction and target address in branch type instruction is effective address.

(4) Operand Fetch : The operands are fetched from the source using the effective address.

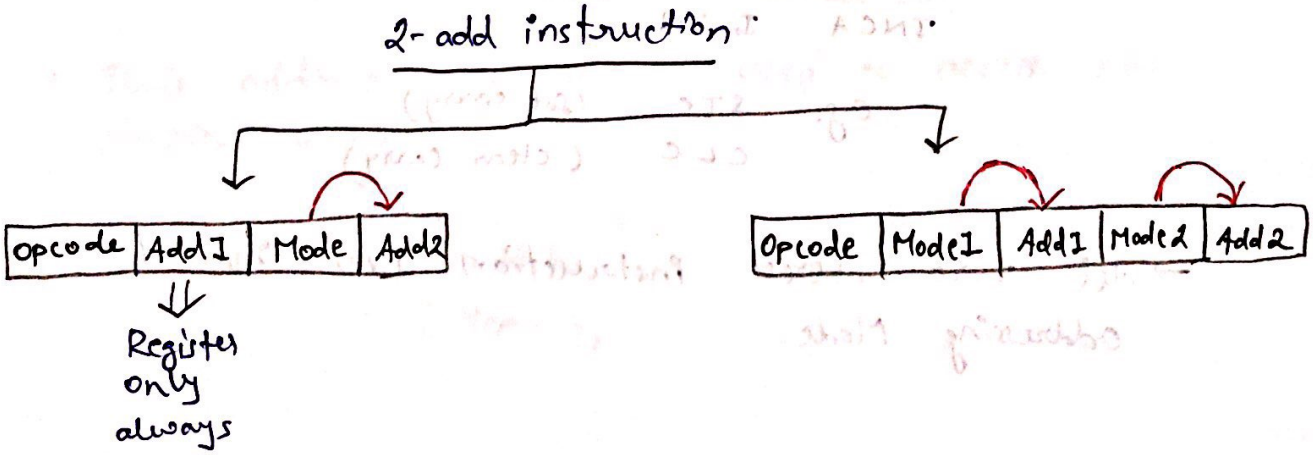
(5) Execution : CPU performs the operation and generates the result.

(6) Write Back : The generated result is copied back to the destination.

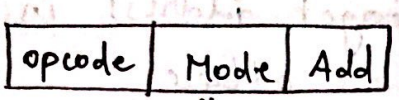
NOTE: All type of instructions do not required all six phases.

* Addressing Mode:

It specifies how and from where the operand is obtained using the address ~~bit~~ field of instructions.



1-add. inst



Ex. 00 ⇒ operand value
01 ⇒ Register reference
10 ⇒ Memory Address

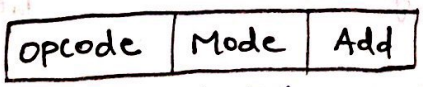
→ Addressing mode helps to obtained the Effective addresses.

* Types of Addressing Mode

(1) Implied Mode (or Implicite): The opcode definition itself specifies the operand. hence no need to specify any explicite address for the operand.

→ There is no effective address in implied mode.

Ex:

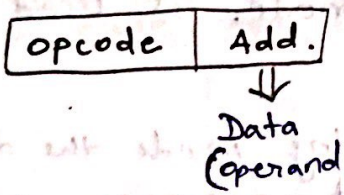


e.g. STC (set carry)
CLC (clear carry)

→ All zero address instruction use implied addressing Mode.

(ii) Immediate AM: The address field of instructions specifies the operand value.

→ This mode is used to initialized register with constant value.

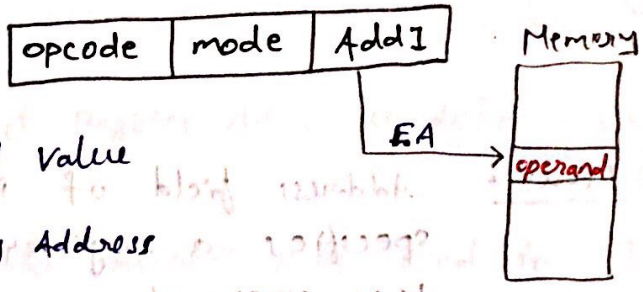


Eg. ADDI R0 23H : R0 ← R0 + 23H

→ This addressing mode is used to access the constant.

(iii) Direct / Absolute AM:

The address field of instruction specifies the effective address.



E.A = Add. field value
⇓
Memory Address

→ This addressing mode is used to access the static variable.

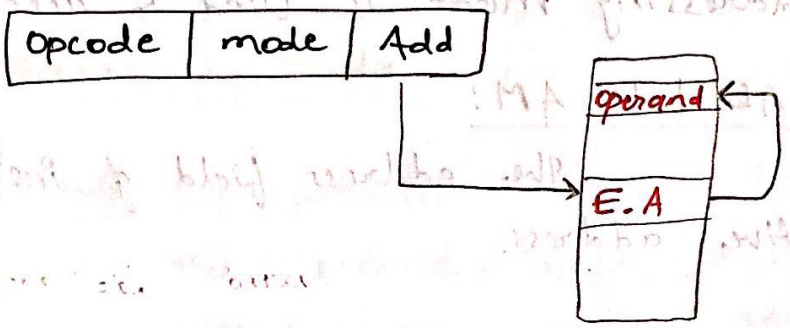
Data = [E.A]
= [Memory]

(iv) Indirect Addressing Mode:

The address field of instruction specifies the address of E.A.

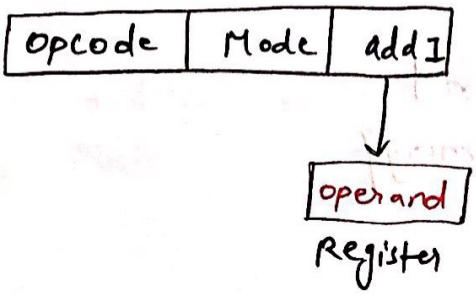
→ This addressing mode is used to implement the pointer.

→ In this addressing mode the data is present in the memory, that memory address may be available in register or memory.



(v) Register Mode: Address field of instruction specifies a register which holds the operand.

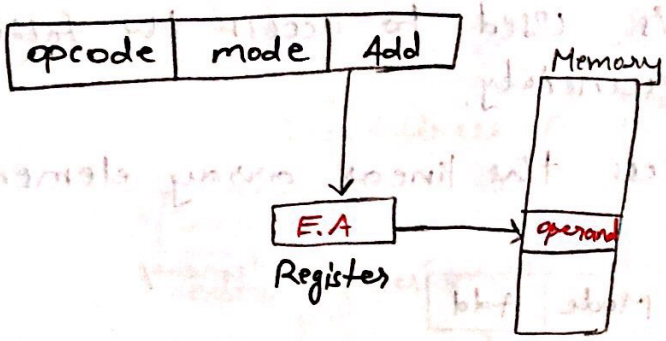
→ E.A is always the register name.



→ This mode is used to access the local variable.

(vi) **Register Indirect** : The address field of instruction specifies a register which holds effective address.

→ This mode is used to shorten the instruction length.



How ?

4-GB RAM = Memory address = 32-bit
and
64 registers (G.P.Rs) = Reg. reference = 6-bit

For Direct ~~Register~~ A.M = | opcode (x) | Mode (y) | Add (32) | (x+y+32) bit

For Indirect Register A.M = | opcode (x) | Mode (y) | Add (6) | (x+y+6) bit

Hence, Register Indirect A.M used to shorten the instruction length

Access time ?

Time to access operand in ~~Register mode~~ =
Direct Addressing mode = 1 memory access time

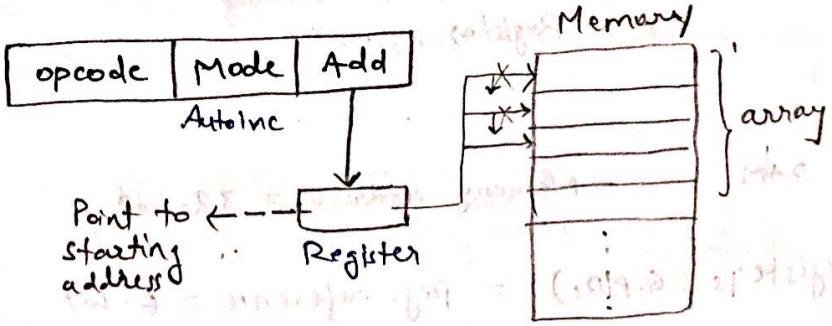
Time to access operand
in Indirect register A.M = 1 reg. access time + 1 Mem. access time
$\cong$ 1 memory access time

(vii) Autoincrement / Autodecrement

It is a variant of register indirect mode in which the content of register (EA) is automatically increment or decrement.

→ This mode is used to access the table of content (Array) sequencially.

→ Used to access the linear array element.



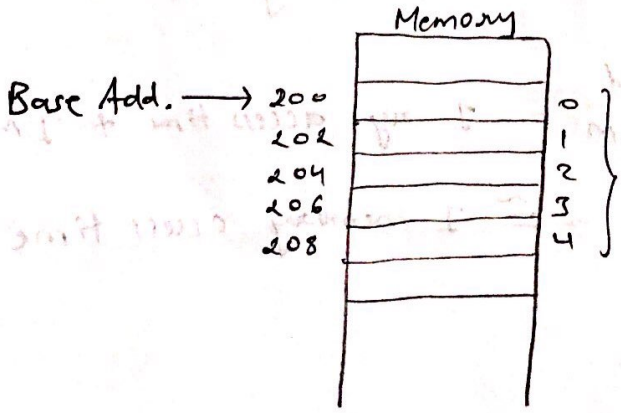
→ E.A can be calculated either by increasing or decreasing the step size with the base address.

Autoincrement ⇒ post inc.
Autodecrement ⇒ Pre dec.

(viii) Index Addressing Mode or Index Reg. Mode

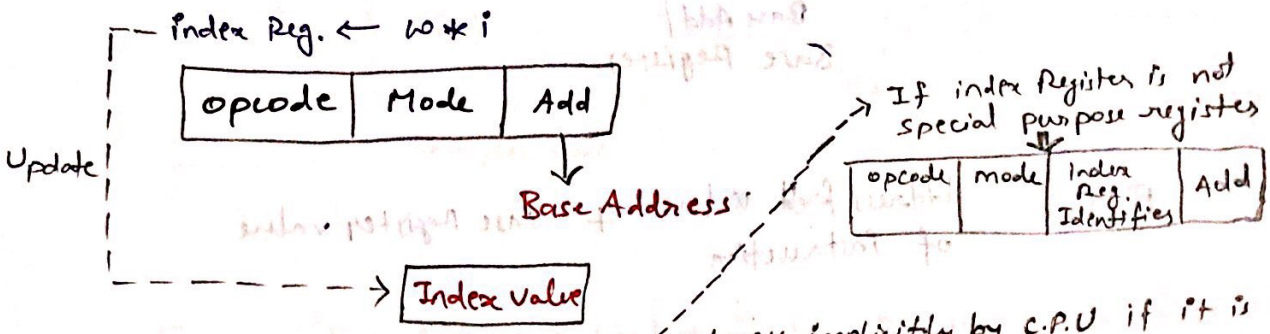
This addressing mode is used to implement or access the array.

→ ~~This AM is also called based indexed addr. mode~~



address (A[i]) = Base Address + (w * i)

w * i: w → Size of an element; i → Index Value



Index Register → Access implicitly by C.P.U if it is special purpose register.

$$\therefore E.A = [\text{address field value of instruction}] + [\text{Index Register value}]$$

$$= \text{Base Address} + \text{Index value}$$

$$\therefore \text{Data} = [E.A]$$

$$= [\text{Base. Add.} + \underbrace{[\text{Index Rg.}]}_{\text{1 Reg. Ref.}}]$$

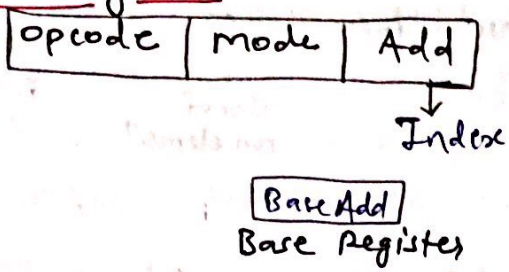
1 Arithmetic computation

1 memory reference

NOTE: If the program data (array) is relocated then the new base address should be updated in the address part of the instruction. Updation of instruction is costly operation.

To solve this problem we have Base Register Add. Mode

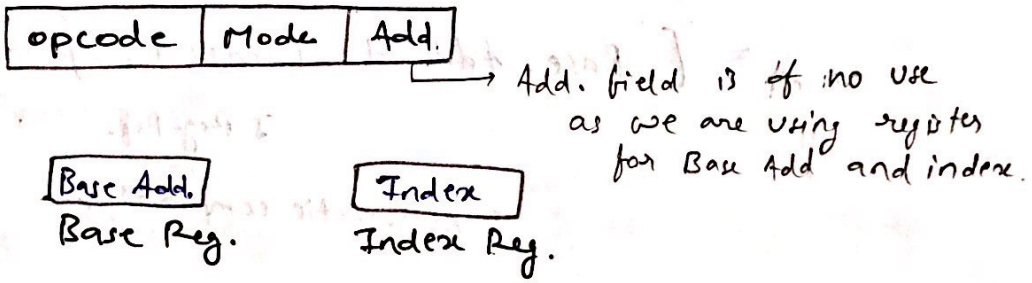
(IX) Base Addressing Mode



E.A = Address field value of instruction + Base Register value

NOTE: If program data (array) is relocated then the new Base address should be updated in the Base register, hence no need to update the instruction

(X) Base + Index Register A.M:



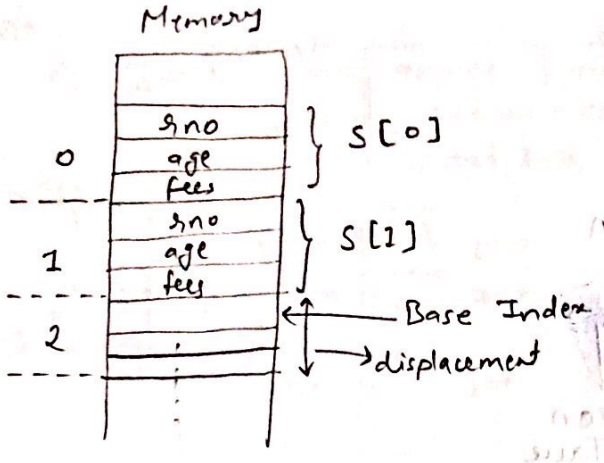
(XI) Base + Index Reg. + Desplacement Mode

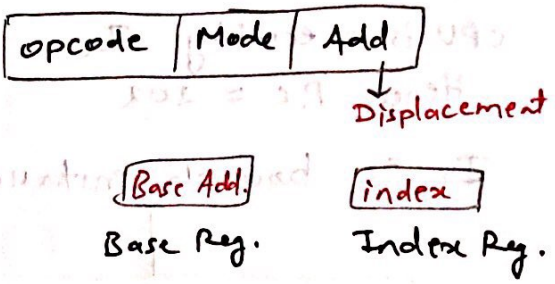
This mode is used to access array of records (structures).

Example:

```
struct student
{
  int rollno;
  int age;
  float fee;
};
```

```
struct Student S[10];
```

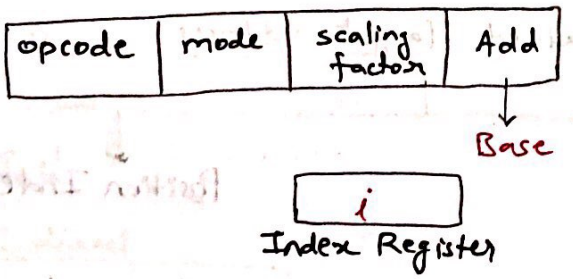




E.A = [Base Register value] + [Index Register value] + [Add. field value of inst^{n}]

= Base Address + Index + Displacement

(Xii) Scaled Mode:



→ Perform in 1 cycle while index add mode required 2 cycle

inst^{n} : index reg. ← 2 * i

inst^{n} : Base + index

E.A = Address field of Instruction + Scaling factor * Index Register value

(Xiii) PC-Relative Mode / position independent Mode

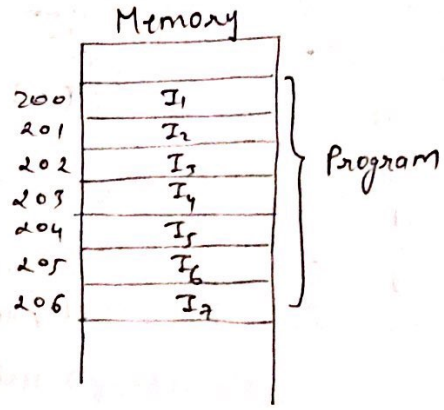
This mode is used for branch type of instructions.

Let,

CPU is executing I_2

Hence P.C = 202

I_2 is branch instruction



Condition false

⇓

Next instruction in the sequence is executed i.e., I_3

→ No change in P.C

Condition True

⇓

Next instruction executed is target instruction (Let I7)

∴ Target Add. = 206

→ We have to change P.C value

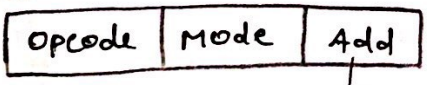
PC = Target Address

Obtained Target Address

Position Dependent

Target Address

Position Independent



Relative no. of location to skip

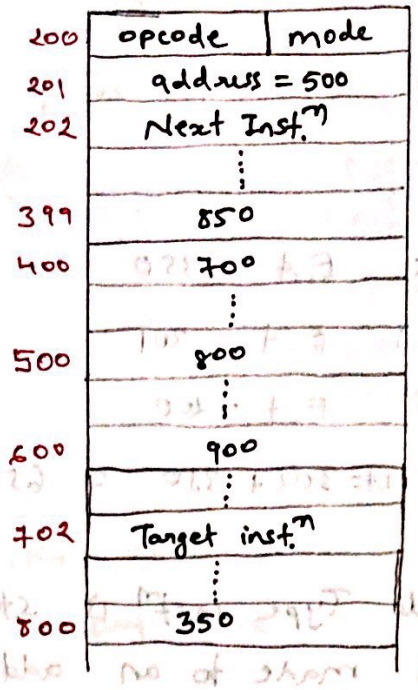
Target Address = PC value + Relative Location

Example :

PC ~~200~~ 202

Reg. 400
399

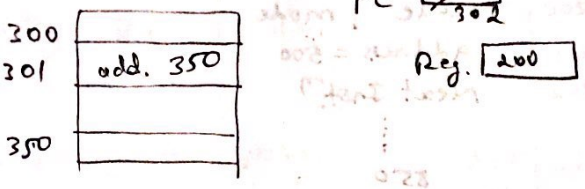
XR 100
= Index Reg.



Addressing Modes	Effective Add.	Operand
1. Immediate	201	500
2. Direct Mode	500	800
3. Indirect Mode	800	350
4. Register Mode	-	400
5. Register Indirect	400	700
6. Auto decrement	399	850
7. Index Mode	500 + 100 = 600	900
8. PC-relative	202 + 500 = 702	-

Q- An Instn is stored at location 300 where its add. field is at location 301. The add. field value has 350. A processor reg. contains the no. 200 evaluate the E.A if the add. mode is

(i) Direct Mode
(ii) Immediate Mode
(iii) Reg. Indirect
(iv) PC-relative



(i) Direct Mode : E.A = 350

(ii) Immediate Mode : E.A = 301

(iii) Reg. Indirect : E.A = 200

(iv) PC- relative : EA = 302 + 350 = 652

Q. Relative Branch Mode Type inst^n is stored at Memory 300. The Branch is made to an add. 450.

(i) What should be the value of relative add. field of the inst.^{n}

(ii) Determine the value of PC before inst.^{n} fetch, after inst^n fetch & after Execution phase of inst.^{n}

→ P.C 300 / 301

(i) ~~450 = 300 + x~~ ~~∴ 450 - 300 = 149~~

(ii) P.C before inst.^{n} fetch = 300
P.C after inst.^{n} fetch = 301
P.C after ~~inst.^{n}~~ Execute = 450

(i) E.A = PC + Displacement (Relative add. field value)

450 = 301 + x

∴ 450 − 301 = x

∴ 149 = x

Q-12 For computers based on 3-add. inst^n formate, each add. field can be used to specify

S1 : Memory operand ✓

S2 : Processor register ✓

S3 : An implied accumulator reg. → X

Any special purpose reg. will not be access using explicitly, it can be access implicitely only.

Instruction Cycle

- Fetch Cycle
  - Inst.ⁿ fetch
- Execution Cycle
  - Decode
  - E.A calculation
  - Operand fetch
  - Execution
  - Write back

Q-10 ADD A [R₀], @B

No. of Memory cycle

→ $M[A+[R_0]] \leftarrow M[A+[R_0]] + M[M[B]]$

1 mem cycle — 1 mem cycle — 2 Mem. cycle

Total = 4 Memory cycle.

* CPU Design

CPU cycle time: The amount of time in which CPU can perform one micro operation

Cycle Per Instⁿ (CPI): No. of CPU cycle required to execute an instruction.

$$\text{Clock Rate} = \frac{1}{\text{CPU cycle time}}$$

$$\text{CPU time} = \text{Execution Time (E.T)} = \#\text{Instⁿ} * \text{CPI} * \text{cycle time}$$

$$\text{E.T} = \frac{\#\text{instⁿ} * \text{CPI}}{\text{clock rate}}$$

→ Program contain different instruction and different instruction required different cycle therefore program execution time is

$$\text{CPU time} = \sum IC_i * CPI_i * CT$$

i is types of inst.ⁿ

Inst.ⁿ	CPI	no. of inst.ⁿ	
Load & store	6	40	240
ALU	4	20	80
Branch	5	30	150
other	4	10	40
		Total = 100	Total = 510

$$\therefore CPI_{avg} = \frac{510}{100} = 5.1$$

$$\therefore CPI_{avg} = \frac{\sum_{i=1}^{m} CPI_i * n_i}{\sum_{i=1}^{m} n_i}$$

Performance of C.P.U

Performance of C.P.U is given by MIPS (Million Instⁿ per Sec.)

$$MIPS = \frac{\#inst.^n}{\text{Execution time} * 10^6}$$

$$MIPS = \frac{\#inst^n}{IC * CPI * \text{Cycle time} * 10^6} = \frac{\text{Clock rate}}{CPI_{avg} * 10^6}$$

→ Some times MIPS may provide the wrong result because it does not include (consider) the capability and complexity of instruction.

Q- Consider two processor P_1 and P_2 executing same inst^n set. Assume that under identical condition for same input. A program running on P_2 takes 25% less time but incurs 20% more CPI. compared to program running on P_1. If the clock freq. of P_1 is 1 GHz. then what is the clock freq. of P_2.

→

$$1 = \frac{IC * CPI * \text{GHz}}{1\ GHz}$$

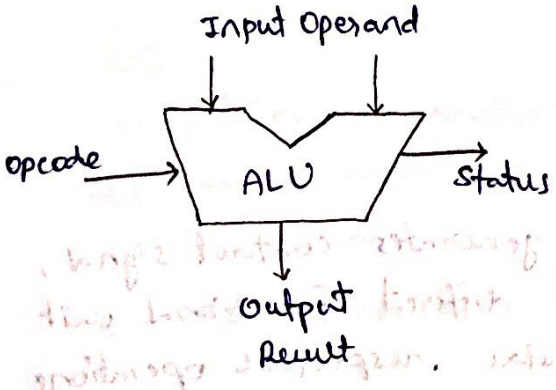
$$0.75 = \frac{IC * 1.2\ CPI * \text{clock rate}}{\text{clock rate}}$$

$$\frac{1 \times 1\ GHz}{CPI} = \frac{0.75 \times \text{clock rate}}{1.2\ CPI}$$

$$\frac{1\ GHz \times 1.2}{0.75} = \text{Clock rate}$$

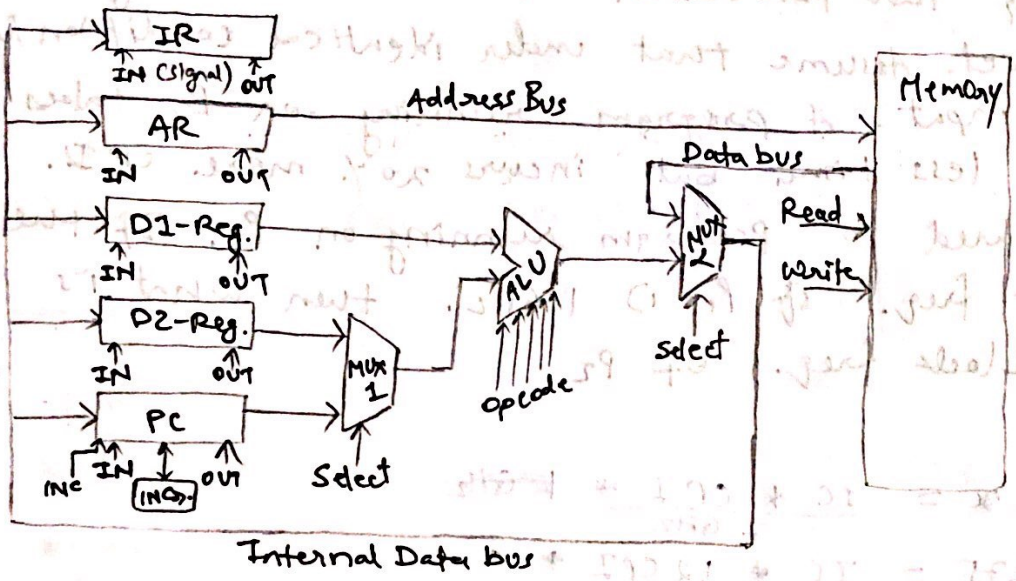
$$1.6\ GHz = \text{clock rate}$$

* ALU (Arithmetic & Logical Unit)



→ provide arithmetic & logic operation.

Data Path collection of functional unit ALU, mux etc. to perform dataprocessing operation is known as data path



Instruction fetch

IR ← M[PC] --> Memory can not be access directly, AR can access memory. So we have to copy PC value to AR & AR can access memory.

↓

1. AR ← PC
2. IR ← M[AR] , PC ← PC+1

These opⁿ should be perform in sequence if both are perform simultaneously then over internal data bus carrieng data and, IN signal of IR and AR both enable which leads to inconsistency in IR and AR

These both operation can be perform simultaneously as they are mutualy exclusive

Control Unit

It is part of CPU which generates control signal, and sends those signals to different different Functional unit so that the units can perform their respective operations.

Control Variable: The name of the control signal.

Ex. Read, Write

Control Word: Collection of all control signals generated by the control unit

IR		AR		DR		DR		PC			MUX 1	MUX 2	ALU	Memory
IN	OUT	IN	OUT	IN	OUT	IN	OUT	IN	OUT	INC	Selc.	Selc.	11111	Read/Write

AR ← PC

0 0 1 0 0 0 0 0 0 1 0 1 1 10101 0 0 ← control word

IR ← M[AR], PC ← PC+1

1 0 0 1 0 0 0 0 0 0 1 x 0 xxxxx 1 0

Control Unit Organization

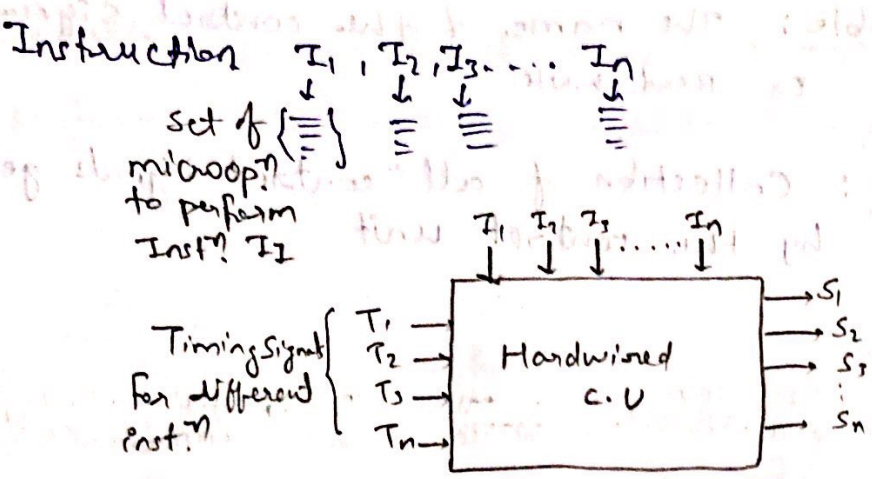
- Hardwired Control Unit
- Microprogrammed Control Unit

Hardwired C.U: The control logic is implemented using flipflop, logic gates, decoder & other digital circuit.

☺ faster mode of operation (faster control unit)

☹
- → Updation in control logic is difficult because rearranging the ~~wire~~ wire among circuits is difficult
- → It cannot handle complex inst.n as circuit is complex
- → More costlier

→ RISC ⇒ hardwired C.U.



at T_1, To execute I_1 ∧ T_1 will enable these set of signals

	I_1	I_2	I_3 ...	I_n
T_1	$S_1 S_2 S_4$	...	...	...
T_2	$S_2 S_3$	...	...	...
T_3	...	..	...	...
⋮				
T_n	...	...	...	...

Microprogrammed control Unit

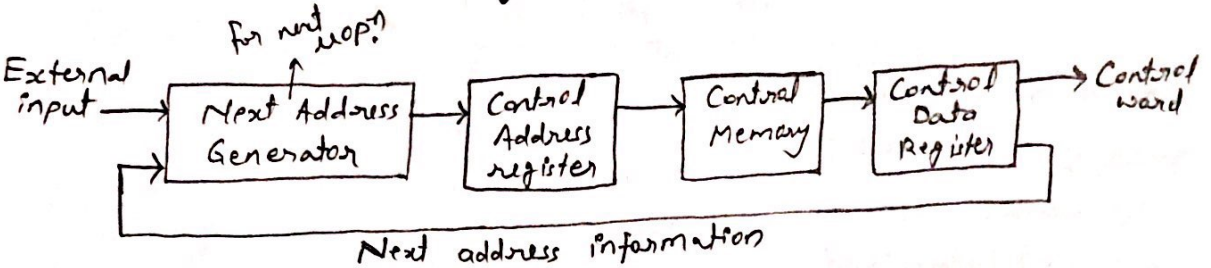
All possible control words are stored in a memory (control memory) and based on the requirment, the control word is read from the memory and the signals are sent to the respective units.

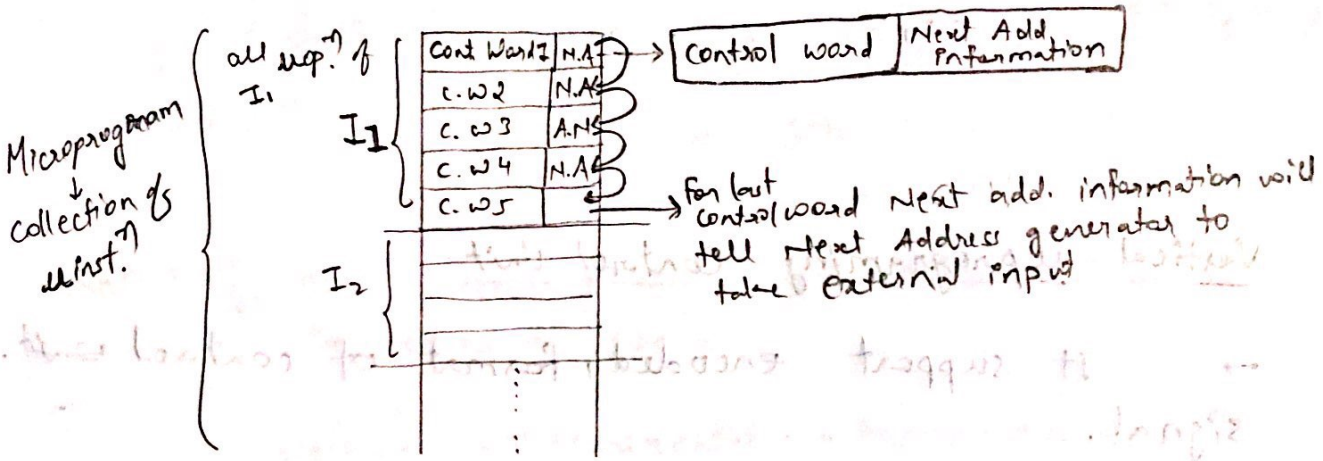
(i) Updation in control logic is easy
(ii) Can handle complex Inst^n

(i) slower than hardwired control unit as memory access is slower

→ Used in CISC (complex inst^n set comp.)

Control word Sequencing.





Types of uprog. control Unit

- Horizontal
- Vertical

Horizontal uprog. Control Unit

< Decoded format of control signal >

→ It support decoded format of control signal means 1 bit / control signal.

Eg. If processor supports N control signal then this format required N bits in control field.

→ It supports longer control word.

→ There is no need of external decoder to generate the control signal. Therefore it is faster than the vertical uprog. control unit.

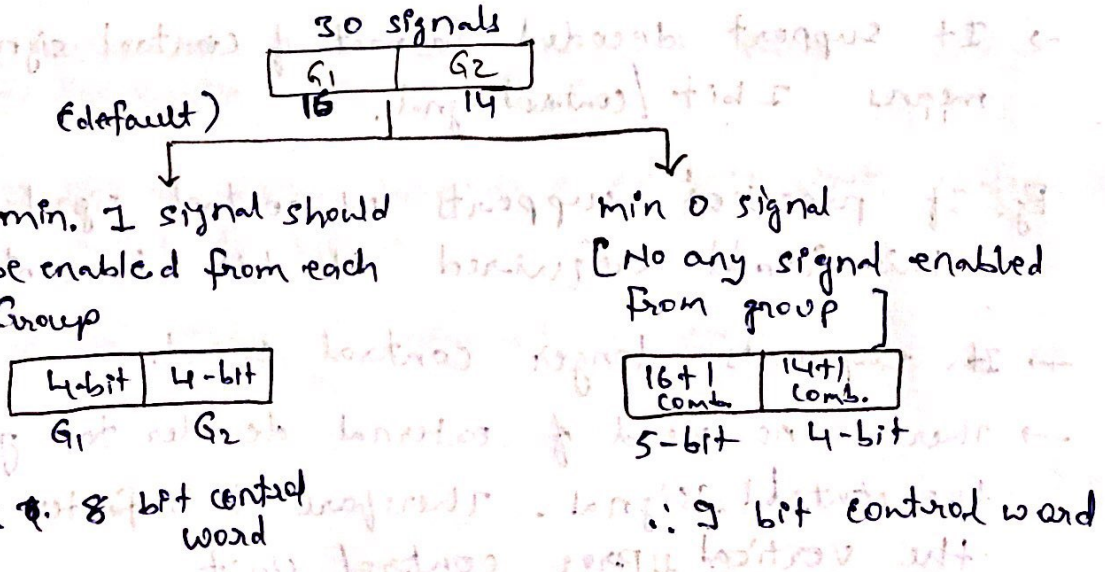
Vertical uprograming Control Unit

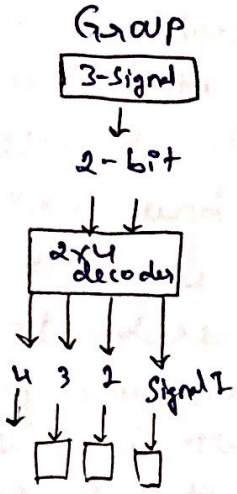
→ It support encoded, format of control unit. signal.

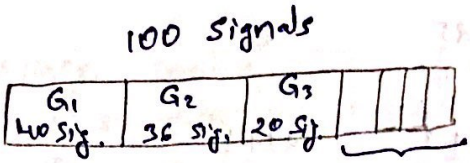
→ 'N' encoded control signal requires $\log_2 N$ bits.

→ It support shorter control word.

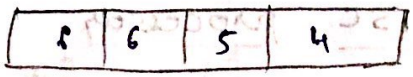
→ There is need of external decoder to generate the control signal. therfore, it is slower than horizontal.







4-signal can not be kept in any group hence stored like horizontal (2-bit for each signal)



∴ 21 - bits

NOTE: (1) The ascending order of control unit design in terms of speed is

Vertical → Horizontal → Hardwire

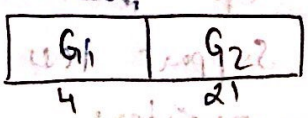
low ⟶ High

(2) The ascending order of control unit design in terms of flexibility.

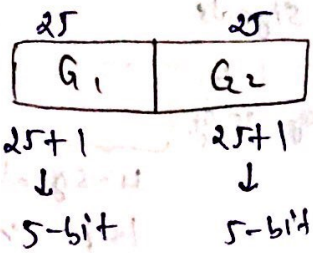
Hardwire → Horizontal → Vertical

low ⟶ High

Q- A uprog. control unit is required to generate total of 25 signals. Assume that during any inst^n at most 2 control signals are active. Min no. of bits required in the control word to generate the required control signal will be?



if all combination of 2 signals cannot be active at a time (if we require two signal & which are from Group which is not possible)



∴ Total 10-bits are required

* RISC & CISC processor.

RISC	CISC
→ Reduce instⁿ set computer	→ Complex instruction set computer
→ Support simple type of instⁿ	→ support complex instⁿ type
→ It support less no. of instruction	→ support more number of instruction
→ It has fixed size instruction	→ Variable size instruction
→ less no. of addressing mode	→ More no. of addressing mode
→ Support hardwired control unit	→ Support both hardwired & upragramed control unit
→ Register to Register arithmetic opⁿ only	→ Reg. to Mem. or mem. to Reg. Arith. opⁿ possible.
→ Support more no. of register	→ Support less no. of registers
Eg. Motorola, ARM power PC	Eg. Pentium

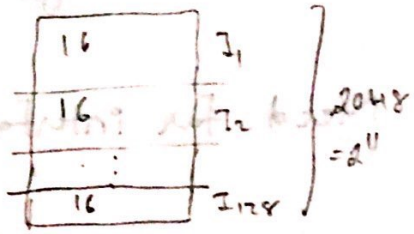
Q- Consider Horizontal μprog. control unit design in which CPU has to support 128 distinct inst.ⁿ Each inst.ⁿ required 16 μop.ⁿ. The system is using 63 control signals and has 1-add. μinst.ⁿ format. Each μinst.ⁿ has three fields namely control word, next address & condition select. There are 16 flag conditions.

What is the size of μinst.ⁿ.

What is the size of control memory in bits.

→ # inst.ⁿ = 128

Each inst.ⁿ = 16 - micro-opⁿ

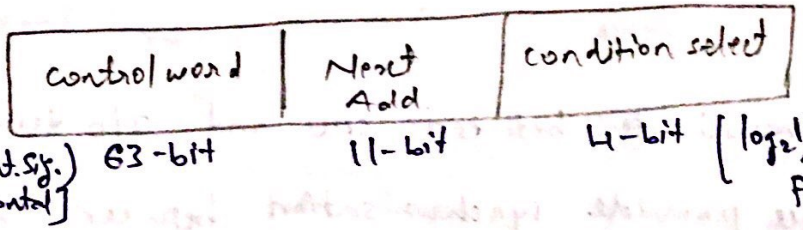


Total μinstⁿ = 128 × 16

= 2048

= 2^{11}

Control memory address = 11 bits



Control memory size = 2^{11} * 78 bits

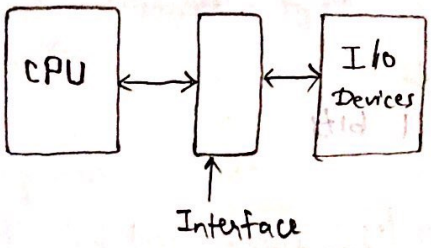
=

* I/O organization

Peripheral Device: The devices which are connected to CPU externally apart from memory.

Peripheral Devices
- → Input Device
- → Output Device
- → Storage Device

Need for interface:



→ To communicate between cpu and I/o device.

→ Interface provide synchronisation between them, as CPU is faster device & I/o devices are ~~slow~~ slower.

→ Conversion of signals.

→ I/o devices should not interfere to other device & CPU. which is managed by interface.

→ Conversion of data formates

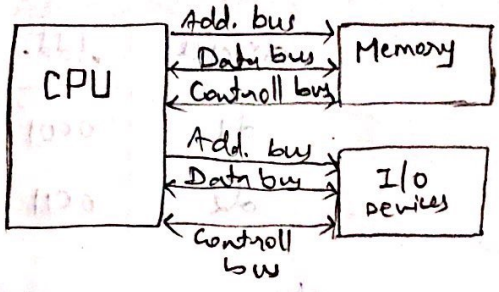
→ Interfaces are H/w + S/w

I/O Process
- → Interface
- → DMAC (DMA controller)
- → I/O instruction execution

Communication of CPU with Memory and I/O

In three ways CPU can communicate with memory and I/O.

(i) Separate buses for both memory and I/O.



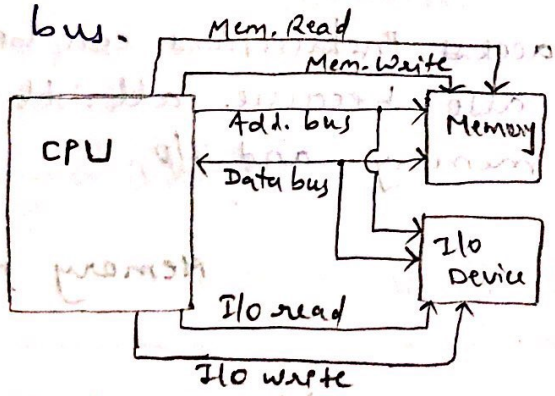
Adv.

Easy to implement

Disadv.

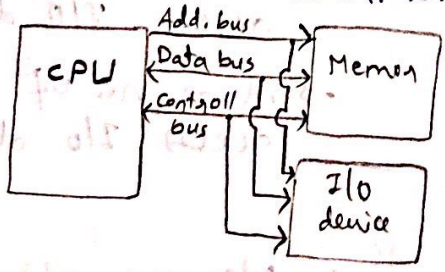
Cost will be more

(ii) Address bus & data bus is common but separate control bus.



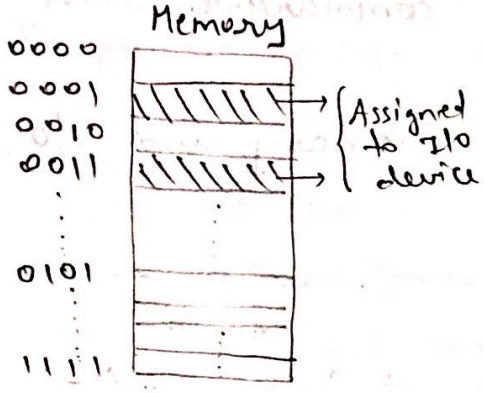
Isolated I/O
or
I/O-mapped I/O
or
Port-mapped I/O

(iii) All the buses are common for I/O and memory.



Memory-mapped I/O

Assume system with 16 Byte memory (Byte Addressable) and 2 I/O devices (d1, d2)



Few addresses are assigned to I/O devices

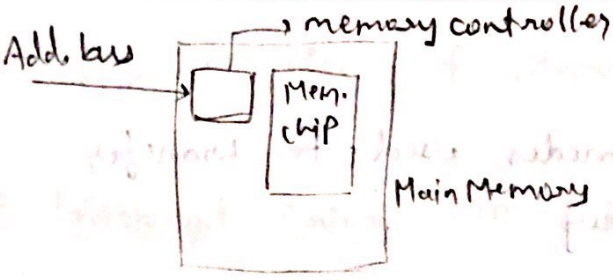
device	Add.
d1	0001
d2	0011

Whenever CPU generates address, it check whether it is assigned to I/O devices. if its assigned then I/O devices is accessed, otherwise memory will access.

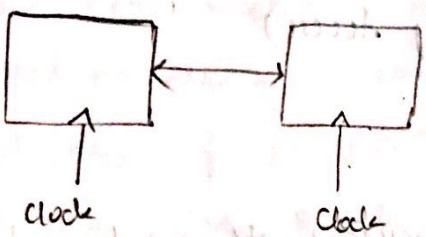
→ Some memory are wasted.

→ All the memory access instructions can be used to access I/O devices also because all three buses are common for memory and I/O.

I/O-mapped I/O	Memory mapped I/O
→ No memory wastage	→ Some memory are wasted
→ less no. of instructions to access I/O devices compared to Mem. mapped I/O	→ More no. of instⁿs. to access I/O devices.
→ Less no. of addressing modes to access I/O devices	→ More no of add. modes to access I/O devices.
→ separate address space for both memory & I/O	→ Memory address space is shared between both.



Seriel Asynchroneous Data Transfer

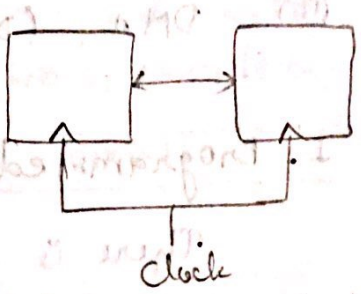


Seriel Asynchroneous data transfer.

→ separate clock generates for both device

→ required external synchronous device, because one may be slower & one faster

→ Performance is better

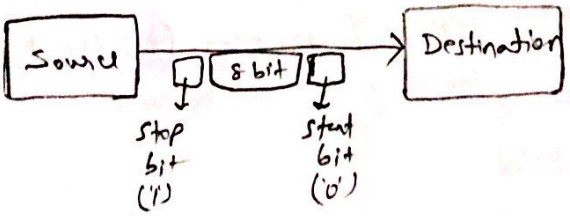


Seriel synchronous data transfer

→ common clock generator for both device

→ if any device is slower then transfer will be synchronise based on slower device

→ Performance is degraded



Mode of Data Transfer

There are three different modes used to transfer data b/w I/O & CPU & memory

(i) Programmed I/O

(ii) Interrupt driven I/O

(iii) DMA (Direct Memory Access)

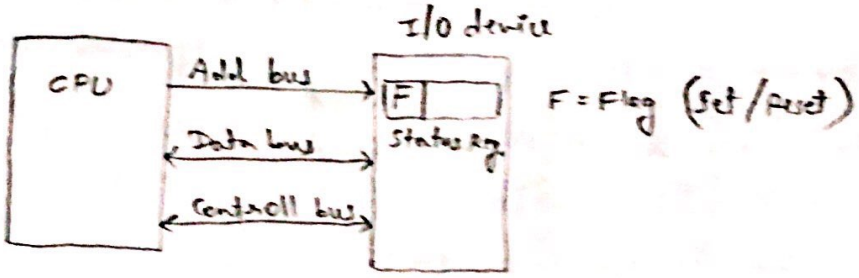
I. Programmed I/O

There is no any provision through which I/O device can inform CPU about data transfer. In such case the device sets its own status and waits.

CPU runs a program periodicaly and checks the status of each and every device one by one.

If any device has its status set then CPU performs data transfer for that device.

→ CPU time will be wasted because CPU is undergoes waiting until the completion of the I/O operation. This waiting time is depending on the speed of I/O device.



Total Data Transfer Time = Time to check status + Actual data transfer time

= Time required to read status Register from device + actual data transfer time

→ Default size of status register is 1 Byte.

II. Interrupt Driven I/O / Interrupt Initiated I/O

I/O devices have provision (interrupt signal) to inform CPU about data transfer.

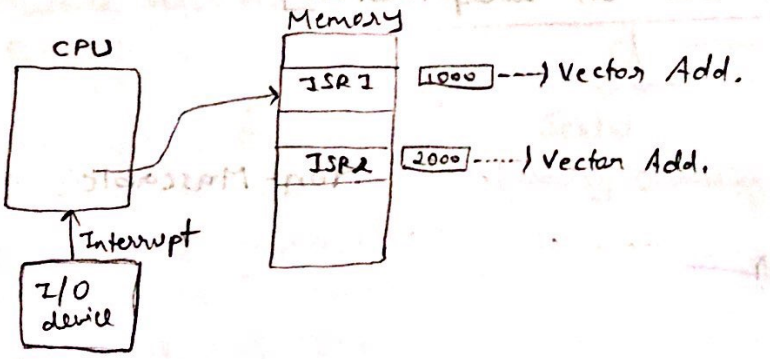
When CPU receives interrupt from the device

(i) CPU executes current instruction completely
(ii) Status of current running process is stored on to the stack
(iii) CPU takes a branch to service the interrupt
(iv) after the interrupt service, the previous program is resumed using the status from the stack.

ISR :- interrupt Service Routine

A function or routine execution of which services the interrupt

Vector Address : Starting address of ISR is known as vector address



Interrupts based on vectored Add.

- **Vectored Interrupt**
  - device sends starting add. of ISR along with the interrupt signal. In such case CPU jumps to that add. directly and executes ISR.
- **Non-vectored Interrupt (Scalar)**
  - Device sends only interrupt signal to CPU in such case, CPU executes a default service routine first. which helps CPU to resolve the scenario and helps CPU to get actual ISR Address.

~~In Non-vectored interrupt~~ & ~~ISR~~.

→ Every time an interrupt occur CPU takes branch to a specific address only, this type of interrupt is non-vectored interrupt.

Interrupts Based on acceptability

- **Mascable**
  - CPU can accept or can reject
    - Abort the interrupt
    - Keep in pending
- **Non-Mascable**

Interrupts based on Source

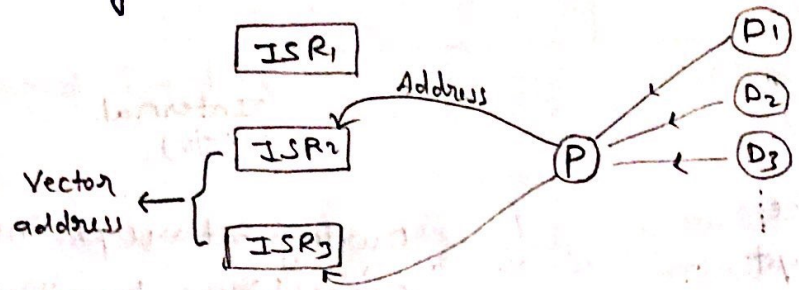
- **External (H/w)**
  - If any I/O device generates interrupt
- **Internal (S/w)**
  - During interrupt instruction execution, because of any unacceptable error CPU interrupts itself.
  - Eg. (i) page fault
    (ii) Addressing error in segmentation
    (iii) Divide by zero.

→ If two device generates interrupt simultaneously, then the interrupt from highest priority device should be interrupt first.

Interrupts based on priority handling

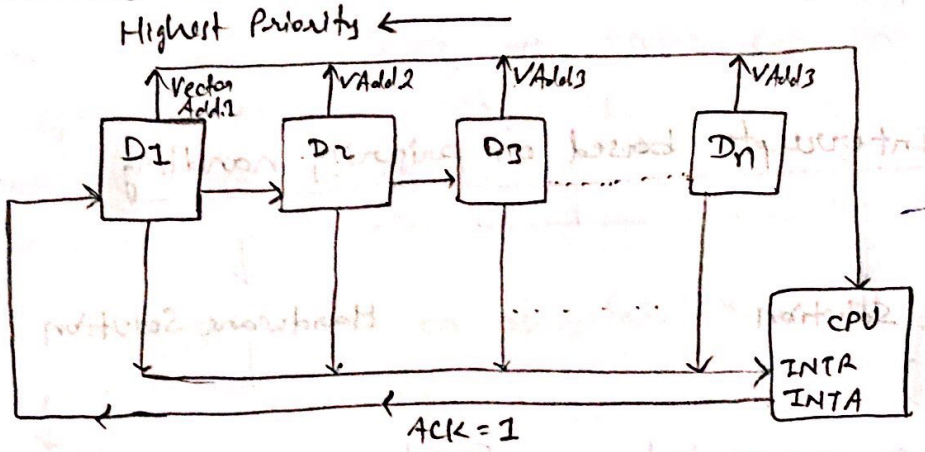
- **Software Solution (Polling)**
  - all interrupts are serviced by branching to the same service program. This program then check with each device if it is the one generating the interrupt. The order of checking is determined by the priority that has to be set.
  - → this method is quite slow.
- **Hardware Solution**
  - **Serial (Daisy chaining)**
  - **Parallel**

Polling



→ Device having highest priority is checked first and then devices are checked in descending order of priority.
If device is checked to be generating the interrupt, another service program is called which works specifically for that particular device.

Daisy Chain (H/w polling)



→ ACK sent to device one, if it doesn't need it, it send to another device like wise up to the end.

→ If two device raise the interrupt, then priority is given to one which is closer to the

→ Priority is based on the physical location of device.

→ Starvation is theoritically possible but not practically.

Time Required for data transfer (interrupt service) = Interrupt overhead time + actual interrupt service time [Execution of ISR]

Q-6 A device with data transfer rate 10 kB/sec is connected to CPU. Data is transferred byte-wise. Interrupt overhead be 4 μsec. What is the min. performance gain of operating device under interrupt mode over operating it under program controlled mode?

→ In Programmed mode:

10 kB → 1 sec.

1 B → $\frac{1 \text{ B} \times 1 \text{ sec}}{10 \times 10^3 \text{ B}}$ = 0.1 ms [To check status]

In interrupt mode = 4 μsec [overhead only]

Performance gain = $\frac{0.1 \times 10^{-3}}{4 \times 10^{-6}}$ = 25

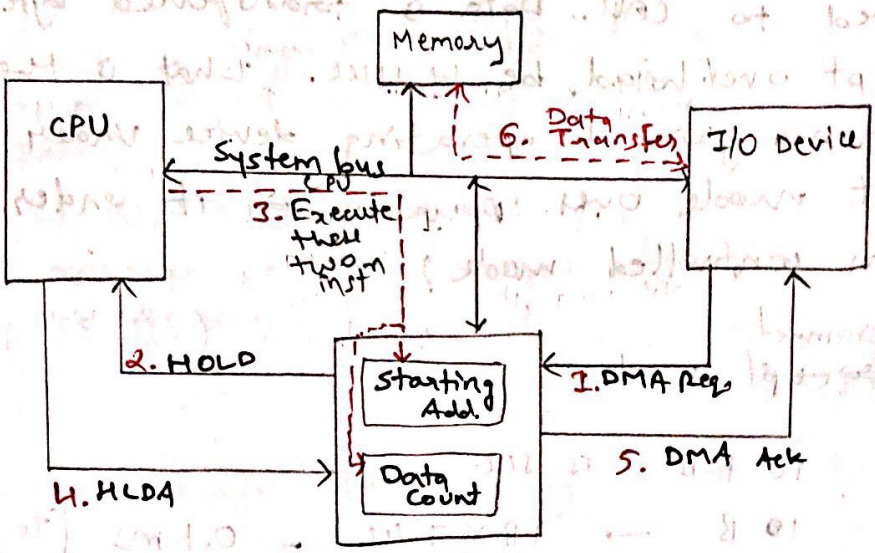
Performance gain or Speedup = $\frac{\text{Performance of interrupt mode}}{\text{Performance of prog. mode}}$

= $\frac{\frac{1}{t_{interrupt}}}{\frac{1}{t_{prog.}}}$ = $\frac{t_{prog.}}{t_{interrupt}}$ = $\frac{0.1 \text{ ms}}{4 \mu s}$ = 25

DMA

It enables direct data transfer between memory and I/o device without any interference of CPU.

To implement DMA technique a specific hardware is used known as DMA controller (DMAC).



Starting Add. : The address starting from where the data transfer should be performed in memory.

Data Count : Size of data to be transfered.

In Byte ⇒ Byte addresable memory

In words ⇒ Word addressable memory

→ 1. IO device will send DMA req. to DMAC

→ 2. DMAC will send HOLD signal to CPU

→ 3. CPU execute two instruction to load DMA registers IOAR and Data Count

→ 4. CPU wait for next DMA breakpoint (complete execution of current inst^n) and CPU will send HLDA

to DMAC, It means microprocessor has released control of address bus, data bus to DMAC.

→ 5. Now DMAC will send one Ack to IO device.

→ 6. Now with the help of ip-op read and memory write signal the data is transferred from IO device to memory.
After word is transfer, IOAR & Data count updated.

→ During DMA transfer, CPU can perform only those operations which do not required System Bus, which means CPU will be idle mostly.

→ During data transfer if Data count not yet reached zero but I/O device is not ready to send or receive next data, then DMA controller release the system bus by deactivating bus request.

→ If Data Count is zero, then DMA controller interrupts CPU and CPU will check status of DMA.

DMA Transfer Mode

Once DMA controller gets the control of the System bus, after how much time (how much data is transferred) The CPU takes the control of the buses back.

(1) **Burst Mode**: The entire amount of data is transferred before CPU takes back the control of buses.

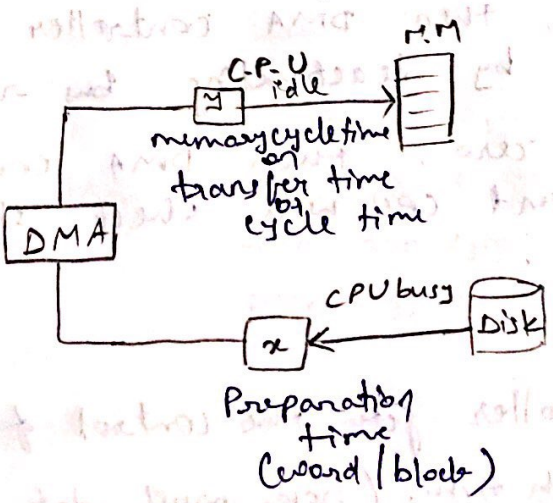
(2) Block Mode:

DMA return the control of buses after a transfer of block.

Size of Block : 512 B – 1024 B
(General)

(3) Cycle Stealing Mode:

The slow IO device takes some time to prepare the data, during that time CPU keeps the control of buses. Once the data is ready the control of the buses will given to DMA controller for one cycle in which the prepared data can be transferred to memory.



Time required to prepare the data = t_x

Time required to transfer the data = t_y

$$\% \text{ of CPU is blocked} = \frac{t_y}{t_x + t_y} * 100$$

$$\% \text{ of CPU is Busy} = \frac{t_x}{t_x + t_y} * 100$$

The data ~~transf~~ preparation can be performed during data transfer.

$$\% \text{ of time CPU is blocked} = \frac{t_y}{t_x} * 100\%$$

; $t_x \gg 2t_y$

$$= \frac{t_y}{t_x + t_y} * 100\%, \text{ otherwise}$$

	Burst Mode	Block transfer Mode	Cycle stealing Mode
1. Used to Transfer	Small data size	Moderate	Large size data
2. In one time data transfer	Large	moderate	Small
3. CPU waiting	More	Moderate	less
4. DMAC Waiting	Less	Moderate	More

→ DMA controller is a special purpose processor which is used for Data transfer b/w memory & I/O
↓
It can generate addresses & control signal like read & write for memory

Q-2 The size of data count register of DMA controller is 16 bits. The processor needs to transfer file of 29,154 KB from disk to m.m. Mem. is byte addressable. The min. no. of times the DMA controller need to get control of system bus from processor to transfer the file from disk to M.M is.

$$\frac{\text{File size}}{\text{Reg. Count}} = \frac{29154 \times 2^{10} \text{ Byte}}{\underbrace{2^{16}-1}_{\text{at count 0 we are not transferring any Byte}}} = 455.53$$

$$= 456 \text{ times}$$

Q- Consider a device with 1MBPS transfer rate which is operating in cycle stealing mode of DMA. It requires 2 μs to transfer the data from device to memory when it is ready or prepared. The data preparation starts only after the previous data is transfered. What is the % of time processor is blocked due to DMA. If data size is 16 Bytes.

→

1MB 1sec

$$16B = \frac{16B}{1MB} = 16 \text{ μs}$$

∴ preparation time = 16 μs

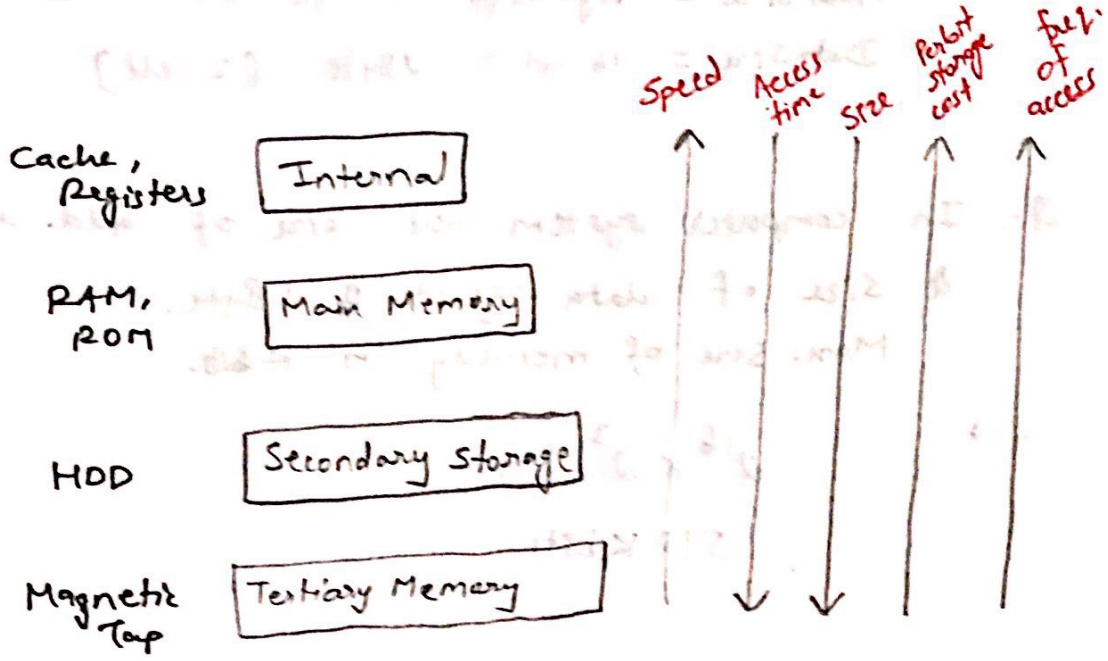
∴ Transfer time = 2 μs

$$\% \text{ CPU block} = \frac{2}{16+2} = \frac{2}{18} = 11.11\%$$

* Memory Organization

Memory: A physical device to store the content in the system.

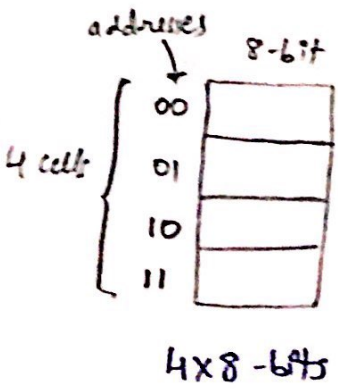
→ Instead of using single memory a memory hierarchy are used in the system.



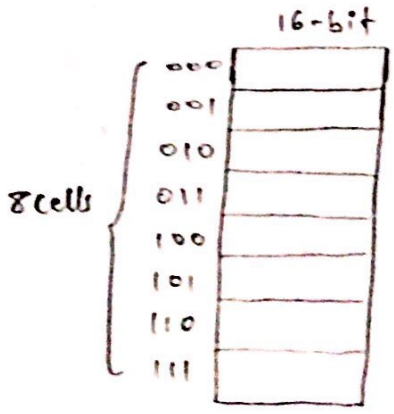
Goal: 1. To maximize memory access speed.
2. To minimize per bit storage cost.

Memory Representation

Memory is represented by no. of cells x 1 cell capacity.



4x8-bits



8x16-bits

Memory Representation = #cell * 1 cell capacity

Q- Memory: 128 K x 16 -bit

Add. size = $\log(128K$ = 17 -bit

Data Size = 16-bit = 2 Byte [1 cell]

Q- In computer system the size of add. reg. is 16-bit & size of data register is 1 Byte. Max. size of memory in K-Bits.

→ $2^{16} \times 2^{3}$ bits

512 K bits

Main Memory

It is used to store current running processes (Instruction) and their data.

ROM: non-volatile memory
RAM: Volatile Memory

Types of ROM

(i) EROM: (Erasable ROM)

With the help of UV rays or laser rays the content of such ROM can be erase.

(ii) PROM: (Programmable ROM)

One time writing can be performed in such ROM and after that only read.

(iii) EPROM: Erasable Programmable ROM

(iv) EEPROM: Electricaly Erasable Programmable ROM
With the help electric current the content can be erase.

(v) EAPROM: Electricaly Alterable Programmable ROM
With the help of electric current, content can be update or alter.

Q- A ROM is used to store table of multiplication of two 4 bit unsigned integer. The size of ROM is.

→

A	B	A * B	A + B	Mult.^{n} Table	Add. Table
4-bits	4-bits	8 bits (Max)	5 bits	$2^8 \times 8$ bits	$2^8 \times 5$ bits
n-bits	n-bits	2n bits	(n+1) bits	$2^{2n} \times 2n$ bits	$2^{2n} \times (n+1)$ bits

Memory = $2^8 \times 8$ bits
= 2K bits

RAM

Types of RAM

Static (SRAM)

Built : Flip-flop

No recharge or refresh

Dynamic (DRAM)

Built : Capacitors (Content is stored in the form of electrical charges)

Electrical charges tend to discharge

Periodical refresh or recharge is required

SRAM

→ Faster

→ Costly

→ More power consumption

→ Used for cache memory

DRAM

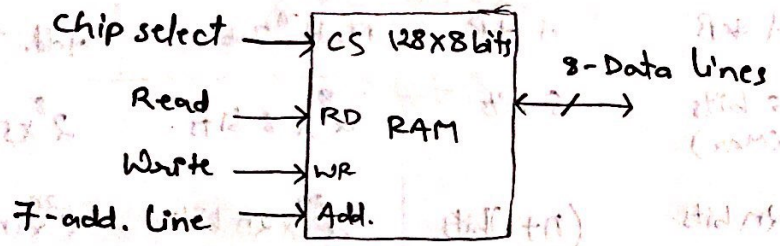
→ Slower [during refresh, read/write can not be performed]

→ less costly

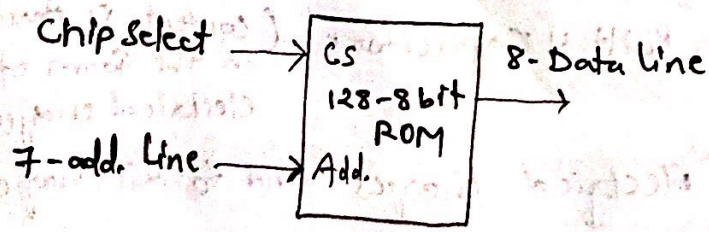
→ less power consumption

→ Uses for main memory of computer systems

RAM Chip



CS	RD	WR	operation
0	X	X	No op.ⁿ
1	0	0	No op.ⁿ
1	0	1	Write op.ⁿ
1	1	X	Read op.ⁿ



CS	operation
0	No op.ⁿ
1	Read

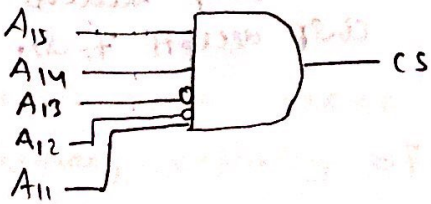
No. of pins = 1+7+8
= 16 - pins

Q- A ROM chip of 1024 x 8 bits has 4 select line and operates from a 5 watt power supply. How many pins are needed.

→ select line = 4
add. line = 10
data line = 8
Power supply = 1
Ground = 1
Total = 24

Q- ~~In memory select design~~ system

Q- Chip select logic for certain DRAM chip in memory system design is shown below.
Assume that the memory system has 16-address line A_{15} to A_0. what is the range of add. (In Hexa.) of the memory system that can get enable by chip select (cs) signal.



(A) C800 to CFFF
(B) C800 to C8FF
(C) DA00 to DFFF
(D) CA00 to CAFF

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10} ... A_0
1	1	0	0	1	0 0 ... 0
1	1	0	:	:	0 0 ... 1
1	1	:	:	:	0 0 ... 1 0
:	:	:	:	:	: : :
1	1	0	0	1	1 1 ... 1 1

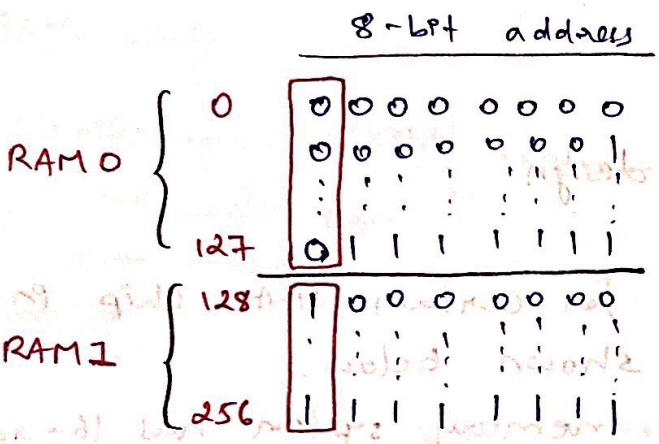
Using Multiple Chips (M.M)

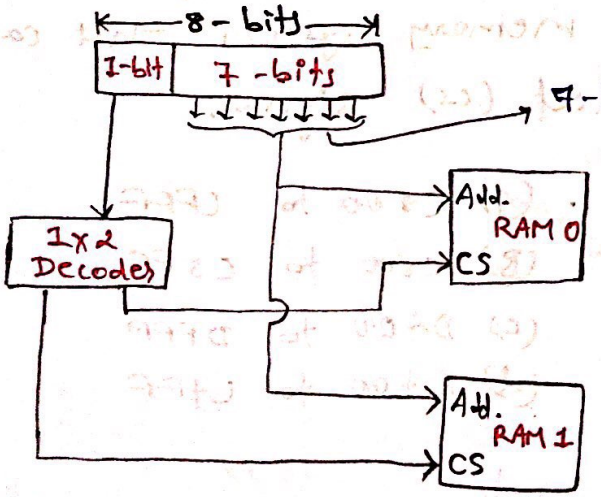
Total capacity = #chips * 1 chip capacity

Assume,
in memory system 2 chips of size 128 x 8 bits are used

∴ 128 x 8 - bits 7-bit address ⇒ RAM 0
128 x 8 - bits 7-bit address ⇒ RAM 1

Total 256 x 8 - bits 8-bit address





7-line address are common to each chip because each chip accepts 7-bits address

Q- (i) How many 128×8 bits RAM chips are needed to provide a memory capacity of 2048 Bytes.

(ii) How many lines of the address bus must be used to access 2048 Bytes of memory. How many of these lines will be common to all chips

(iii) How many lines must be decoded for the chip select. Specify the size of decodes.

→ Capacity = 2048 × 8 bits

(i) RAM size = 128 × 8 bits

$$\#RAM = \frac{2^{11} \times 2^3 \text{ bits}}{2^{10} \text{ bits}} = 16$$

(ii) address lines = $\log 2048$

= 11 lines

Common lines to all chips = 7 bit

(iii) lines for decoded = $\log 16$

= 4 lines

size of decoder is 4×16

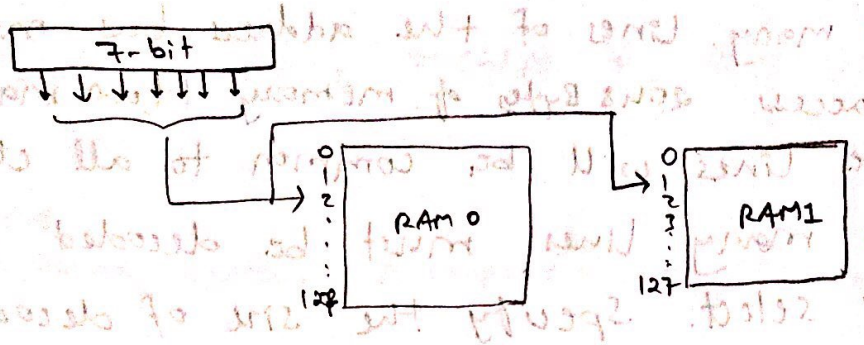
Q- How many 128×8 RAM chips are needed to provide a memory capacity of 128×16 bits?

→ $\frac{128 \times 16}{128 \times 8} = 2$

Total capacity = 128 × 16 - bit Add. line = 7

1 RAM Chip capacity = 128 × 8 - bit Add. line = 7

Horizontal Arrangement



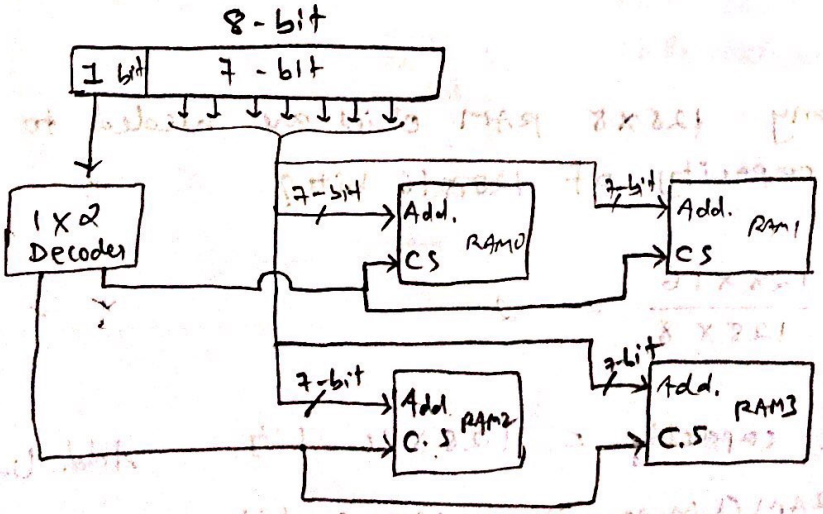
NOTE: (i) If required no of address are more, use vertical arrangment

(ii) If required no. of data lines are more, use Horizontal arrangment

(iii) If both are more, use hybrid arrangment (vertical & Horizontal)

Q- How many 128x8 RAM chips are needed to provide memory capacity of 256 x 16 bits?

→ Hori $\frac{256}{128} \times \frac{16}{8} = 4$



14

9- Consider a M.M system that consists of 8 memory modules attached to the system bus, which is 1 word wide. When a ~~right~~ Write request is made, the bus is occupied for 100 ns. by the data, add and control signal. During the same 100 ns and for 500ns there after, addressed memory module executes 1 cycle accepting and storing the data. The (internal) operation of different memory modules may overlap in time, but only one request can be on the bus at any time. The max. no. of stores (of 1 word each) that can be initiated in 1 ms is ?

→

In 100 ns 1 store initiated

$$1\text{ ms} = \frac{1\times10^{-3}}{100\times10^{-9}}$$

$$= 10{,}000 \text{ stores initiated}$$

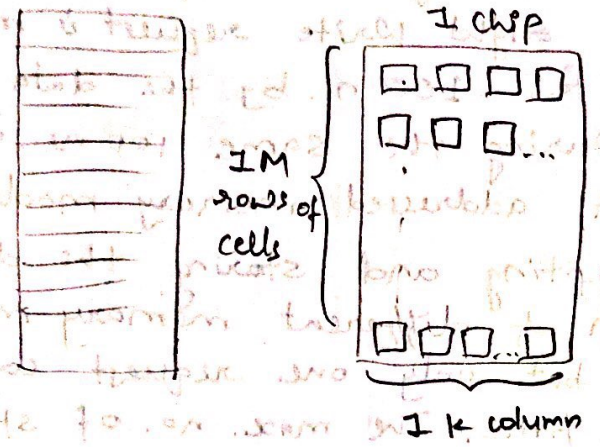
~~10,000~~
~~5~~
~~9995~~

How many request can be successfully completed in 1 ms.

→

10,000
- 5
9995

* Arrangment of cells in DRAM chip



Assume DRAM chip of size 1GB, which has 1M rows of cells with 1k cells in each row.

→ In one refresh operation, 1 row of cells can be refreshed

1 chip refresh time = no. of rows * 1 refresh time

if 'n' chips are there in one memory system.

Memory system refresh time = 1 chip refresh time

9- A main memory unit with capacity of 4MB is built using 1M x 1 bit DRAM chip. Each DRAM chip has 1k rows of cells with 1k cells in each row. The time taken for a single refresh operation is 100 ns. The time required to perform 1 refresh operation on all the cells in the memory unit is.

→ $\frac{4 \times 2^{20} \times 2^3 \text{ bit}}{2^{20} \times 1} = 32$ DRAM

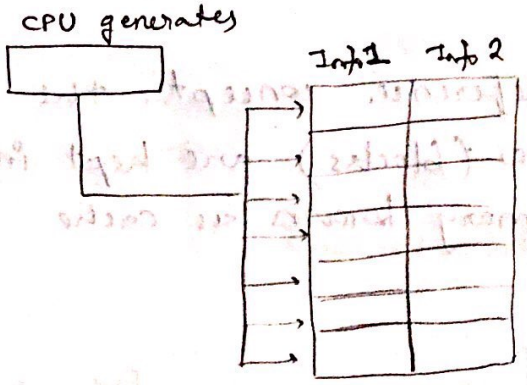
1 chip refresh time = 2k * 100 ns

$= 2^{10} \times 100$ ns

∴ Memory refresh time = 1 chip refresh time

$= 100 \times 2^{10}$ ns

* Associative Memory



→ Cells do not have addresses

→ Every cell contain 2 info.
1. info 1 = CPU knows
2. info 2 = CPU requires

→ Searching performed based on content ⇒ Content addressable Memory

→ Content matching is done in parallel in all the cells, because each cell has one matching logic hardware.

→ Associative Memory is faster & expensive.

Locality of reference

If CPU refers the memory at specific address then the same address or the nearby address will be referred soon. This phenomenon is known as locality of reference.

Types of locality Reference

Spatial (space)

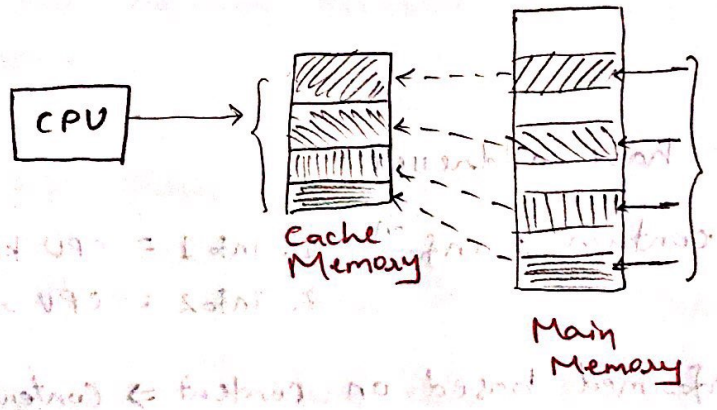
if nearby addresses are refered soon

Temporal (Time)

if same address is refered soon.

* Cache Memory

→ Based on locality of reference concept, the current demanded localities (blocks) are kept into a smaller and faster memory known as cache memory.



Every time searching in MM for current demand block takes more time, So kept those blocks in cache Memory.

→ Use of cache decreases the avg memory access time.

Working of cache Memory

If requested content is present in cache memory then its cache hit.

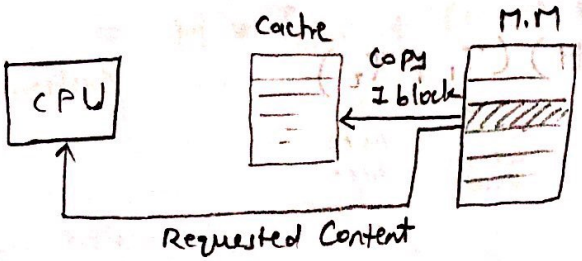
If required content is not present in cache memory then its cache miss.

Performance of cache is given by cache hit ratio.

Hit Ratio (h) = $\frac{\text{No. of hits}}{\text{Total references}}$

Miss ratio = $(1-h)$

In case of cache miss one block (which contains requested content) ~~one block~~ is copy from main memory to cache memory.

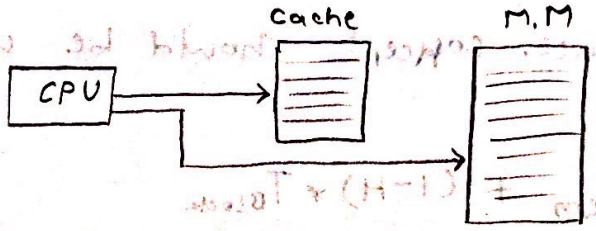


Avg. Memory Access time = Hit * Mem. access time when hit in cache + (1-H) * Mem. access time when miss in cache

* Types of Cache Access

1. Simultaneous Access

The request for cache and main memory are generated simultaneously.



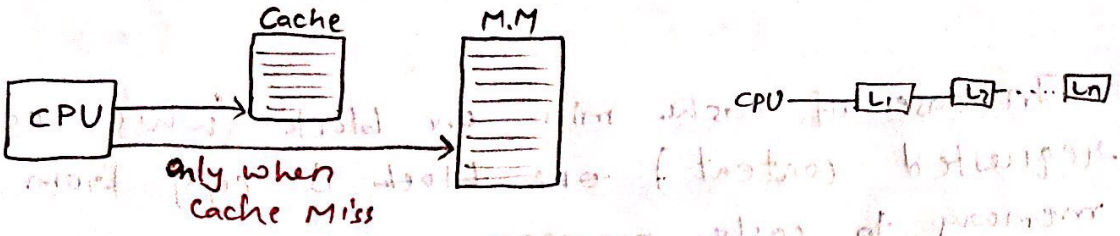
$T_{avg} = H * T_1 + (1-H) * T_2$

T_1 ↓ Cache access time

T_2 ↓ M.M access time

2. Hierarchical Access

Only the faster memory is access first.
If cache miss, then only it search in M.M.



$$T_{avg} = H * T_1 + (1-H)(T_1 + T_2)$$

(T_1 → Cache access time; T_1 → cache access time; T_2 → M.M access time)

$$= HT_1 + T_1 - HT_1 + T_2 - HT_2$$

$$= T_1 + (1-H)T_2$$

→ If Cache memory & M.M access time not given use generalized formula

→ If cache memory & M.M access time are given
- If any one of 2 word given Hierarchical or level
- else, Simultanous access

→ If locality of reference concept should be used then,

In simultaneous access

$$T_{avg} = H * T_{cm} + (1-H) * T_{Block}$$

$$T_{Block} = \text{Block size} * T_{M.M}$$

In Hierarchical Access.

$$T_{avg} = H * T_{c.M} + (1-H)(T_{c.M} + T_{Block})$$

when, block has to be transfered from M.M to cache then,

In simultanous access

$$T_{avg\ (with\ transfer)} = H * T_{c.M} + (1-H) * (T_{Block} + T_{c.M})$$

↓ To write back in cache memory

In Hierarchical Access

$$T_{avg\ (with\ transfer)} = H * T_{c.M} + (1-H)(T_{c.M} + T_{Block} + T_{c.M})$$

↓ cache miss; ↓ Block access in M.M; ↓ write back in cache Memory

Q- In two level mem. Hier. if the top level has an access time of 8ns. & bottom level has an access time of 60ns. What is the hit rate on top level required to give an avg. access time of 10ns.

$$T_{avg} = H \times T_1 + (1-H) * (T_1 + T_2)$$

→ $10\ ns = H * 8\ ns + (1-H)(8\ ns + 60\ ns)$

$10\ ns = 8H\ ns + 68\ ns - 68H$

$\frac{-58}{-60} = H$

$\therefore \frac{58}{60} = H$

$H = 96.6\%$

Q- Uncached mem. access takes 7 times longer than accessed to cache. If cache has hit ratio 0.9. The ratio of cached mem. acc. time to uncached mem. access time is?

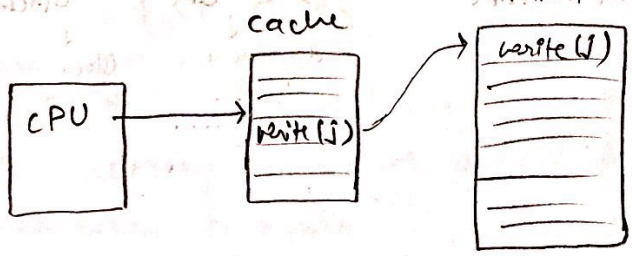
→ Cache - x

Uncached - $7x$

$$\frac{T_{Cache}}{T_{Uncache}} = \frac{0.9 * x + 0.1(7x)}{7x} = \frac{1.6}{7}$$

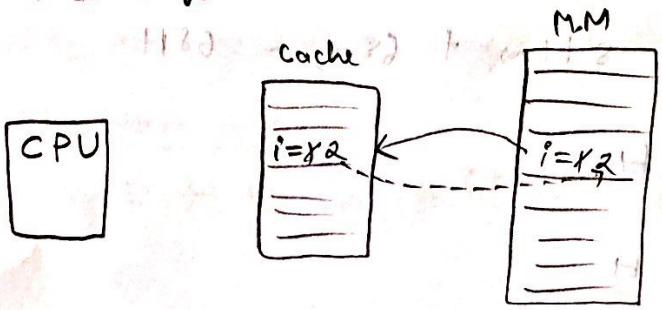
$$= 0.228$$

* Cache Write



If C.P.U performs write operation on cache content (duplicate copy) then the respective main memory content (original copy) should also be updated

I. Write Through



i=1 to 1000, it

As soon as the content of cache is updated, the respective main memory content is also updated. which means for the write operation C.P.U generates the request for both cache and main memory

$$T_{avg\ write\ through} = \max(T_{cm}, T_{m.m})$$

T_{cm} → tavg for cache, $T_{m.m}$ → tavg for m.m

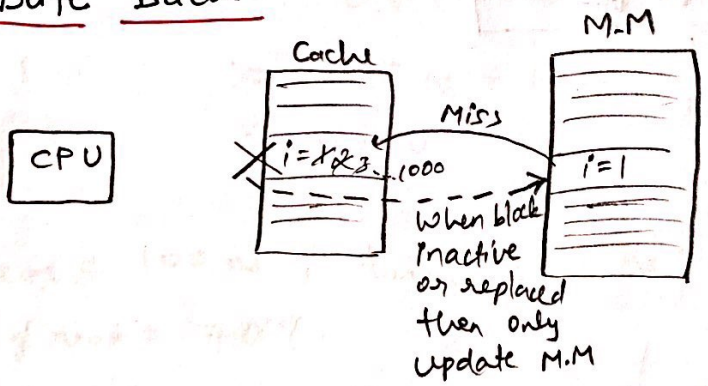
Adv.

Main memory always contains the updated content

Disadv.

Time consuming as every time when we update in cache we have to acces M.M also to update

II Write Back



Main memory is updated only when the cache block is inactive or to be replaced.

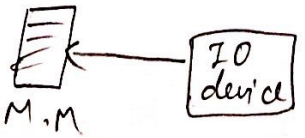
Adv.

Time saving

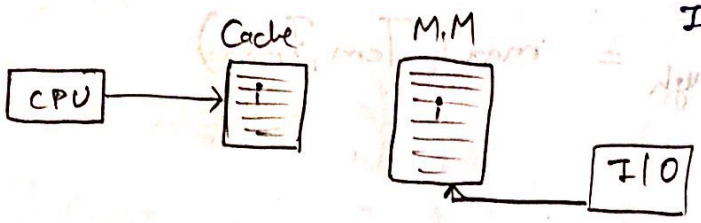
Dis Adv.

For the same content, cache & M.M may contain different value.

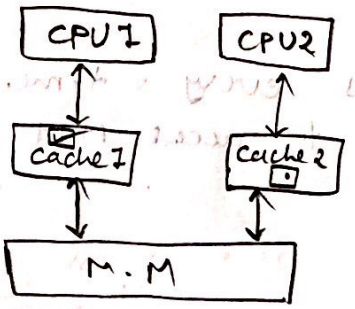
Ex. if I/O device wants to access memory content then it may read inconsistant data which is called as Memory coherence.



Memory Coherence: If two different devices (CPU & I/O) trying to access the same content from two different memories (CPU from cache & I/O from M.M)



Cache Coherence:



Multiprocessor System

Write through (T_{avg} overall)

$$T_{avg\ write\ through} = \max(T_{c.m}, T_{m.m})$$

$T_{avg\ read}$ = T_{avg} formula using hit ratio etc.

$$T_{avg\ overall\ (read\ write)} = f_r * T_{avg\ read} + f_w * T_{avg\ write}$$

Eg. $t_{avg\ read} = 20\ ns$

$t_{avg\ write} = 50\ ns$

Total 100 mem. reference

read = 80

write = 20

$\therefore T_{avg\ overall} = 0.8\ (20\ ns) + 0.2\ (50\ ns)$

$= 16\ ns + 10\ ns$

$= 26\ ns$

Q- $t_{c.m} = 100\ ns$, $t_{m.m} = 1000\ ns$, $H_{read} = 80\%$

% of read = 70%

Write through policy employed

$t_{avg\ read} = ?$, $t_{avg\ write} = ?$ $t_{avg\ overall} = ?$

→ $T_{avg\ write} = 1000\ ns$

$T_{avg\ read} = 0.80\ (100\ ns) + 0.20\ (1000\ ns)$

$= 80 + 200$

$= 280\ ns$

$T_{avg\ overall} = 0.7\ (280) + 0.3\ (1000)$

$= 196 + 300$

$= 496\ ns$

Q The memory access time is 1 ns for a read op^n with hit in cache, 5 ns for read op^n with miss in cache, 2 ns for a write op^n with a hit in cache & 10 ns for write op^n with a miss in cache. Execution of sequence of inst^n involves 100 IF op^n, 60 mem. operand read op^n and 40 memory operand write op^n. The cache hit ratio is 0.9. The avg memory access time (in ns) in executing the sequence of inst.^{n} is ?

→

~~Cache~~ hit = 0.9
~~Cache~~ Miss = 0.1

Read Cache hit time = 1 ns
Read Cache Miss = 5 ns
Write Cache hit = 2 ns
Write Cache Miss = 10 ns
Read op^n = 60 + 100
Write op^n = 40
Total = 200

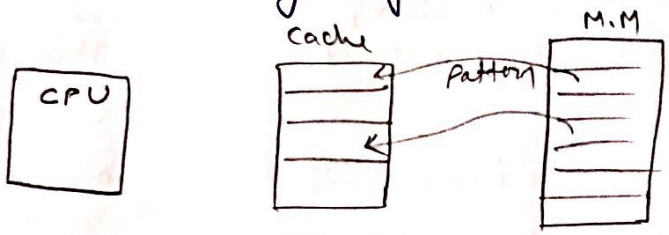
$$T_{avg.read} = 0.9\,(1\text{ ns}) + 0.1\,(5\text{ ns})$$
$$= 0.9\text{ ns} + 0.5\text{ ns}$$
$$= 1.4\text{ ns}$$

$$T_{avg\ write} = 0.9\,(2\text{ ns}) + 0.1\,(10\text{ ns})$$
$$= 1.8\text{ ns} + 1\text{ ns}$$
$$= 2.8\text{ ns}$$

$$T_{avg} = \frac{160}{200}(1.4) + \frac{40}{200}(2.8)$$
$$= 1.12 + 0.56$$
$$= 1.68\text{ ns}$$

* Cache Mapping

CPU always generates Main memory address.



There should be a pattern, with the help of which the main memory content from specific address is copied into cache memory to a specific location, this pattern is known as cache mapping.

With the help of mapping, cache memory is searched using M.M access.

Transformation of main memory data into cache memory is called as cache mapping.

Types

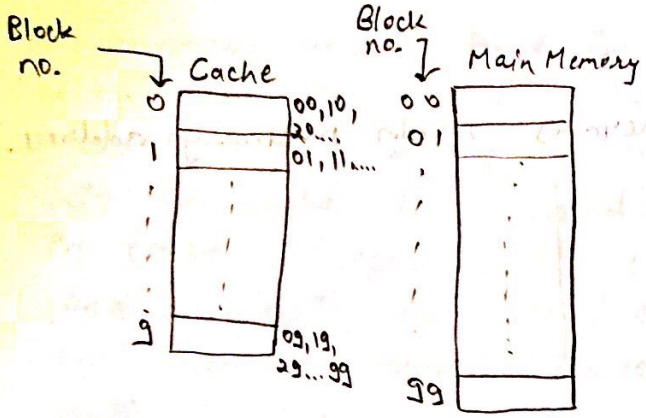
→ Direct Mapping
→ Set-Associative Mapping
→ Fully Associative Mapping

Direct Mapping

Assume,

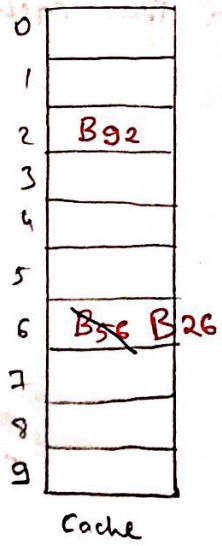
Cache memory contains 10 blocks (0-9)
Main memory contains 100 blocks (00-99)

C.M	0	1	2	3	4	5	6	7	8	9
M.M	00	01	02	03	04	05	06	07	08	09
	10	11	12	13	14	15	16	17	18	19
	⋮	⋮								⋮
	90	91	92	93	94	95	96	97	98	99

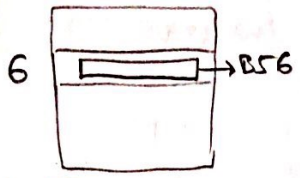


Cache Memory block no. for mapping = $k \bmod n = i$

k = main mem. block no.
n = no. of cache line
i = cache memory line no.



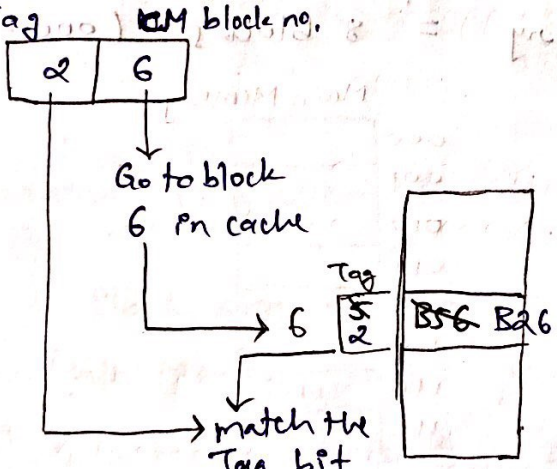
CPU Req. (M.M block no.)	C.M block no.	
56	56%10 = 6	Goto block no. 6 in cache, & check. It's miss in cache Bring Block no. 56 Content in cache at block no. 6
92	92%10 = 2	Its miss in cache, Bring B92 content from M.M to cache at 2.
26	26%10 = 6	Goto block no. 6 in cache & check. Block no. 56 is available in cache Hence, B26 is miss, Replace B56 by B26



Whenever B26 will come at cache block no. 6, B56 is already present, then how we will get the information that at cache block 6 there is Block of 56.

This problem can be resolve by using tag bit, which distinguish blw all competetion.

For request of M.M block 26



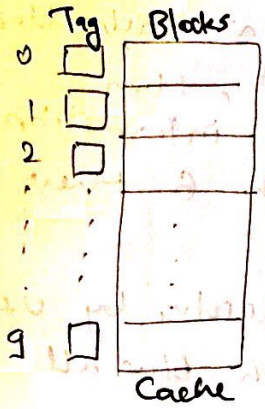
If Not matched

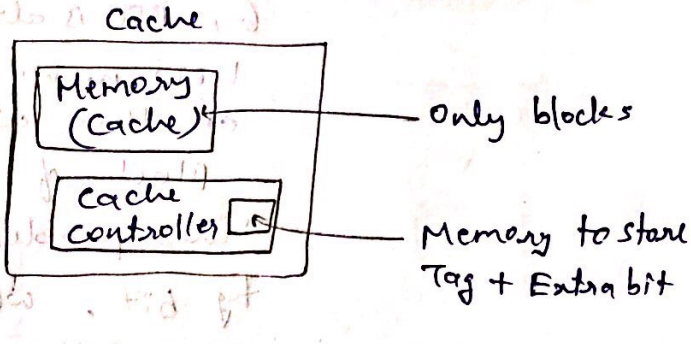
↓

Cache Miss

↓

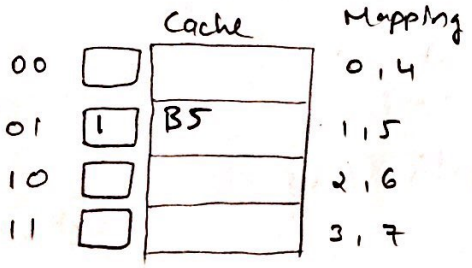
Bring block 26 from M.M to cache in block 6 in cache and update tag bit by 2

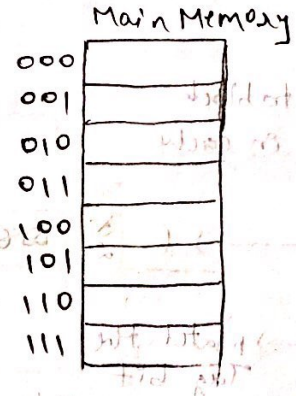




A tag information is stored in cache controller for each block in cache.

Assume, cache memory = 4 blocks (00 - 11)
Main Memory = 8 blocks (000 - 111)

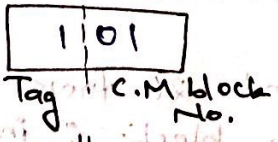




CPU Request block no. 5 $(101)_2$ of Main Memory

→ Mapping 5 % 4 = 1

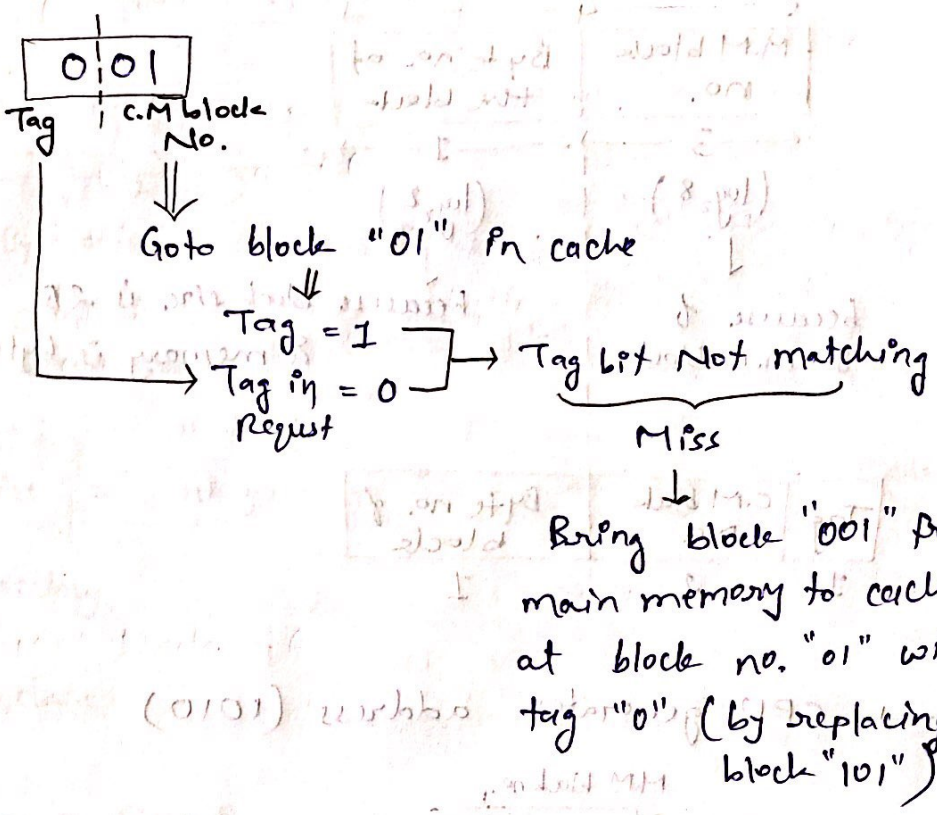
M.M. block No.



⇓

Goto Block "01" in cache & it's a miss, Bring B5 (101) in cache at block "01" with tag "1".

CPU request for block no. 1 $(001)_2$ of Main Memory



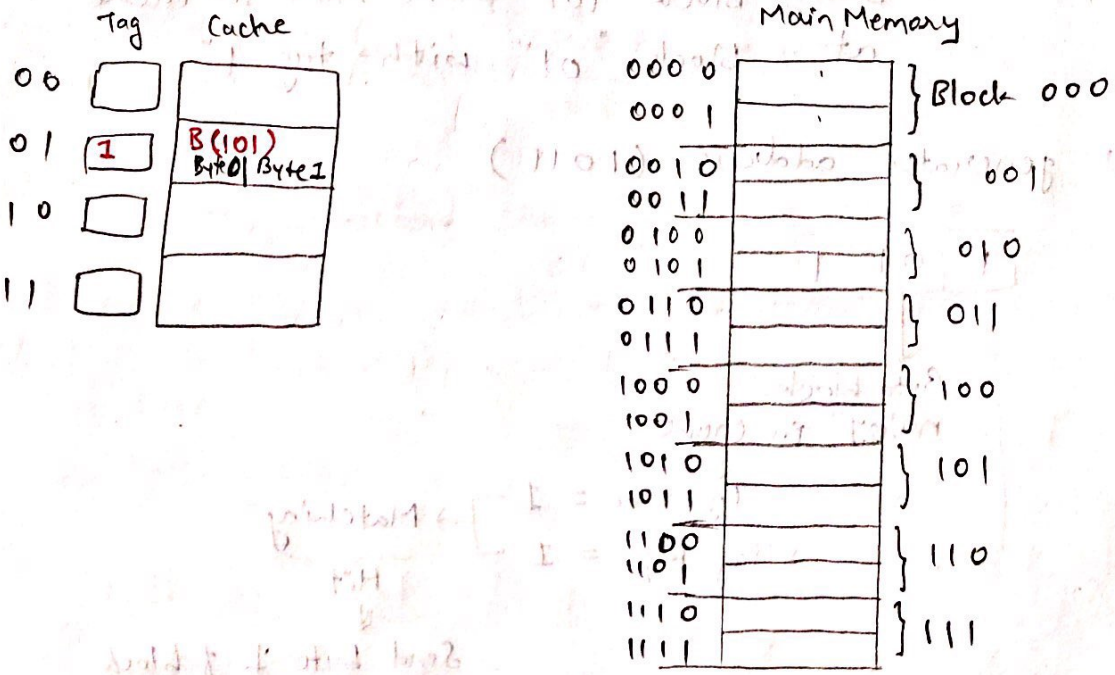
→ Comparator is used to compare tag bit.

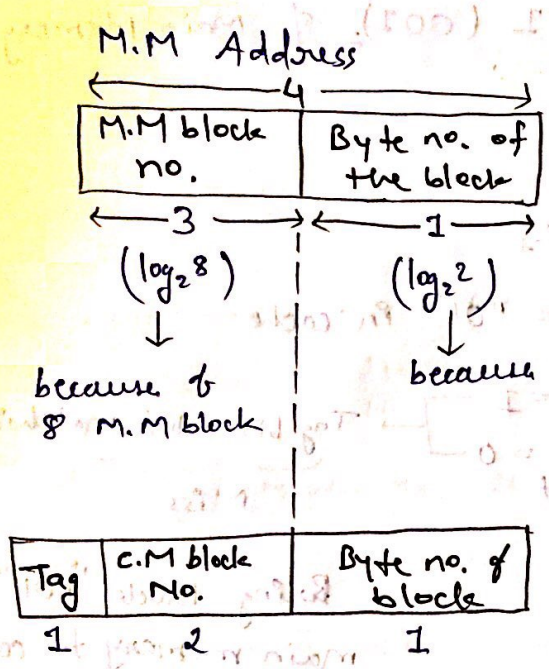
Assume, Block size = 2B

Main Memory Size = 8 * 2B = 16B

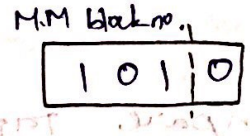
Main Memory is Byte Addressable

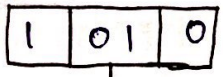
Main Memory address = 4-bit





→ CPU generates address (1010)





Goto block "01" in cache

⇓

Miss

⇓

Bring block "101" from M.M to cache at block "01" with tag "1"

→ CPU generates address (1011)

Goto block no:01" in cache

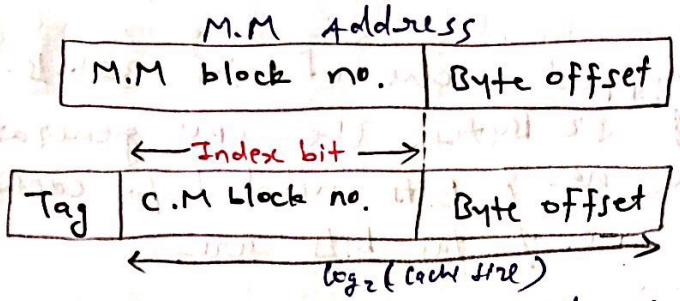
Tag mm = 1

Tag Req. = 1

→ Matching

Hit

Send byte 1 of block for access to CPU.



No. of bits for Byte offset = $\log_2(B.S)$ - bits

No. of bits for C.M block no. = $\log_2(\#C.B)$ - bits

Tag Memory size = no. of cache block * (Tag-size + Extra bits)

or

Size of Tag Directory

or

Size of meta-data

Ex: Cache size = 64 KB

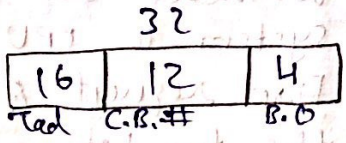
Block size = 16 B

M.M add. = 32-bits (Byte add.)

Direct Mapping

Tag size = ? , Tag directory size = ?

→



$\#\text{Cache line} = \frac{64\text{ KB}}{16\text{ B}} = 4\text{K}$

Tag directory size = 4K * 16 bit

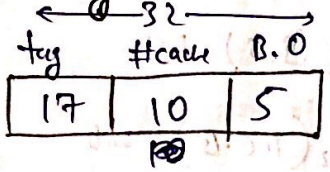
= 4K * 2 B

= 8 KB

2005

Q- Consider a Direct Mapped cache of size 32KB with block size of 32 Bytes. The CPU generates 32 bit addses. The no. of bits needed for cache indexing and no. of tag bits are.

→

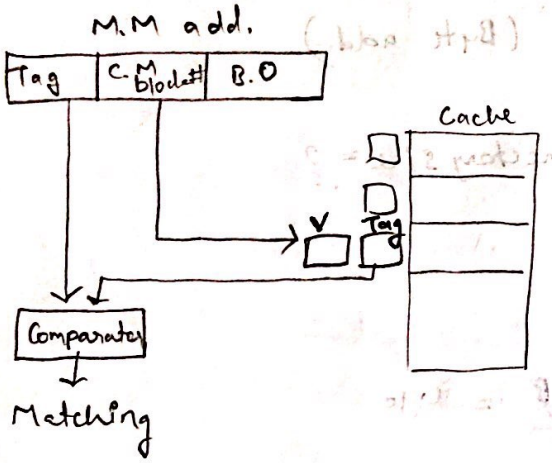


$$\#\text{ cache line} = \frac{32KB}{32B} = 1K$$

index bits = 10

tag bits = 17

Cache Initialization



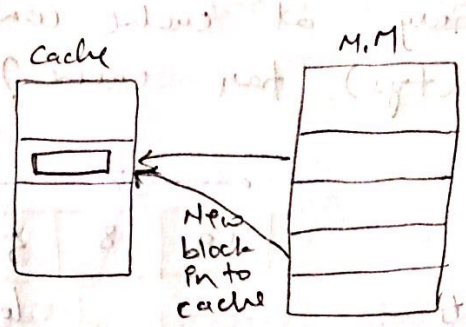
When system turn off, Memory get flushed & some garbage value is available.

Whenever we turn on the system & CPU generate first request & compare the tag bit in cache if somehow they match then CPU will access unwanted data from cache.

→ To solve this problem there is one valid bit in cache which gives the information whether tag field & block in cache is valid or not.
So initial valid bit is "0" when CPU generates request 1st time.

Valid bit 'V' → 0 invalid tag & cache block
→ 1 valid tag & cache block

Performance improvement of write-back Cache

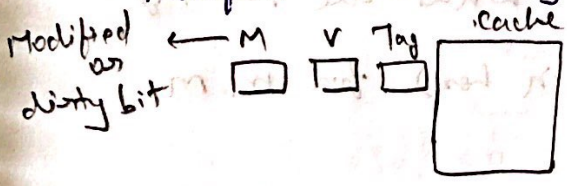


During read operation in cache we don't required any updation in M.M. So whenever conflict occurs in cache to replace block, we can just replace without updating in M.M.

During write operation in cache, if any conflict occurs in cache to replace block, then first we have to ~~wait~~ update in M.M ~~before~~ and then replace the block in cache.

So, for read operation we can save some amount of time & hence performance can be improve.

To improve performance we need to keep the information if read or write. So we required Modified bit or dirty bit in cache.



Q-6 8KB direct mapped write-back cache is organized as multiple block, each of size 32-bytes. The processor generates 32-bit add. The cache controller maintains the tag information for each cache block comprising of following:

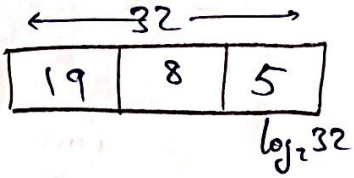
- 1 valid bit
- 1 Modified bit
- As many bits as the minimum needed to identify the memory block mapped in cache.

What is the total size of memory needed ~~to identify the memory~~ at cache controller to store meta-data (tags) for cache?

→ Cache size = 8 KB

Block size = 32 B

M.M add = 32-bits



$\#\text{cache line} = \frac{8\text{ KB}}{32\text{ B}} = 2^8$

Valid bit = 1 bit

Modi bit = 1 bit

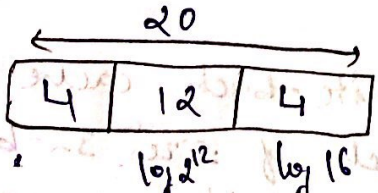
Size of Meta data = $2^8 \times (19 + 1 + 1)$

$= 2^8 \times 21$ bits

$= 5376$ bits

Q-19 Consider machine with byte add. main memory of 2^{20} Bytes, B. size = 16 B & Direct Mapped cache having 2^{12} cache lines. Let add. of two consecutive bytes in M.M be $(E201F)_{16}$ & $(E2020)_{16}$. What are the tag & cache line address (in hex) for M.M address $(E201F)_{16}$

→



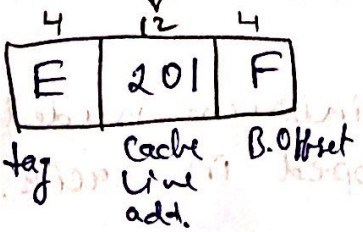
(a) E, 201

(b) F, 201

(c) E, E20

(d) 2, 01F

E 2 0 1 F



Q-15 A 64 word cache & 256 word M.M are partitioned into 16 word blocks. The tag information is

Block	Tag
0	10
1	10
2	00
3	01

Identify correct statement w.r.t availability of M.M words in cache

(a) words 50 & 132 are available in cache

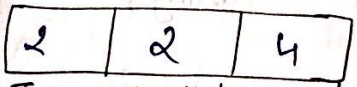
(b) Words 150 & 132 are available

(c) words 178 & 150 are available

(d) Words 35 & 50 are available

→

8



$\log_2 4$ $\log_2 16$

$$\#\text{cache block} = \frac{64}{16} = 4$$

Word no. 50 : 00110010

00 11 00 10

B.No 3 → Tag = 01 → Not matching

Tag = 00

So, 50 is not in cache

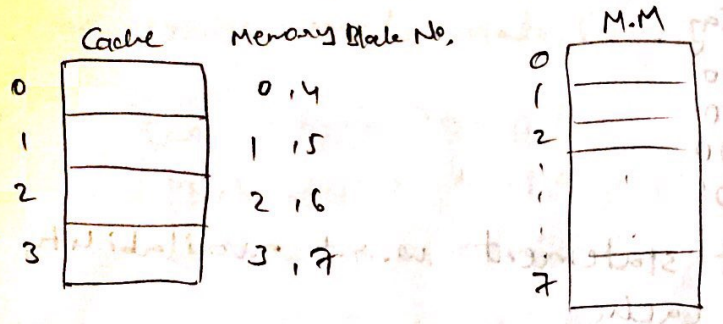
Word No. 132 : 10000100

B.No. 0 → Tag = 10 → Matching

Tag = 10

So, 132 is present

* Set - associative Mapping

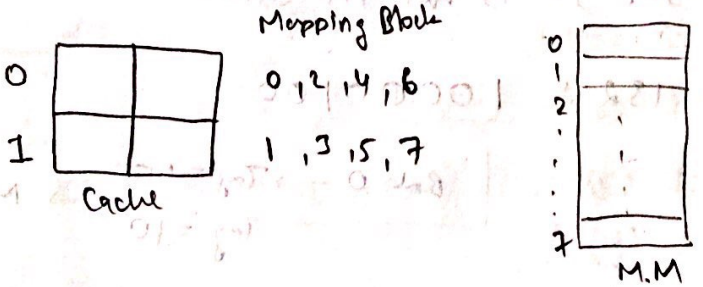


CPU Req. M.M block No.	Maps	Hit or Miss	Replacement
1 →	1	Miss	Bring B1	No
5 →	1	Miss	Bring B5	B1
1 →	1	Miss	Bring B1	B5
5 →	1	Miss	Bring 5	B1
1 →	1	⋮	⋮	⋮
5				

Here in direct mapping if request is generated by CPU as above then every time it leads to miss which consumes more time in replacement.

To Improve performance we design cache in n-way set associative.

In 2-way set associative Mapping



M.M block req : 1, 5, 1, 5, 1, 5

Miss Miss All Hit

In set associative mapping each cache line can be hold multiple ~~one~~ blocks at a time.

$$\text{No. of index or No. of sets} = \frac{\#\text{cache block}}{k\text{-way}}$$

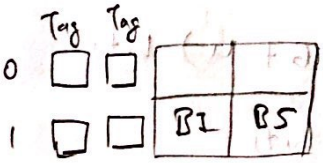
k = no. of main memory block are possible in each cache line.

Main Memory Address

Tag	set offset	Byte offset

Mapping function = M.M block no. % no. of set in cache

No. of bits in set offset = $\log_2$ (#sets in cache)

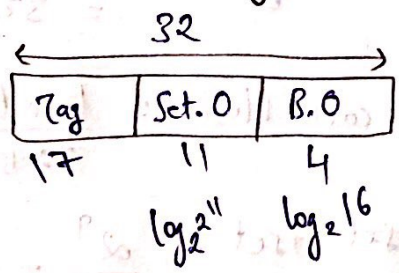


For every block we required tag.

∴ Tag directory size = #cache block * (Tag + Extra bits)

Ex: M.M address = 32-bits
Cache size = 128 kB
block size = 16 Byte
4-way set associative cache
Size of tag directory.

→



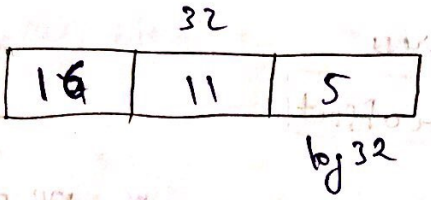
$$\#\text{block in cache} = \frac{128\,kB}{16\,B} = 2^{13}$$

$$\text{No. of set} = \frac{2^{13}}{2^2} = 2^{11}$$

∴ Size of tag directory = $2^{13} * 17$
= 17 x 8 K bits
= 136 k bits

Q-7 Cache size = 256 KB, 4-way set associative
Block size = 32 Byte
M.M add = 32-bits
Tag directory entry contains in addition to tag, 2 valid, 1 Modi., 1 Replacement bit.
#bits in tag field ?
Cache tag memory size ?

→



#cache line = $\frac{2^8 \times 2^{10} B}{2^5 B}$

$= 2^{13}$

No. of set = $\frac{2^{13}}{2^2} = 2^{11}$

tag field = 16 bits

tag memory size = $2^{13} * (16+4)$ bits

$= 20 \times 8$ K bits

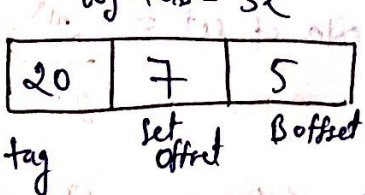
$= 160$ K bits

Q-9 4-way set-associative cache memory unit with capacity of 16 KB is built using block size of 8 words. The word length is 32-bits. The size of Physical add. space is 4 GB. The no. of bits for the TAG field is ___.

→

block size = 8×32 bits $= \frac{2^8 \text{ bits}}{2^3} = 2^5$ Byte

$= 32$ Byte



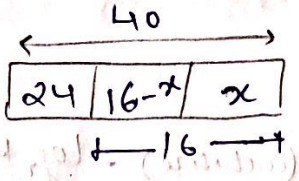
#cache block = $\frac{16 KB}{32 B} = 2^9$

#set = $\frac{2^9}{2^2} = 2^7$

TAG field = 20 bits

Q-22 The width of physical address on machine is 40 bits. The width of tag field in 512 kB 8-way set associative cache is ___ bits.

→



~~#cache size~~

~~#Blocks~~ $\#\text{set} = \frac{512\text{ kB}}{8} = 2^{16}$

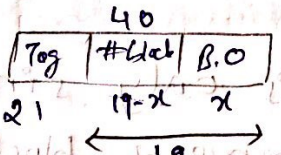
$\frac{2^{19}}{2^3}$

Let B. size $= 2^x$

$\#\text{cache line} = 2^{19-x}$

$\#\text{set} = \frac{2^{19-x}}{2^3} = 2^{16-x}$

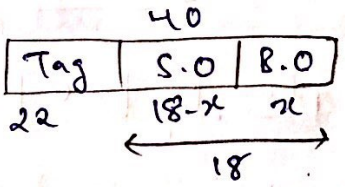
Direct Mapping



$\text{Block size} = 2^x$

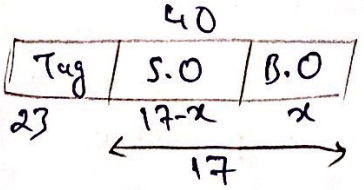
$\#\text{Block} = \frac{2^{19}}{2^x} = 2^{19-x}$

2-way set associative



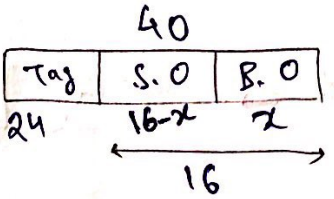
$\#\text{sets} = \frac{2^{19-x}}{2^1} = 2^{18-x}$

4-way set associative



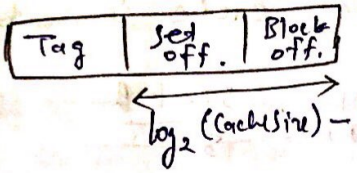
$\#\text{sets} = \frac{2^{19-x}}{2^2} = 2^{17-x}$

8-way set associative



$\#\text{sets} = \frac{2^{19-x}}{2^3} = 2^{16-x}$

For k-way Set Associative

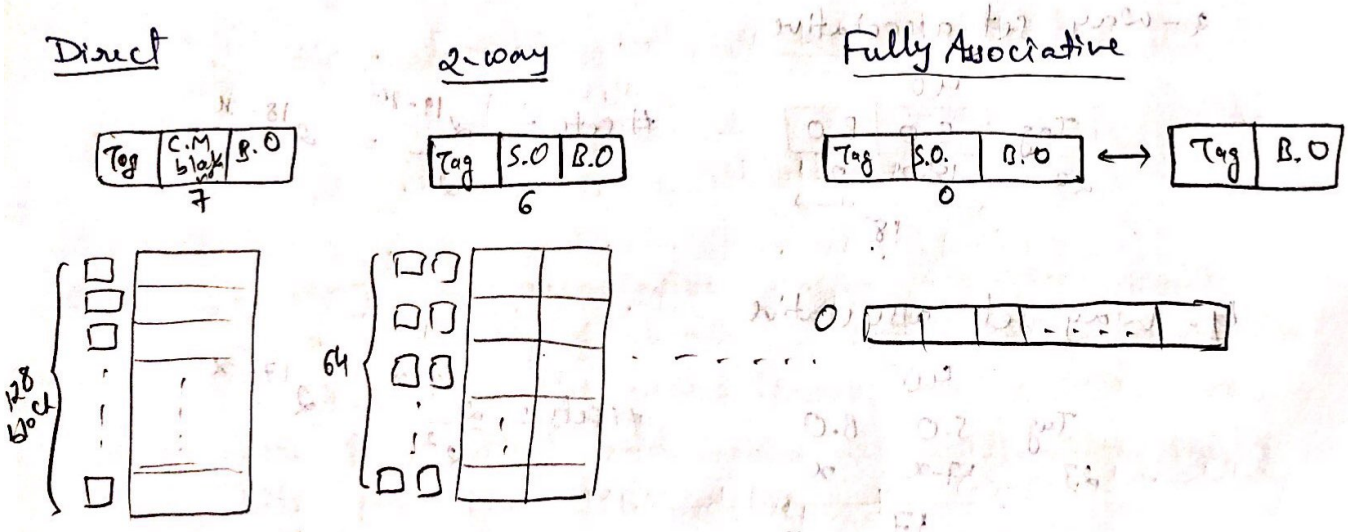


$\therefore$ Tag = M.M add size - $(\log_2(\text{cache size}) - \log_2 k)$

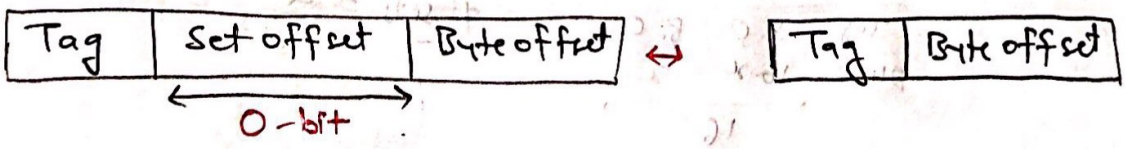
= M.M add size - $\log_2(\text{cache size}) + \log_2 k$

* Fully Associative Memory

In this mapping there is only single block on which all the main memory blocks mapped.



→ There is no index bit (set offset) in fully associative



No. of set = 1

$\therefore$ set offset is 0, $2^0 = 1$

Q. A fully Associative cache contains 2^{10} blocks the M.M has 2^{16} blocks of the same size as the cache. The size of memory required at cache controller to store the cache directory?

→

2^{10} cache block

2^{16} M.M block

$2^x = \frac{\text{cache size}}{2^{10}}$

$2^x = \frac{\text{M.M size}}{2^{16}}$

~~In Direct Mapping~~ M.M Address →

M.M block no.	Byte offset

In Fully Associative

Tag	Byte offset

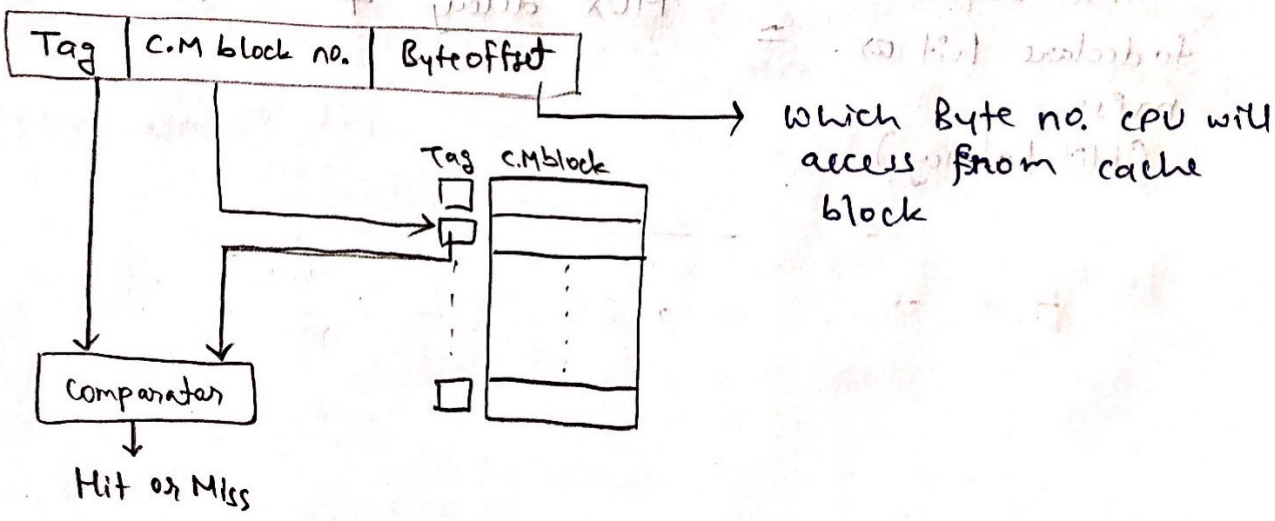
@ 16

∴ Size of tag directory = $2^{10} * 16$ = 16 k bits

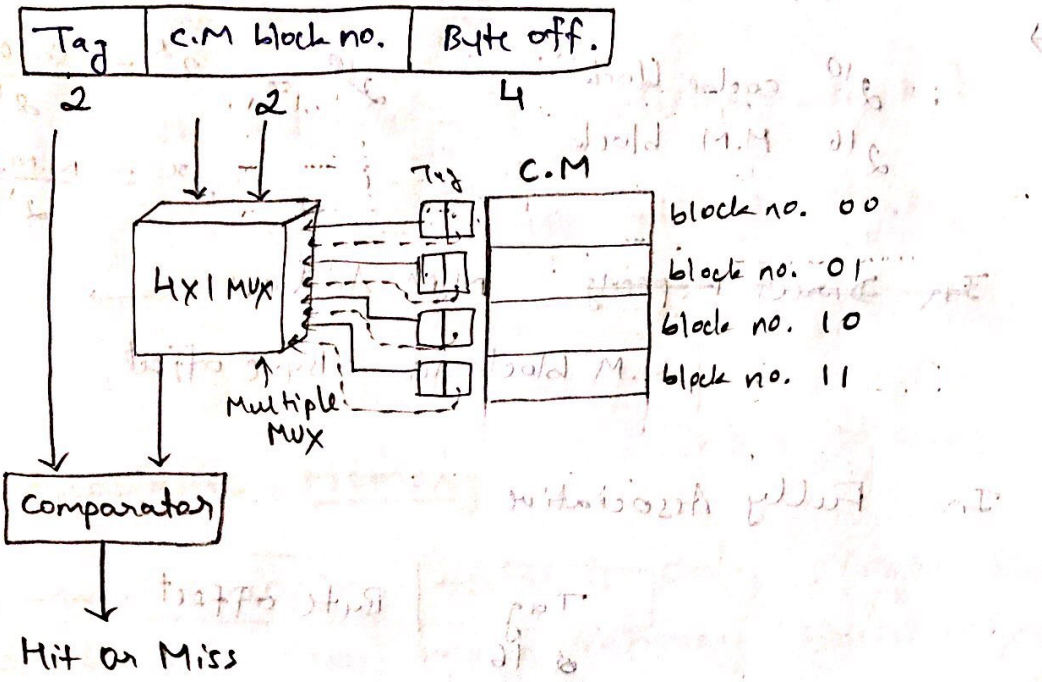
⇒ In fully associative memory Tag field is nothing but main memory block no.

* Hardware Implementation of Mapping

I. Direct Mapping



Ex: No. of ~~bit~~ blocks in Cache Mem. = 4
Main. Mem. address = 8-bits
Block size = 16 Byte

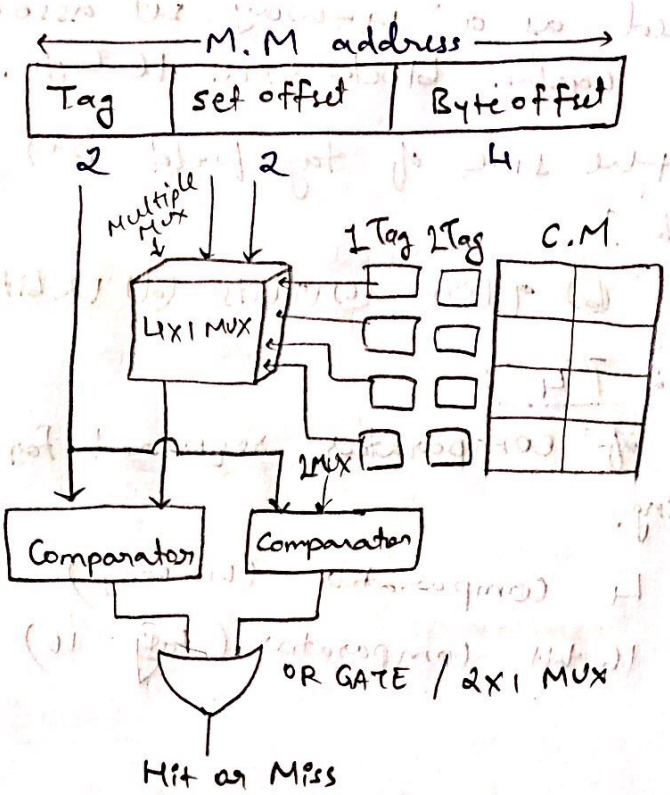


MUX can forward only 1-bit of tag, so we required multiple MUX

No. of Mux = No. of tag-bits
Size of MUX = No. of cache blocks X 1
Size of comparator = No. of tag bits.

Time required to declare hit or miss (Hit latency) = MUX delay + Comparator delay

II. 2-Way Set Associative



→ 2-Way
→ Cache block = 8
→ No. of set = 4 = 2^2
→ set offset = 2

For k-way set associative

No. of MUX = k * No. of tag-bits

Size of MUX = No. of sets in cache x 1

Size of comparator = no. of tag-bits

no. of Comparator = k

MUX to say hit or miss = $1^{k} \times 1$ (Size)

Time required to declare hit or miss (Hit Latency) = MUX delay + Comparator delay + k to 1 MUX delay

Q-1 CPU has 32-bit memory add. and 256 kB Cache organized as a 4-way set associative cache with cache block size 16 byte.

(i) what is the size of tag field per cache block?

(a) 16 bits (b) 9 bits (c) 19 bits (d) 12 bits

32

16	12	4

(ii) no. & size of comparator required for tag matching.

→ 4 comparator (4-way)

16 bit comparator (tag = 16)

Q-11 1st cache → 32 kB, 2-way set ass.

Block size = 32 B

2nd cache → same size but direct mapped

Address = 32 bits.

2×1 MUX latency = 0.6 ns

K-bit comparator latency = K/10 ns

Hit latency of set associative is h_1

Hit latency of direct mapped is h_2

$h_1 = ?$, $h_2 = ?$

→ 2-way set

32

18	9	5

Direct mapped

17	10	5

~~# MUX = 2^9 × 1~~

~~= 256~~

~~# MUX~~

h_1 = Comparator delay + 2x1 MUX delay

= $\frac{18}{10}$ + 0.6

= 2.4 ns

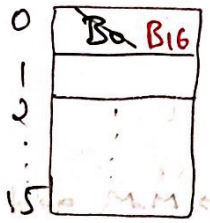
h_2 = Comparator delay

= $\frac{17}{10}$ ns

= 1.7 ns

* Cache Block Replacement

Direct



0, 16

CPU Request M.M block no. = 0, 16

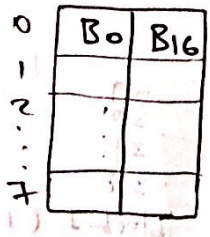
↓

Replace block 0, because No any other choice.

→ In Direct mapping, No replacement policy needed.

Set Associative

2-Way



0, 16, 32

M.M block request = 0, 16, 32

↓

which block is replaced

↓

Selected by block replacement Policy

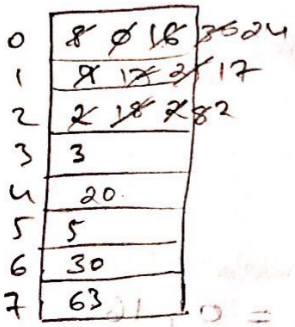
(i) FIFO
(ii) Optimal
(iii) LRU

Q- Consider direct mapped cache with 8 cache blocks (0-7) if M.M block request are in order,

3, 5, 2, 8, 0, 63, 9, 16, 20, 17, 25, 18, 30, 24, 2, 63, 5, 82, 17, 24

which of the following block will not be in cache at the end of sequence?

(a) 3 (b) 18 (c) 20 (d) 30



Q-20 Byte add. memory of 2^{16} B → M.M add 16-bit

Direct Mapped cache

No. of lines = 32

block size = 64 Byte

50×50 2D array of bytes ⇒ 2500 elements

⇒ 2500 Bytes

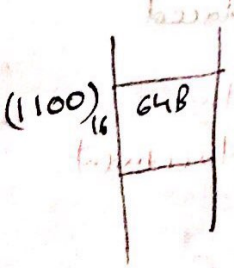
Base address = 1100 H

Array access twice

→

M.M add. = 16

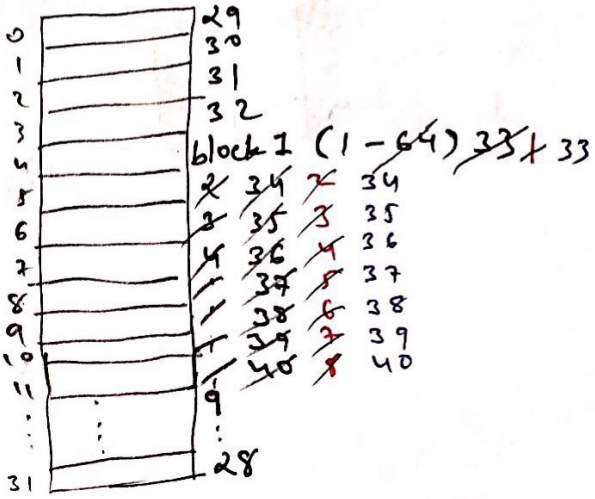
Tag	C.M Block no.	B.O
5	5	6



0001 001 000000

↓

Block No. - 4



No. of blocks required for array $= \frac{2500B}{64B}$

$\cong 40$

After block 32, from 33 to 40 will be miss & replace at block no 4 to 11.

In 2nd access, block 1 to 8 will be miss & will replace block no. 33 to 40. = 8 miss

From block no. 9 to 32 will be hit.

From block 33 to 40 will be miss & replace the block no. 1 to 8. = 8 miss

Total miss = 40 + 8 + 8

= 56

No. of blocks required for array $= \frac{2500B}{64B}$

$\cong 40$

After block 32, from 33 to 40 will be miss & replace at block no 4 to 11.

In 2nd access, block 1 to 8 will be miss & will replace block no. 33 to 40. = 8 miss

From block no. 9 to 32 will be hit.

From block 33 to 40 will be miss & replace the block no. 1 to 8. = 8 miss

Total miss = 40 + 8 + 8

= 56

Q- A CPU has 32KB direct mapped cache with 128B block size. Suppose 'A' is a two D array of size 512×512 with elements that occupies 8 B each. Consider C code segments P_1 & P_2.

```
P1: for (i=0 ; i<512 ; i++)
    { for (j=0; j<512 ; j++)
        { x+= A[i][j];
    } }

P2: for (i=0 ; i<512 , i++)
    { for (j=0 ; j<512 ; j++)
        { x+= A[j][i];
    } }
```

P_1 & P_2 are executed independently with same initial state, namely the Array A is not in the cache & i, j, x in registers. Let no. of cache misses experienced by P_1 be m_1 & For P_2 be m_2.

(i) $m_1 = ?$

(ii) $\frac{m_1}{m_2} = ?$

Cache size = 32 kB

Block size = 128 B

Array A = 512 × 512 = 262144 elements

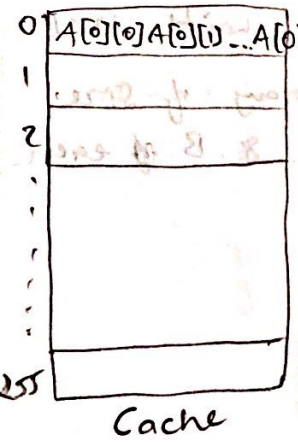
Element size = 8 B

$$\#\text{Cache line} = \frac{32\text{kB}}{128\text{B}} = 2^8 = 256 \text{ lines}$$

$$\text{No. elements in 1 cache line} = \frac{128\text{B}}{8\text{B}} = 16 \text{ elements}$$

Default storage is row major in memory.

(i) In P_1, array is accessed in row major.



For 1st element whole block will bring in to cache i.e., (16 elements) (1 miss)

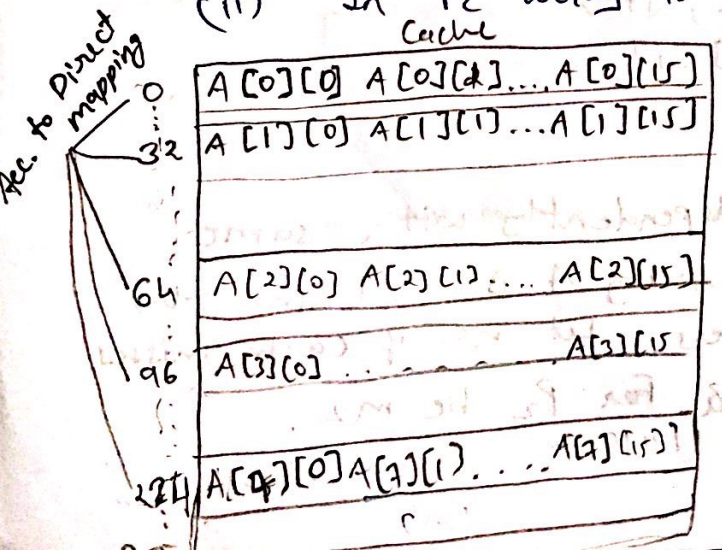
For next 15 element it will be hit.

$$\therefore \text{Total miss} = \frac{512 \times 512}{16}$$

$$= 16384 \text{ miss}$$

$$\therefore m_1 = 16384$$

(ii) In P_2 array is accessed in column major



For 1st iteration we required A[0][0] which is miss & 16 elements will be bring from memory to cache

For 2nd iteration we required A[1][0] which is miss & 16 elements will be bring from memory to cache

∴ In column major access ~~add~~ in every iteration there will be miss always.

∴ Total miss = 512×512

$= 262144$ misses

∴ $M_2 = 262144$

$$\therefore \frac{M_1}{M_2} = \frac{16384}{262144} = 0.0625$$

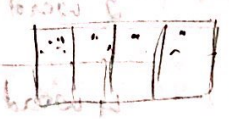
2014

Q- An access sequence of cache block addresses is of length ~~it~~ N and contains 'n' unique block addresses. The no. of unique block addresses between two ~~consey~~ consecutive accesses to the same block addresses is bounded above by k. What is the miss ratio if the access is passed through a cache of associativity $A \geq k$. Exercising LRU policy.

(a) n/N (b) 1/N (c) 1/A (d) k/n

block address = N

'n' unique block add. after k add.

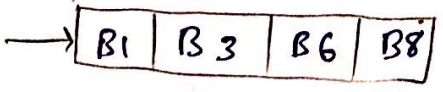


1 3 6 3 6 1 3 8 9 3 1

k=4 no. of unique block access between two consecutive access of a block, should be less then k.

atleast 4-way set associative cache min 'n' no. of misses will occur definitly

Assume all blocks map to same set



there will not be any cache miss because of replacement

∴ n/N

* Cache Miss Penalty

The amount of time CPU spends to service the cache miss

Time required to bring the missed block from main memory to cache memory

Assume,

No. of cycles needed to send add. to mm = 1
No. of cycles needed to access m.m = 10
No. of cycles needed to transfer 1 cell data to cache = 1

Cycle ≅ C.P.U cycle

Cache block size	M.M cell size	cache Miss penalty (send add. / 4 block access / 4 block transfer)	
4 words	1 word	1 + 4*10 + 4*1	= 45 cycle
4 words	2 word	1 + 2*10 + 2*1	= 23 cycle
4 words	4 word	1 + 10 + 1	= 12 cycle

if CPU clock rate is given,

$$1 \text{ cycle time} = \frac{1}{\text{clock rate}}$$

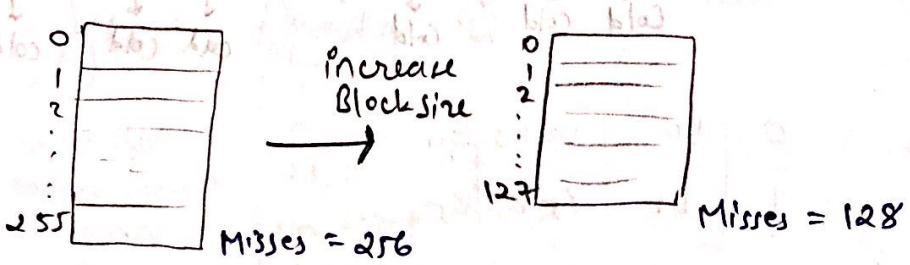
∴ time = no. of cycle * 1 cycle time

→ CPU sends address to memory only once

→ In one access of main memory 1 cell data can be accessed.

* Types of cache Misses:

1. **Cold or Compulsory Miss**: Cache miss occurs due to first access of any block, then it will be compulsory Miss.



→ To reduce the compulsory Misses increase the block size, which leads to less no. of cache line. Therefore no of misses will be less.

2. **Capacity Miss**: Cache miss occurs when the cache memory is full, then that is known as capacity miss.
[It should not be a compulsory miss]

→ To reduce miss then increase capacity of cache.

3. **Conflict Miss**: The cache miss occurs due to mapping constraints. Or the cache miss occurs due to tag mismatch.

→ It should not be compulsory miss and Capacity miss.

→ Conflict miss occurs when cache is not full, but 2 blocks mapping to same cache line.

Example : 2 sets
4 blocks
LRU replacement, initially cache empty.

M.M. block req. = 0 4 0 8 0 4 1 5 1 3 1 5
Cold Cold Hit Cold Hit Conflict Cold Cold Hit Cold Hit Capacity

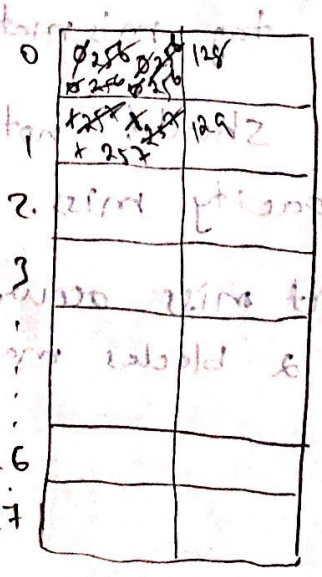
0	B_0	B_4 B_8
1	B_1	B_5 B_3

Q-23 Consider 2-way set associative cache with 256 blocks and uses LRU replacement. Initially the cache is empty. Conflict misses are those misses which occur due to contention of multiple blocks for same cache set. Compulsory misses occur due to first time access to block.

(0 128 256 128 0 128 256 128 1 129
257 129 1 129 257 129)

This sequence is repeated 10 times. The no. of conflict misses experienced by cache is?

→ No. of blocks = 256
2-way set associative
#set = $\frac{256}{2} = 128$



	1st access	2nd access	3rd access
0	Compulsory	Conflict	
128	Compulsory	Hit	Same
256	Compulsory	Conflict	as
128	Hit	Hit	Previous
0	Conflict	Conflict	in
128	Hit	Hit	remaining
256	Conflict	Conflict	8 access
128	Hit	Hit	
1	Compulsory	Conflict	
129	Compulsory	Hit	
257	Compulsory	Conflict	
129	Hit	Hit	
1	Conflict	Conflict	
129	Hit	Hit	
257	Conflict	Conflict	
129	Hit	Hit	

Compulsory = 6
Conflict = 4

Conflict = 8

Total conflict misses = 8 * 9 + 4
= 72 + 4
= 76

Total compulsory miss = 6

NOTE: There can not be any conflict miss in fully associative cache. Because there is only one set in fully associative cache, so block will be replace only when set is full.

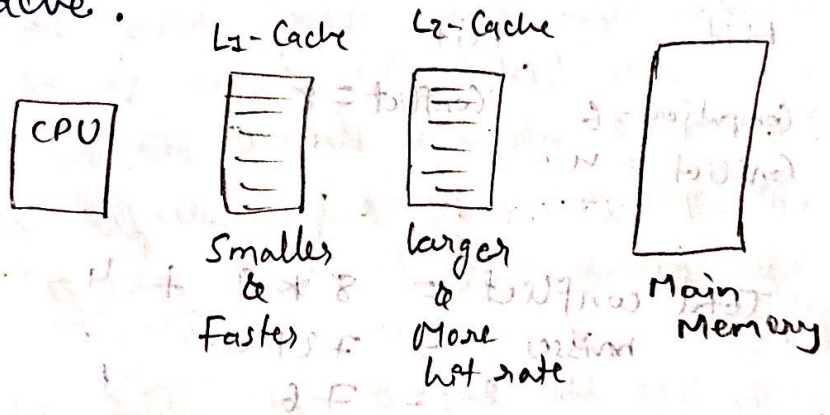
* Goal of Using Cache

1. Minimum average memory access time
2. Maximum cache hit rate
3. Minimize cache miss penalty

smaller cache, larger cache → Multi level cache

→ Non blocking cache, Multi-port or multibank cache

→ To minimize average memory access time we require small cache memory size. And To maximize cache hit rate we require larger cache memory. So to achieve both we implement multi level cache.



→ To minimize cache miss penalty we use technique like Non-blocking cache, multiport or multibank cache.

Multilevel Cache

Simultaneous Access:

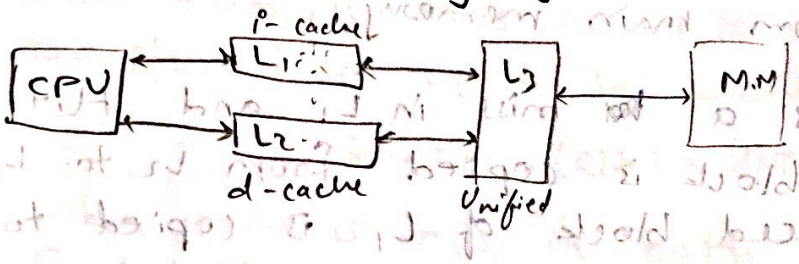
$$t_{avg} = H_1 * t_1 + (1 - H_1)\left[H_2 * t_2 + (1 - H_2)\, t_{m.m}\right]$$

Hierarchical Access:

$$t_{avg} = H_1 * t_1 + (1-H_1)[H_2 * (t_1 + t_2) + (1-H_2)(t_1 + t_2 + t_{m.m})]$$

$$= t_1 + (1-H_1)[t_2 + (1-H_2) t_{m.m}]$$

Q- Consider a multilevel memory Hierarchy. The hit ratio of level L_1, L_2 & L_3 and M.M are 0.8, 0.85, 0.9, 1.0 respectively. The access time of L_1, L_2, L_3 & M.M are 10 ns, 10 ns, 50 ns & 500ns. Among total memory references 60% are data access. What is the avg access time of the memory system.



$$t_{avg_{inst}} = 0.8 * 10 + 0.2(0.9 * (10+50) + 0.1(10+50+500))$$

$$\approx 30 \text{ ns}$$

$$t_{avg_{data}} = 0.85 * 10 + 0.15(0.9 * (10+50) + 0.1(10+50+500))$$

$$= 25 \text{ ns}$$

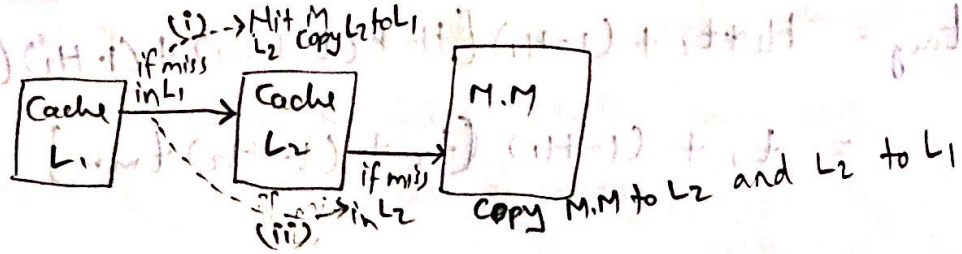
$$t_{avg} = 0.6 * t_{avg\ data} + 0.4 * t_{avg\ inst}$$

$$= 0.6 * 25 + 0.4 * 30$$

$$= 15 + 12$$

$$= 27 \text{ nsec}$$

* Inclusion Vs. Exclusion



Inclusion: Whatever content present in cache L_1 is also present in L_2. So content will be copied from L_2 to L_1.

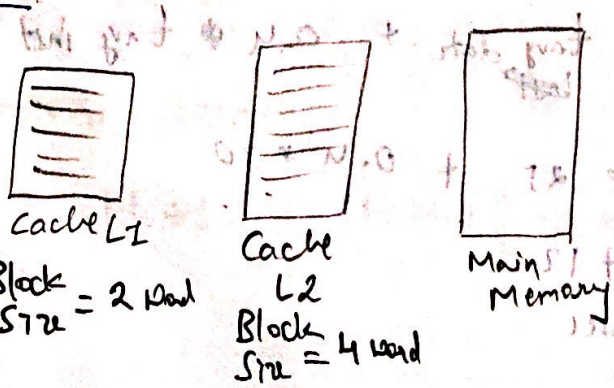
Exclusion: Content present in cache L_1 is no necessarily present in cache L_2. So content in L_1 will be bring from main memory.

→ If there is a miss in L_1 and Hit in L_2 then a block is copied from L_2 to L_1 and replaced block of L_1 is copied to L_2.

→ If there is a miss in L_1 and L_2 both, then a block is copied from main memory to L_1 and replaced block of L_1 is copied to L_2.

→ L_2 cache is also known as victim block.

Inclusion :-



if a block copied from main memory to L2 ⇒ Size of block in L2 (4 words)

if block copied from L2 to L1 ⇒ size of block in L1 (2 words)

10

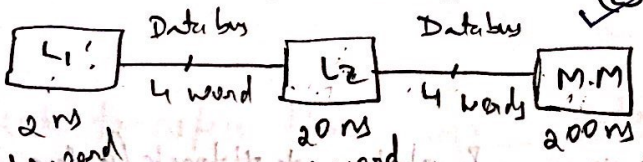
Q Computer system has an L1 & L2 cache & M.M unit, Connected as shown below. Block size in L1 is 4 word, block size in L2 is 16 words the memory access time are 2ns, 20 ns, & 200 ns

(i) When there is miss in L1 & hit in L2 a block is transfer from L2 to L1 cache. Time taken?

(a) 2ns (b) 20 ns (c) 22 ns (d) 88 ns

(ii) When there is miss in both L1 & L2 cache. First a block is transfered from main memory to L2 cache & then L2 to L1. Time taken?

(a) 222 ns (b) 888 ns (c) 902 ns (d) 968 ns



(i) We have to take only transfer time & checking of miss in L1 & Hit in L2 is already done.

∴ Transfer time = L2 → L2

= 20 + 2

= 22 ns

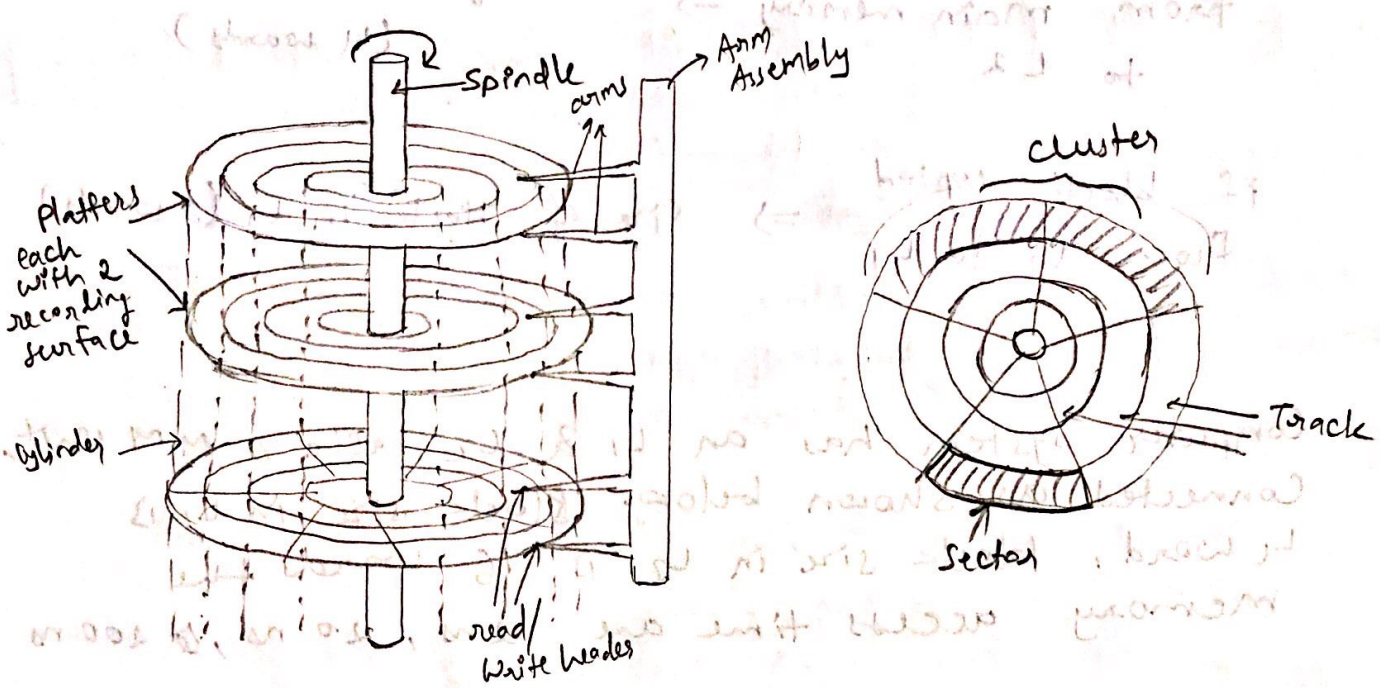
(ii) From M.M to L2 = 4*(200+20) = 880

From L2 to L1 = 22 ns

∴ Transfer time = 880 + 22

= 902 ns

* Secondary Memory



(Default)
Constant sector capacity or Variable storage density

Variable sector capacity or constant storage density

Size of Disk : 2 * no. of platers * #tracks/surface * #Sectors/tracks * Sector capacity

→ Sector is the smallest unit, which can be read or written at once.

→ Every sector has one address

Disk Access Time:

$$\text{Disk Access time} = \text{Seek Time} + \text{Rotational latency} + \text{1 sector transfer time} + \text{disk controller delay} + \text{Extra given time}$$

Seek Time: Time required to move the arm to desired track.

Rotational latency: Time required to rotate the desired sector under read/write header.

1 sector Transfer time: Time required to read/write sector

$$\text{Avg. Rotational latency} = \frac{\text{Rotational latency}}{2}$$

NOTE: In one rotation entire track can be transfered

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

Q- Consider a disk with 16 platters, 2 surfaces/platter, 1k track/surface, 2k sector/track & 2048 Byte/sector. Disk rotates with 3000 rpm. Seek time = 10 ms.

(i) Capacity of Disk in GB ?
(ii) #bits for disk addressing ?
(iii) Find 1 Disk access time.
(iv) Disk data transfer rate.

platter = 16 , 2 surface / platter
1k track / surface
2k sector / track
2048 Byte / sectors.
3000 rpm.
Seek time = 10 ms

(i) Capacity = $2^4 \times 2 \times 2^{10} \times 2 \times 2^{10} \times 2^{11}$

$= 2^{37}$

$= 128$ GB

(ii) #bits = $\log_2 2^{26} = 26$ bits

(iii) rotational latency = $\frac{60}{3000} = 0.02$ sec

$= 20$ msec.

Avg. rotational latency = 10 msec

1 track 20 msec

1 sector ?

$\frac{1 \text{ sec} \times 20 \text{ msec}}{2000 \times 2048 \text{ sector}} = 0.01$

1 Disk Access time = 10 ms + 10 ms + 0.01

= 20.01 ms

(iv) 20 msec 2048×2048

1 sec ?

$\frac{2048 \times 2048}{0.02 \text{ sec}} = 209.71$ MB/s

$2^{11} \times 2^{10} \times 100 = 200$ MB/s

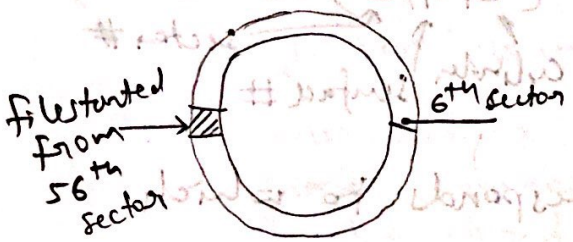
If 'n' no. of sector are to be transfer from the disk.

→ If sectors are stored sequentially,

$$\text{Disk Access time} = \text{Seek time} + \text{Avg Rotational latency} + n * \text{1 sector transfer time}$$

→ If sectors are stored randomly,

$$\text{Disk Access Time} = n * \left[\text{Seek time} + \text{Avg rotational latency} + \text{1 sector transfer time} \right]$$



500 sector in 1 track

10 msec is 1 rotational time

current head position = 6th sector

Sector to Transfer = 56th sector

∴ Rotational latency = time req. to rotate 50 sector only

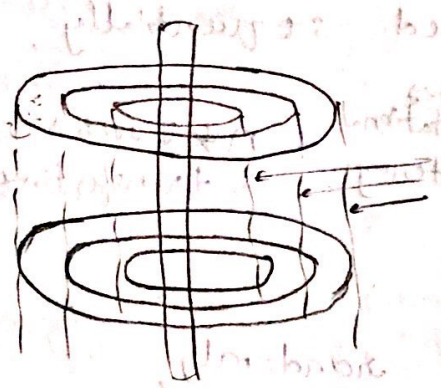
$$= \frac{10 \text{ ms}}{500} \times 50 = 1 \text{ msec}$$

Cylinder: To save seek time.

→ Respective track of every surface makes cylinder.

#Cylinder on disk = #track per surface

Collection of the tracks of the same radius from all the surfaces form a cylinder.



Q-3

63 sector / track
10 platters with 2 surface
1000 cylinders.
Address of sector is (c, h, s)
c → cylinder, h → surface #, s → sector #

(i) (400, 16, 29) corresponds to which sector no.

sectors in 400 cylinder (0 to 399) = 400 * 20 * 63
= 504000

sectors in 16 surface = 16 * 63 = 1008 (0 to 15)

Now add. of (400, 16, 29) = 504000 + 1008 + 29
= 505037

(ii) Add of 103.9th sector is,
surface required = $\frac{1039}{63}$ = 16.49
∴ We require 16 surface
∴ # sector in 16 surface = 16 × 63
= 1008

Now remaining sectors are = 1039 - 1008
= 31

∴ address will be (0, 16, 31)

* Parallel Processing

Technique for simultaneous data processing.

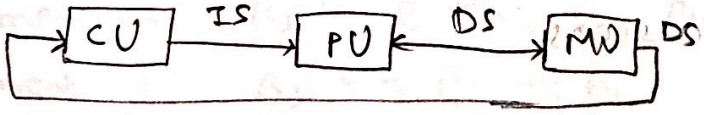
→ Array processing
→ Vector processing
→ Pipeline Processing

Flyn's Classification

(i) SISD

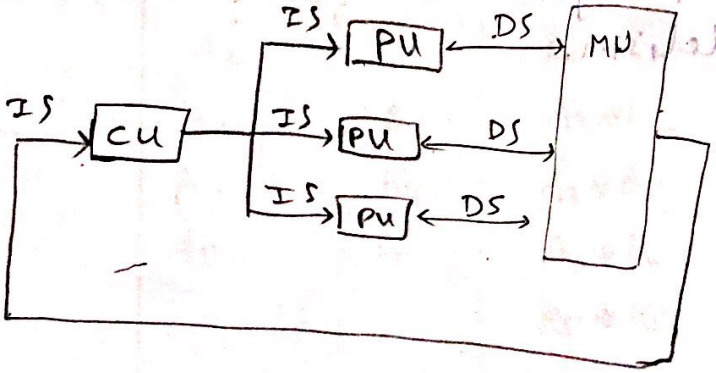
e.g. Von-Neumann

→ Instruction & data both are stored in same memory after the operation the result is also stored in same memory.
→ Sequencial execution



→ Single Instruction & Fetch & Single instruction execution.

(ii) SIMD :



→ Single instruction fetch in one cycle & multiple instruction execution.

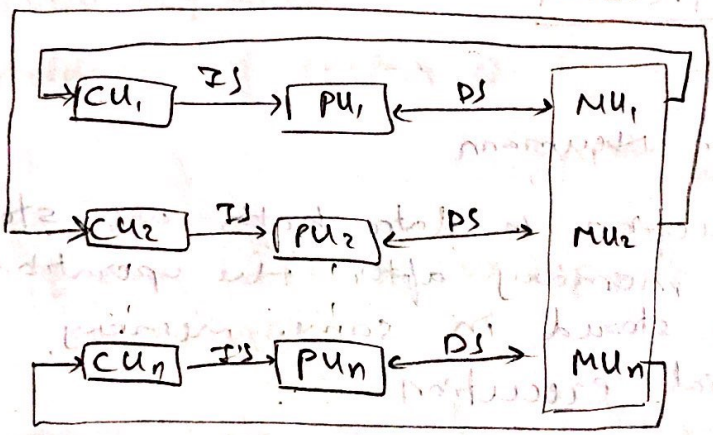
Eg. Pipeline processing

MISD:

→ Multiple instruction fetch in one cycle & single execution.

∴ It is not implemented.

MIMD:



Eg. Multiprocessors

→ Multiple instruction is fetched in single cycle & Multiple instruction execution.

→ Multiple pipeline

→ Such type of system is based on Instruction level parallelism.

* Pipeline Processing

It is technique to divide a sequential operation into sub operation.

Pipeline is applicable only when same processing is performed over multiple input.

Each sub operation is performed in a seperate unit known as segments. All the segment can perform their respective subop? parallely on different different inputs.

An operation performed in all the segments is known as a task.

Example : $A_i * B_i + C_i$

$i = 1$ to 5; A, B, C are memory operand.

Segment 1 : $R_1 \leftarrow A_i$; $R_2 \leftarrow B_i$

segment 2 : $R_3 \leftarrow R_1 * R_2$, $R_4 \leftarrow C_i$

Segment 3 : $R_5 \leftarrow R_3 + R_4$

segments / cycle	seg 1		seg 2		seg 3
	R_1	R_2	R_3	R_4	R_5
1.	A_1	B_1			
2.	A_2	B_2	$A_1 * B_1$	C_1	~~$A_1 * B_1 + C_1$~~
3.	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4.	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5.	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6.			$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7.					$A_5 * B_5 + C_5$

#task	#cycle in pipeline	#cycle in sequencial system
n=5	7	3*5 = 15
n= 6	8	3*6 = 18

→ Pipeline accept the new input before completion of the old input that means the new input are executed along with the old input. Therefore, pipelining allows the overlaping execution.

* General Consideration about Pipeline

Cycle time in Pipeline: Amount of time in which all the segments can perform their respective suboperation

Consider k-segment pipeline with cycle time, t_p to perform n-task.

Time required to perform 1st Task : $k * t_p$

Time required to perform remaining task : $(n-1) t_p$

$(n-1)$

∴ Execution time of pipeline = $[k + (n-1)] t_p$

Consider non-pipeline system, which takes t_n time to perform one task.

∴ Execution time of non-pipeline system $= (n * t_n)$

Performance of pipeline

$$\text{Speed up} = \frac{\text{Performance of Pipeline}}{\text{Performance of non-pipeline}}$$

$$= \frac{\frac{1}{\text{E.T pipeline}}}{\frac{1}{\text{E.T non-pipeline}}}$$

$$= \frac{\text{E.T non-pipeline}}{\text{E.T pipeline}}$$

$$\text{Speedup } (S) = \frac{n t_n}{[k + (n-1)] t_p}$$

when $n >> k$

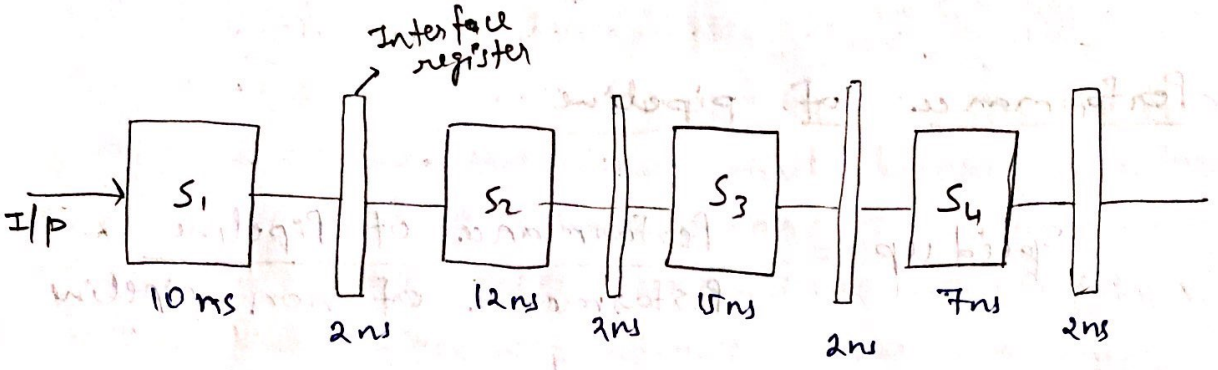
$$S_{\substack{\text{ideal} \\ (\max)}} = \frac{n t_n}{n t_p} = \frac{t_n}{t_p}$$

→ If time required to perform one task in pipeline and non-pipeline system are same,

$$t_n = k * t_p$$

$$\therefore S_{ideal} = \frac{k t_p}{t_p}$$

$$\therefore S_{ideal} = k$$



→ when buffer delay is included

$$t_p = \max(\text{segment delay}) + \text{intermediate buffer delay}$$

$$\therefore t_p = \max(10, 12, 15, 7) + 2\text{ ns}$$

$$= 15 + 2$$

$$= 17\text{ ns}$$

→ If setup time overhead is included

$$t_p = t_p + \text{setup time/skew time}$$

Q- A non pipeline system takes 50 ns to process a task. The same task can be processed in 6 segment pipeline with cycle time of 10 ns. Speedup ? for pipline for 100 task.

→ Speedup $= \frac{100 * 50\text{ ns}}{105 * 10\text{ ns}} = 4.76$

$$\text{Speedup (max)} = \frac{t_n}{t_p} = \frac{50\ ns}{10\ ns} = 5$$

Q-19 4-stage pipeline has stage delays of 150, 120, 160, 140 ns resp. Interstage buffer delay is 5 ns. Registers used constant clock rate. The total time taken to process 1000 data items on this pipeline is —

→ Total time req. = $(k + n - 1)\, t_p$

$= (4 + 1000 - 1)(160 + 5)$

$= 1003 * 165$

$= 165495\ ns$

$= 165.5\ \mu s$

Q-20 CPU takes 12 cycle to complete an instn. The corresponding pipelined CPU uses 6 stage with execution of 3, 2, 5, 4, 6 and 2 clocks. What is the speed up assuming that a very large no. of instn are to be executed?

→ $$S_{ideal} = \frac{t_n}{t_p}$$

$$= \frac{12}{6}$$

$$= 2$$

Q-19 We have 2 design D_1 & D_2 for synchronous pipelined CPU. D_1 has 5 stages with execution time 3ns, 2ns, 4ns, 2ns, 3ns. While D_2 has 8 pipeline stage each with 2ns execution time. How much time can be saved using D_2 over D_1 for executing 100 instⁿ.

→

Execution Time, D_1 = $(5+99) * (4)$

= 416 ns

Execution Time, D_2 = $(8+99) * (2)$

= 214 ns

Saved time = 416 − 214

= 202 ns

Q. Consider a 4-segment pipeline with segment delays 50 ns, 30 ns, 95 ns & 45 ns. The intermediate register delay is 5 ns.

(i) How long would it take to process 100 i/p. in this pipeline.

(ii) How can we reduce total time about $\frac{1}{2}$ of the time calculated in part 1.

→ (i) reg. time = $(4+99) * (95+5)$

= ~~9385~~ ns 103 * 100

= 10300 ns

(ii) $\frac{1}{2} \times 9785 = (k+n-1)\,tp$

$4892.5 = (4+99) * (tp+5)$

$\frac{4892.5}{103} = (tp+5)$

$\therefore 47.5 = tp + 5$

$\therefore 42.5 = tp$

S_1	S_2	S_3	S_4
50	30	95	45

S_{31} S_{32}
50 45

} We can increase the stage while maintaining equal stage delay to increase the performance

Q.13 Consider a non-pipelined processor with a clock rate of 2.5 GHz and average cycle per instn of four. The same processor is upgraded to pipelined processor with five stages; but due to the internal pipelined delay, the clock speed is reduced to 2 GHz. Assume that there are no stalls in the pipeline. The speedup achived in this pipelined processor is ___

→ $k = 5$

$t_n = \frac{1}{2.5\ GHz} = 0.4\ ns = 0.4 \times 4 = 1.6$

$tp = \frac{1}{2\ GHz} = 0.5\ ns$

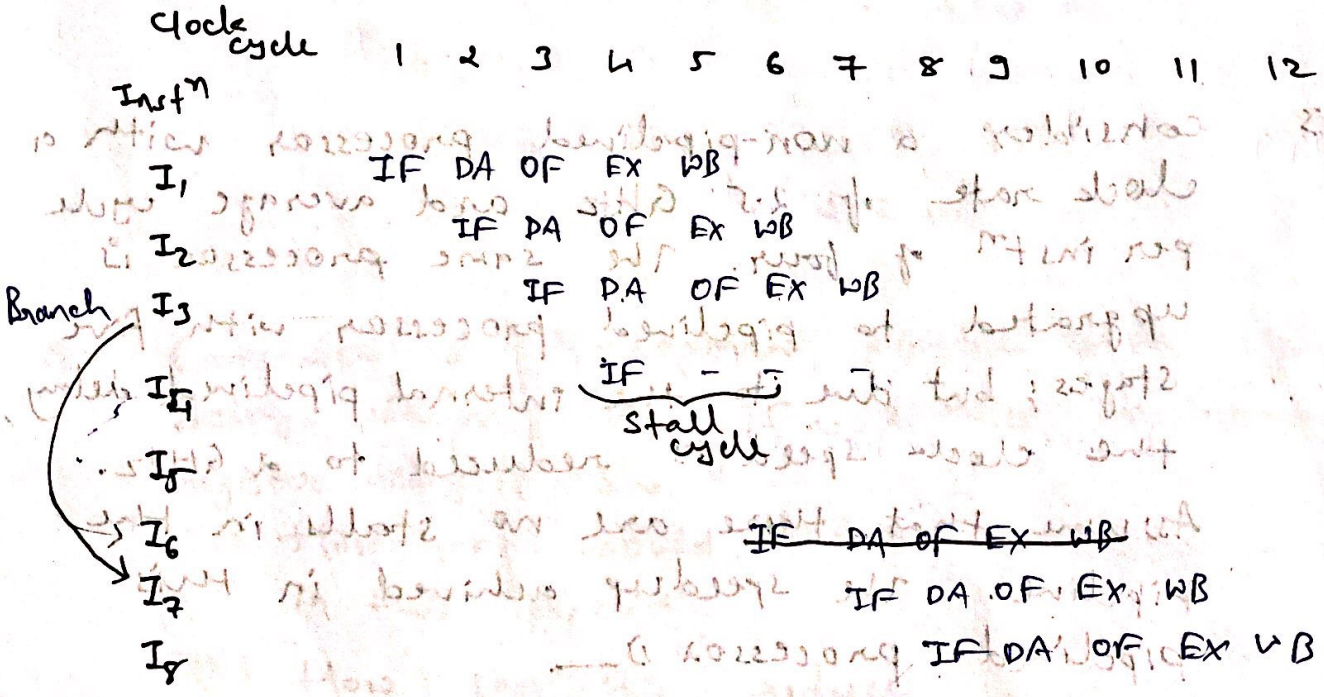
$Speedup = \frac{t_n}{t_p} = \frac{1.6}{0.5} = 3.2$

* Instruction Pipeline

→ Pipeline processing is implemented for instruction cycle.

→ Assume 5 segment pipeline as follows:

IF - Instruction fetch
DA - Decode & Address calculation
OF - Opcode fetch
EX - Execution
WB - Write Back



In cycle 4 CPU decodes that I_3 is branch instruction.

I_1 and I_2 will be completely executed.
Remove all instruction (I_4) from pipeline, fetched after I_3.

→ After EX phase of I_3, branch result (target add.) will be available [in cycle no. 6] ⇒ In cycle no. 7 next inst^n is fetched.

NOTE: After execution phase of the branch inst^n, target Address is available.

Actually no. of instructions executed = 6 (n)

$k = 5$

No. of cycles req. in usual = $k + n - 1$

$= 5 + 6 - 1$

$= 10$

Stall cycle because of branch inst^n = 3

Total no. of cycles = $10 + 3$

$= 13$

NOTE: If ~~after~~ after i^{th} stage of the branch inst^n executed then, no. of stall cycle is $(i-1)$

Stall Cycle: Extra cycle in the pipeline because of any hazard (problem).

Q- FI - 5ns
DI - 7ns
FO - 10ns
EX - 8ns
WB - 6ns
Storage buffer = 1ns

Total inst^n = 12
I_3 is branch inst^n & target is I_9.

Total cycle = (k + n-1) + stall cycle

Total instn will be execute = $(I_4 - I_1) + (I_9 - I_{12})$
= 8

$\therefore k + n - 1 = 5 + 8 - 1 = 12$

$\therefore$ Total cycle = 12 + 3
= 15

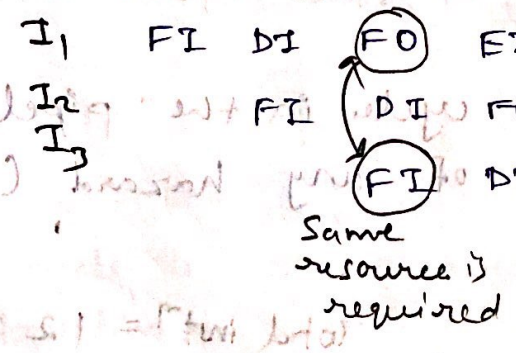
Total time req. = 15 * (10+1)
= 15 * 11
= 165 ns

* **Pipeline Hazard**: The reason because of which the smooth conduct of pipeline is disturbed.

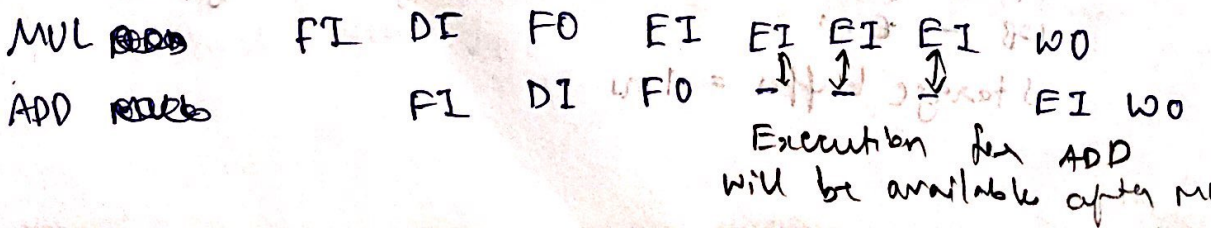
Types

(i) **Resource Conflicts**: If the same resource is required in two stages simultaneously.

Ex 1



Ex 2 I cycle in Execution → ADD
4 cycle in Execution → MUL



(ii) Data Dependency : If the result of an Instruction is used as input in the next instruction

Ex:

i : $R_1 \leftarrow R_2 * R_3$

$i+1$: $R_4 \leftarrow R_1 + R_6$

i : IF ID OF EX WB

$i+1$: IF ID — — OF

Can not perform OF as R_1 is available after WB

→ Regular pipeline can not detect the data dependency.

→ To minimize the no. of stall in pipeline due to data dependency

S/W Soln : Delayed load

H/W Soln : H/W interlock or By passing (operand forwarding)

Delay load : Compiler inserts no operation instructions are independent instn before the dependent instruction.

i : $R_1 \leftarrow R_2 * R_3$ IF ID OF EX WB

$i+1$: Nopn IF ID OF EX WB

$i+2$: Nopn IF ID OF EX WB

$i+3$: $R_4 \leftarrow R_1 * R_6$ IF ID OF EX WB

H/w interlock : Extra stall cycle created in execution by H/w.

IF ID OF EX WB

IF ID → (stall cycle added by H/w.) OF EX WB

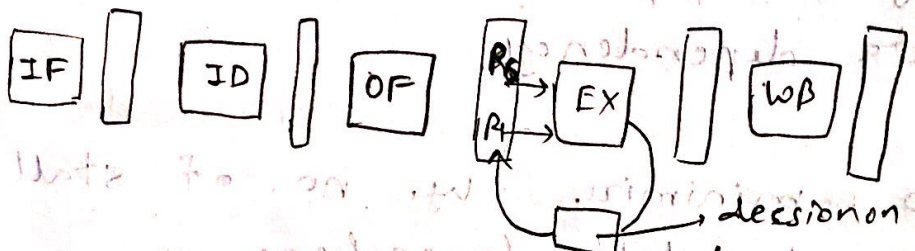
Operand forward :

$R_1 \leftarrow R_2 * R_3$

$R_4 \leftarrow R_1 + R_6$

IF ID OF EX WB

IF ID OF → EX WB



decision maker

Condition (because every instn we dont required operand forwarding)

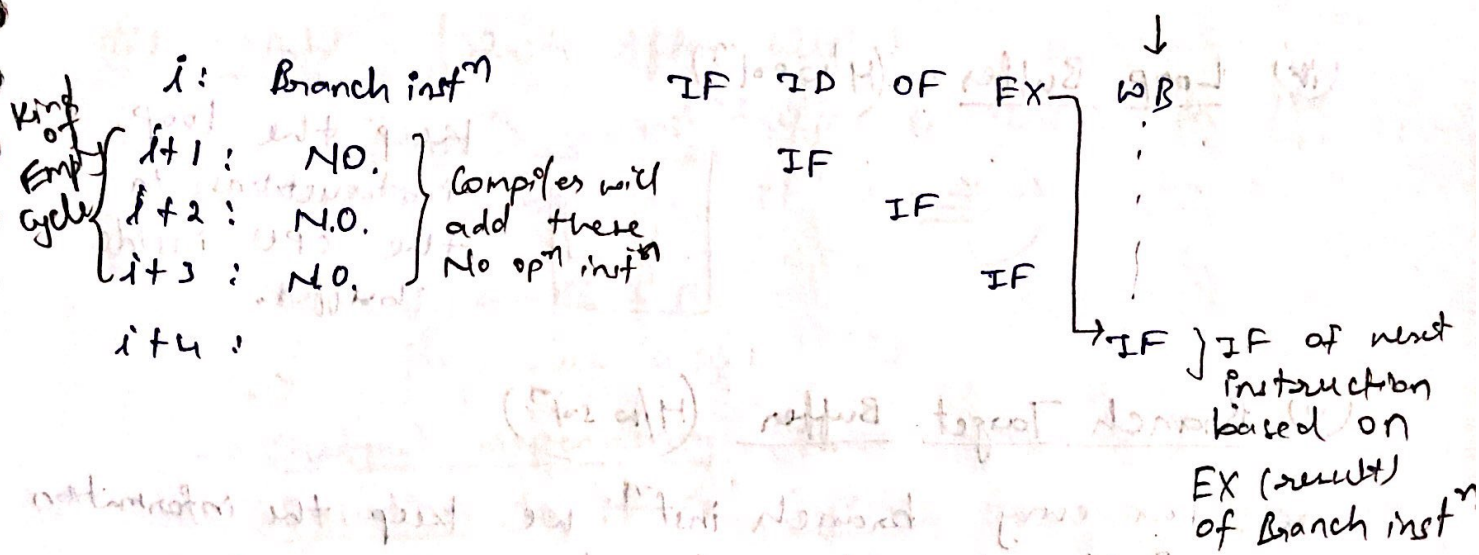
If operand forwarding is used then no any stall cycles for the data dependency.

(iii) Branch Difficulty / Control Dependency

It occurs because of branch instn

(i) Delayed branch (s/w soln)

→ Provided by compiler



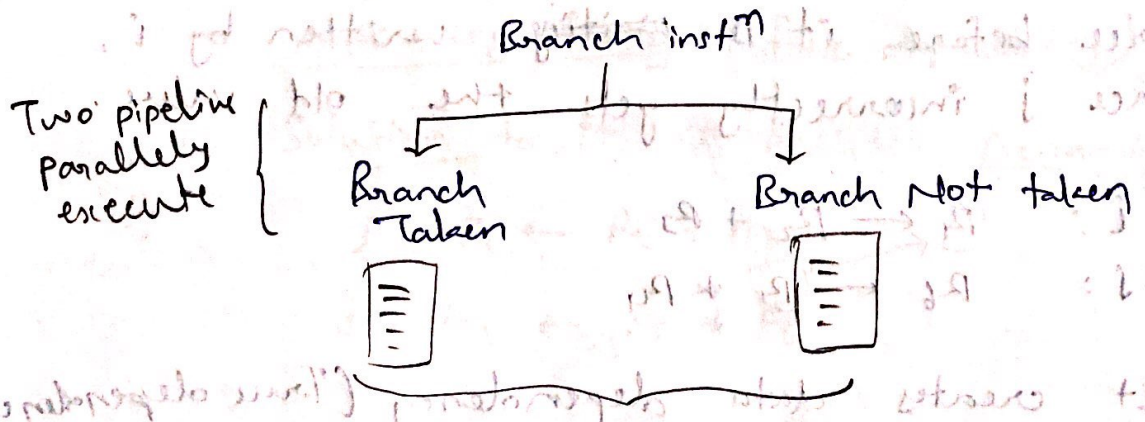
(ii) Branch Prediction: (H/W solⁿ)

Branch predictor, predict ~~the~~ whether branch is taken or not.

If branch is predicted (taken), condition will be verified if true then there is no extra stall cycle required.

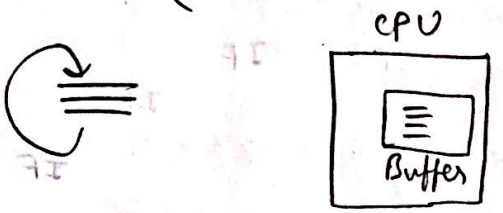
If it is false rollback to corresponding instruction & execute based on target address available which required extra stall cycle.

(iii) Prefetched Target Instruction: (H/W solⁿ)



After execution of Branch instⁿ we came to know condⁿ is true or false.

(iv) **Loop Buffer** : (H/w sol^n)



Keep the loop instructions in the CPU inside buffer.

(v) **Branch Target Buffer** (H/w sol^n)

For every branch inst^n we keep the information of target address based on its previous execution.

If previously taken branch will not be the target address then rollback.

Branch inst^n	Target Add.
I_4	I_5

* **Data Hazard Classification**

Consider two inst^n i, j; i execute befor j.

(i) **RAW** (Read After Write)

Instruction 'j' is trying to read a source before it is ~~waiting~~ written by i, hence j incorrectly gets the old value.

i : $R_1 \leftarrow R_2 + R_3$

j : $R_6 \leftarrow R_1 * R_4$

→ It creates data dependency. (True dependency) which can be solved by delayed load, operand forwarding.

(ii) W.A.W : (Write After Write)

Instⁿ j is trying to write in the destination before it is written by i

i : R4 ← R2 + R3

j : R4 ← R5 * R6

→ Output dependency (False dependency)

Solution to it is Register Renaming (H/w sol^n)

i : R4 ← R2 + R3

j : R4 ← R5 * R6

(iii) W.A.R (Write After Read)

Instⁿ j is trying to write a destination before it is read by i, hence i incorrectly gets new value.

i : R4 ← R2 + R3

j : R2 ← R5 * R6

→ Anti-dependency (false dependency)

Solution to it is Register Renaming (H/w sol^n)

i : R4 ← R2 + R3

j : R4 ← R5 * R6

Q-6 IF = 1
ID = 1
OF = 2ns
PO = 1
WO = 1

For ADD PO → 1 ns
For SUB PO → 1 ns
For MUL PO → 3 ns
For DIV PO → 6 ns

Operand forwarding is used in pipeline.

I_0 : MUL R_2, R_0, R_1 $R_2 \leftarrow R_0 * R_1$
I_1 : DIV R_5, R_3, R_4 $R_5 \leftarrow R_3 / R_4$
I_2 : ADD R_2, R_5, R_2 $R_2 \leftarrow R_5 + R_2$
I_3 : SUB R_5, R_2, R_6 $R_5 \leftarrow R_2 - R_6$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I_1	IF	ID	OF	PO	PO	PO	WO								
I_2	IF	ID	–	–	OF	PO	PO	PO	PO	PO	PO	WO			
I_3		IF	ID	–	–	–					OF	PO	WO		
I_4			IF	ID	–	–	–					OF	PO	WO	

15 cycle required.

#instⁿ = 4
k = 5

Usual no. of cycle without hazard = k + n − 1 = 5 + 4 − 1 = 8

All instⁿ takes 1 cycle for PO, WO.

For MUL → 3 → extra 2
For DIV → 6 → extra 5
For ADD/SUB → 1 → extra 0
7

∴ Total cycle = 8 + 7 = 15

For 'n' instruction,

$$\#cycles = k + n - 1$$

Without Hazard $CPI_{Avg.} = \frac{k+n-1}{n}$

With Hazard $CPI_{avg} = \frac{k+n-1+x}{n}$ [x = extra stall cycle]

In ideal condition,

for 'n' instruction, $\#cycles = n$ [Ignore $k-1$]

$$CPI_{avg} = \frac{n}{n} = 1$$

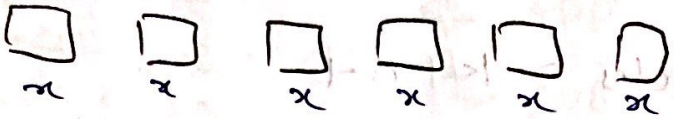
Without Hazard, $CPI_{avg} = 1$

With hazard, $CPI_{avg} = \frac{n+x}{n}$

Average instruction execution time = $CPI_{avg} * t_p$

Q-16 Consider 6-stage inst^n pipeline, when all stages are perfectly balanced. Assume that there is no cycle-time overhead of pipelining. When an appl^n is executing on this 6-stage pipeline, the speedup achieved with respect to non-pipelined execution if 25% of inst^n incur 2 pipeline stall cycles is ___.

$k = 6$



$t_p = x$

$t_n = 6x$

freq. of stall inst^n = 25%

stall cycle = 2

$$S = \frac{t_n}{CPI_{avg} * t_p}$$

$$CPI_{avg} = 0.75 * 1 + 0.25 (1 + 2)$$

$$= 0.75 + 0.75$$

$$= 1.5$$

$$\therefore S = \frac{6x}{1.5 * x}$$

$$= \frac{6}{1.5} = 4$$

Q-3 Inst^n pipeline has five stages where each stage takes 2 ns. & all inst^n use all five stages. Branch inst^n are not overlapped, i.e., the inst^n after the branch is not fetched till the branch inst^n is completed.
Calculate the avg. inst^n execution time assuming that 20% of all inst^n executed are branch inst^n. Ignore the fact that some branch inst^n may be conditional.

$t_p = 2ns$

$CPI_{avg} = 0.80 \times 2ns + 0.20(5 \times 2ns)$

$= 1.6ns + 2ns$

$= 3.6ns$

9-10 From above question, if a branch inst^n is a conditional branch inst^n, the branch need not be taken., if branch is not taken, the following inst^n can be overlapped. When 80% of all branch inst^n are conditional branch inst^n, & 50% of the conditional branch inst^n are such that the branch is taken, what is the average inst^n execution time?

20%

80%

$0.20(0.80[0.50 \times 2 + 0.50(5)] + 0.20[5]$

$+ 0.80 \times 1$

0.20

$CPI_{avg} = 0.8 * 1\ cycle + 0.2[0.80(0.5 * 1\ cycle + 0.5 * 5 cycle) + 0.2 * 5]$

Non branch | Branch | Conditional | branch taken | branch not taken | Unconditional

$= 0.8 + 0.2(0.8(3) + 1)$

$= 1.48\ cycle$

$t_p = CPI_{avg} * t_p$

$= 1.48 \times 2ns$

$= 2.96ns$

* Efficiency & Throughput

if efficiency is 100% which means the speedup of pipeline is Maximum speedup.

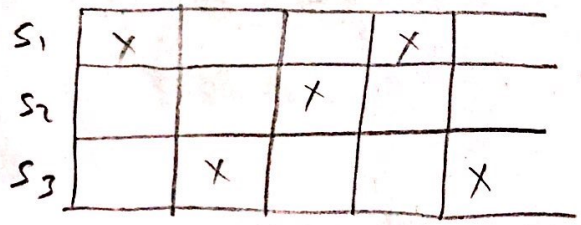
Throughput:

1 cycle → 1 instⁿ

t_p time → 1 instⁿ

$$\text{Throughput} = \frac{1}{t_p}$$

* Dynamic Pipeline



Execution Sequence: $S_1\ S_3\ S_2\ S_1\ S_3$

Initiation: Providing a new input to the pipeline is known as initiation.

Latency: Number of time units between two initiation.

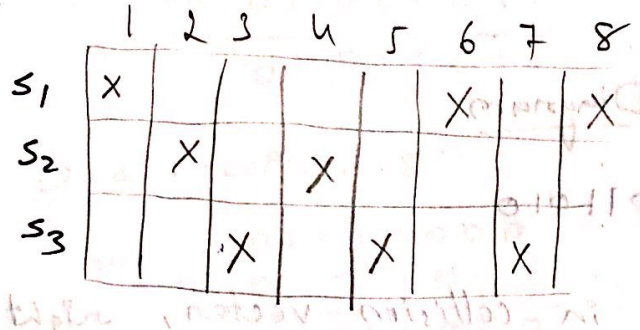
Or After how many cycle, next input provided to the pipeline.

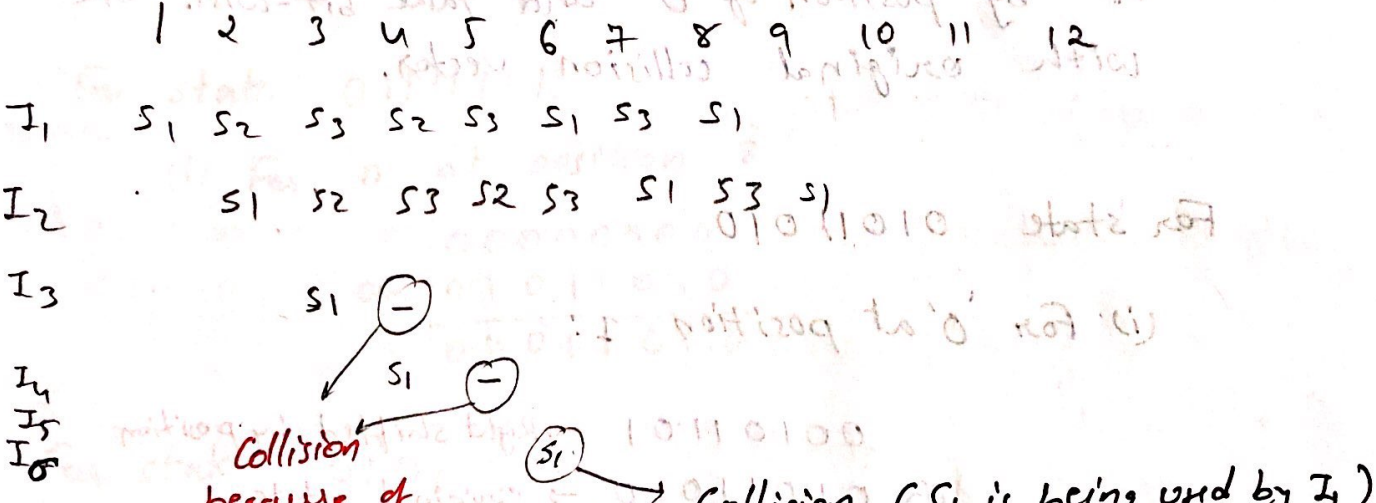
Collision: Resource conflict is known as collision.

Permissible Latency: The latency which does not cause collision.

Forbidden Latency: The latency which causes collision.

Example





Forbidden Latency : (6-1), (8-1), (8-6), (4-2), (5-3), (7-3), (7-5)

$= 5, 7, 2, 2, 2, 4, 2$

$= 7, 5, 4, 2$

Permissible latency : 8, 6, 3, 1

Collision Vector:

for collision (for forbidden) = 1
for no collision (for permissible) = 0

C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1
0	1	0	1	1	0	1	0

State Transition Diagram

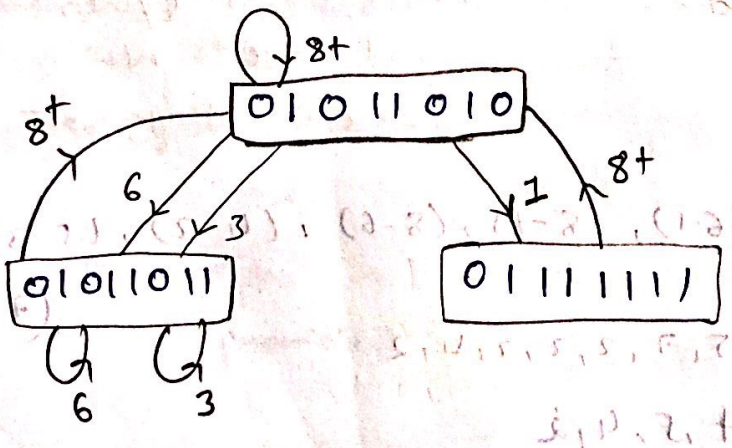
01011010

→ For every 0 in collision vector, right shift it by position of '0' and take bit-wise OR with original collision vector.

For state 01011010

(i) For '0' at position 1:

00101101 → Right shifted by position
OR 01011010 → original state
01111111



(ii) For '0' at position 3:

```
  00001011
OR 01011010
-----------
  01011011
```

(iii) For '0' at position 6:

```
  00000001
OR 01011010
-----------
  01011011
```

(iv) For '0' at position 8:

```
  00000000
OR 01011010
-----------
  01011010
```

For state 01111111

(i) For '0' at position 8:

```
  00000000
OR 01011010
-----------
  01011010
```

For state 01011011

(i) For '0' at position 3

```
  00001011
OR 01011010
-----------
  01011011
```

(ii) For '0' at position 6

```
  00000001
OR 01011010
-----------
  01011011
```

Simple Cycle: A cycle in which every state appears only once. (except the starting state)

(8), (1,8), (3,8), (6,8), (3), (6)

Greedy Cycle: The simple cycle in which the minimum latency from every state is considered.

(1,8) (3)

Average latency (1,8) $= \frac{1+8}{2} = 4.5$

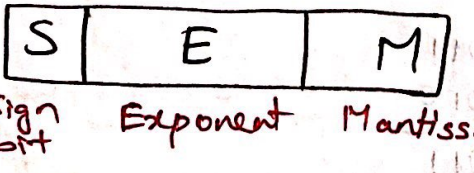
Avg. latency (3) $= \frac{3}{1} = 3$

} Min Avg. latency $= 3$

* Floating Point Representation

→ A larger range of numbers can be represented using limited number of bits as compared to the fixed point representation.

→ In this representation every number is represented as follows.



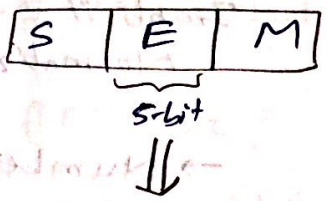
Sign bit always 1-bit

0 → -ve no.

1 → +ve no.

→ Exponent is represent in biased form

→ Mantissa is a normalized (explicit or implicit) fraction numbers.



Range of sign no. = -2^4 to $+2^4-1$

= -16 to +15

↓

biased {represent in unsigned no! i.e., +ve}

↓

0 to 31

original exponent from number (e)	stored Exponent (E)
-16	0
-15	1
-14	2
⋮	⋮
0	16
1	17
⋮	⋮
15	31

Excess-16 code

Bias = 16

Bias = 16

$E = e + bias$

$Bias = 2^{k-1}$; if k-bits are used to represent Exponent

Normalization

101.01

Explicit Normalization

→ Number should be in form 0.1___

i.e., $0.10101 * 2^3$

M = 10101 [No. after the point]

e = 3

E = e + bias

Implicit Normalization

→ Number should be in form 1.___

i.e., $1.0101 * 2^2$

M = 0101

e = 2

E = e + bias

$Value = (-1)^S * 0.M * 2^{E-bias}$

$Value = (-1)^S * 1.M * 2^{E-bias}$

→ Conventional Floating point can not represent '0'.

Q- Consides a 16-bit register used to represent floating point numbers. Mantissa is normalized sign magnitude fraction number. Exponent is in Excess-32 form. What is the 16-bit pattern representing $(-10.25)_{10}$ in this register.

→ Exponent is Excess-32

∴ bias = 32 = 2^{k-1}

∴ k = 6 [k represent Exponent]

S	E	M
1	6	9

$(10.25)_{10} = (1010.01)_2$

Explicit Normalization

$(1010.01)_2 = 0.101001 \times 2^4$

$M = 101001$

$e = 4$

$E = e + \text{bias}$

$= 4 + 32$

$E = 36$

$= (100100)_2$

$\therefore$

S	E	M
1	100100	101001 000 (Padding bit)

Q- What is the maximum value which can be represented in above register.

→ For max. no. $S = 0$

$E = E_{max} = 11\ldots1$

$M = M_{max} = 11\ldots1$

$\therefore \text{Value} = (-1)^0 * 0.111111111 * 2^{63-32}$

$= 0.111111111 * 2^{31}$

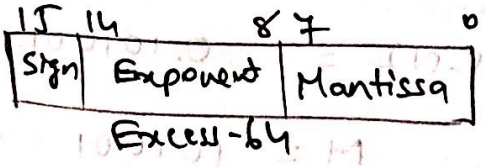
$= 1.111111111 * 2^{-9} * 2^{31}$

$= (2^9 - 1) * 2^{22}$

$= 511 * 2^{22}$

Q-4 Mantissa is a pure fraction in signed magnitude form. The decimal no. 0.239×2^{13} has the following hexadecimal representation without normalization and rounding off

(a) 0024 (b) 0D4D (c) 4D0D (d) 4D3D ✓



→

S = 1-bit
E = 7-bit
M = 8-bit

0.239×2^{13}

0.239 * 2 = 0.478 0
0.478 * 2 = 0.956 0
0.956 * 2 = 1.912 1
0.912 * 2 = 1.824 1
0.824 * 2 = 1.648 1
0.648 * 2 = 1.296 1
0.296 * 2 = 0.592 0
0.592 * 2 = 1.184 1
~~0.184 * 2 = 0.368 0~~
~~0.368 * 2 = 0.736 0~~
~~0.736 * 2 = 1.472 1~~

$0.239 \times 2^{13} = 0.00111101 * 2^{13}$

M = 00111101
e = 13
E = 13 + 64 = 77 = $(1001101)_2$

∴ 0 1001101 00111101
0x 4 D 3 D

Q-5 The normalized representation for the above format is specified as follows. The mantissa has an implicit preceding the binary (radix) point. Assume that only 0's are padded in while shifting a field. The normalized representation of the above (0.239×2^{13}) is

→ $0.00111101 * 2^{13} = 1.11101000 * 2^{10}$

$M = 11101000$

$e = 10$

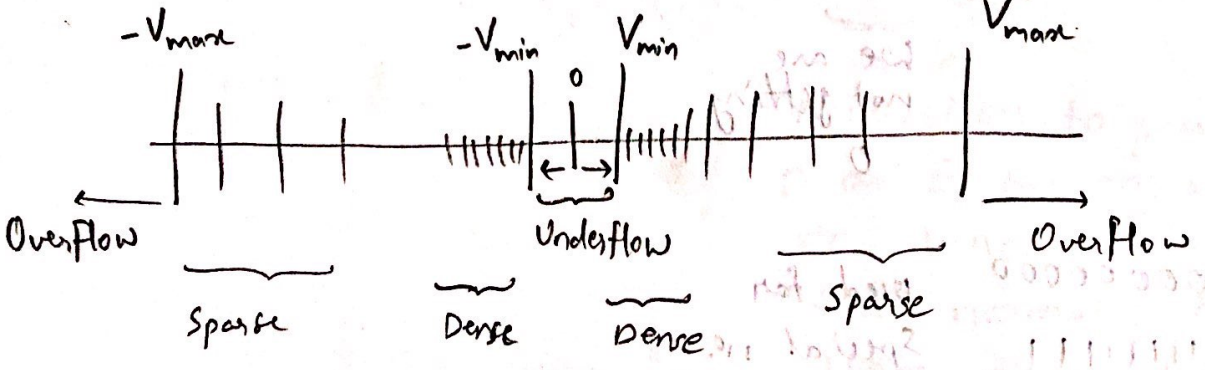
$E = 10 + 64 = 74$

$= (1001010)_2$

∴ 0 1001010 11101000

0x4AE8

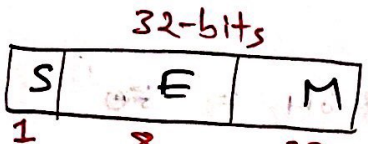
Distribution of Nos.



More bits in exponent → Large range
More bits in Mantissa → Better precision

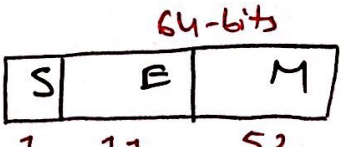
* IEEE-754 Floating Point Standard

Single Precision



Bias = 127

Double Precision



Bias = 1023

E = 8-bit
∴ Range = -128 to 127

127 + 127 = 254
e + bias
we are not getting 255

-128 + 127 = -1
we are not getting 0

E = { 00000000 used for
      11111111 Special no. representation

	S	E	M	
Special cases	0	00...0	00...0	+0
	1	00...0	00...0	-0
	0	11...1	00...0	$+\infty$
	1	11...1	00...0	$-\infty$
	0/1	111...1	M ≠ 0	Not A Number
	0/1	00...0	M ≠ 0	Fraction or denormalized no.
	0/1	E ≠ 00..0 and E ≠ 111...1	M = XX...X	Implicit normalized no.

Denormalized No. : A very very small number, which can not be implicitly normalized.

For any number if $e = -127$

biased exponent, $E = -127 + 127$

$= 0$

⇓

Restriction to generate 0 as E for any normal no., hence restrict the normalization upto only -126 position.

Eg. 0.00... 11

$1.1 * 2^{-127}$ → Not allowed

$\therefore 0.11 * 2^{-126} \Longrightarrow$ Store as denormalized

→ If after normalization upto -126 position, we get implicit normal form then that no. is implicit normalized no. otherwise it will be denormalized no. or fractional

Value (denormalized) $= (-1)^S * 0.M * 2^{-126}$ or 2^{-1023} (double)

Value (Implicit) $= (-1)^S * 1.M * 2^{E-bias}$

Q-7 Using single-precision 32-bit floating point format of IEEE-754 standard that uses 1-bit for sign, 8 bits for biased exponent & 23-bit for mantissa. A float type variable X is assigned the decimal value of -14.25. X in hexa is ___.

→ $(14.25)_{10} = (1110.01)_2$

S	E	M
1	8	23

bias = 127

$(1110.01)_2$ → Implicit representation

$= 1.11001 * 2^3$

M = 11001

E = 3 + 127 = 130

$= (10000010)_2$

∴ 1 10000010 11001000...0

0x C164 0000

Q- Consider the following 32-bit number that denotes single precision floating point no. What is the decimal value represented by it.

0 10000011 11000....0

→ S = 0

E = 10000011

= 131

131 - 127 = 4 ⇒ e = 4

M = 11000

Value = $(-1)^0 * 1.11 * 2^4$

Value = $(11100)_2$

= 28

Q- 0 00000000 1100...0

S = 0

E = 0

∴ e = 0 - 127 = -127 → denormalized so e = -126

Value = $(-1)^0 * 0.11 * 2^{-126}$

= 11.0×2^{-128}