(1D)

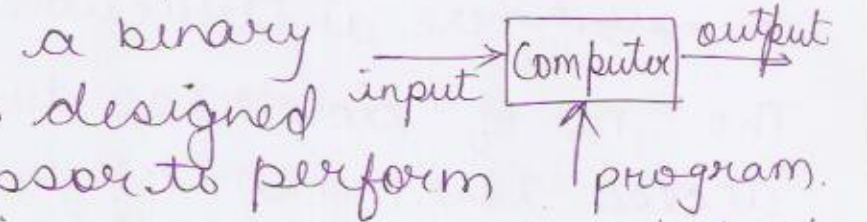
COMPUTER ORGANISATION

What is the functionality of computer?

Computer:- It converts one form of data into another form under the control of program. While execution of program, one form of data is converted into another form. Therefore, functionality of computer is execution of the program.

Program:- It is the sequence of instructions along with data.

Instruction:- It is a binary sequence which is designed inside the processor to perform some task i.e. binary sequence is bindwith the operation. [binary seq ⟺ operat^n rel at process designing]



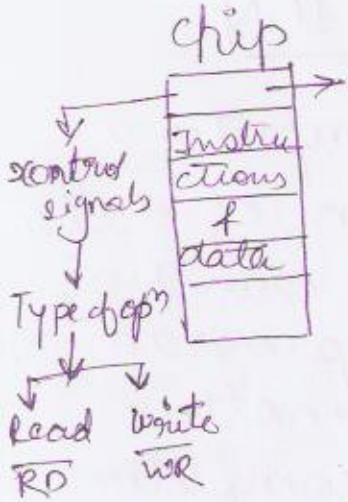
Data:- It is a binary sequence which is associated with the value based on the weighing factor.

To execute the program, the computer contain three(3) basic components named as: (1) CPU (Processing) (2) Memory (Storage) (3) I/O (External Communication)

Program is initially stored in memory.

MEMORY ORGANIZATION:-

- Memory chip is divided into equal parts called as cells.
- Each cell is able to be identified with unique no. called as address.
- Each cell is able to recognize 2 control signals, i.e. read and write.



unique
↓
Address
↓
Enables the mem. cell.

Addnal ckt for address. Not in memory.

- To access any memory content, there is a need of address along with control signal.
- To execute the program, ~~memory~~ CU generates the memory request to access the memory to read the instructions.

The memory request contains

The no. of address bits required to access the memory is depending on the memory chip configuration.

MEMORY CHIP CONFIGURATION:- The memory chips are configured as follows:-

64K×8, 64K×16, 64K by 32 ....

Let us consider 64.8 memory chip

(64K) ×8 → capacity of memory chip. i.e.

64K → # of Memory cells present in the memory chip. i.e. 2^{16} cells → # of address bits required to access 1 cell. i.e. 16 bits.

CNOTE :- If the memory chip contains 2^n cells, n-bit address is required to access 1 cell among 2^n cells) Address space/Addr range/mem.map of the mem chip i.e.

000... (16 times) 0 to 111... (16 times) 1 →

0000H to FFFFH (Based on limited human recognization space is converted into hexadecimal address space. $(0000)_{16}$/H/0x to $(FFFF)_{16}$/H

64K x (8) → Cell size.

Based on the cell size, the memory chips are classified into 2 types → byte addressable memory and word addressable memory.

Byte addressable memory :- When the memory cell size is 8 bits, then the corresponding address space is called as byte address.

Word addressable memory :- When the memory cell size is based on the word length of the processor then the corresponding address space is called as word address.

Processor	Byte(B)	Word(W) wordsize = wordlength of the processor
8 bit MP	8 bit	8 bit
16 bit MP	8 bit	16 bit
32 bit µp	8 bit	32 bit
n-bit µp	8 bit	n bit
	↑ fixed irrespective of size of µp	↑ variable size

Byte → Unique meaning i.e. unambiguity

Word → Multiple meanings i.e. ambiguity.

DEFAULT → 64KB → 64K x 8 ✓ (definite)

64KW → 64K x W → 64K x ? (undefinite)

NOTE :- When the processor word size is greater than the memory cell size then there is a need of accessing the multiple memory cells to transfer the word from memory to processor. Eg.

16 bit processor :- MOV AX, [2000]

MOV → Mnemonic → User understandable format of the operation → converted to opcode (M/c understandable formats of the operation) → Indicates type of operation, MOV indicates data transfer operation.

AX → Destination reg (Registers are always referred by the names) → 16 bits.

Register size is equal to processor size.

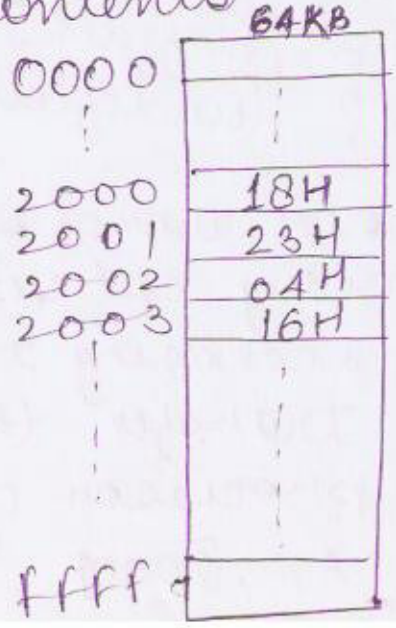
[2000] → Source → Memory (Mem. is referred by the addresses) → Byte address. → points to 8 bit data (cell size). But 2 cells are required
∴ {M[2000], M[2001]} data sent to AX.

Eg → 32 bit processor

MOV R0, [2000]
- MOV → operat^n
- R0 → DEST → Register (32 bit)
- [2000] → SRC (Memory add^r) (8 bit)

{M[2000], M[2001], M[2002], M[2003]} data transferred to R0 register.

* Consider 64KB memory with following contents



AX = M[2000] M[2001]

If [2000] contains LB
AX → 2318H

If [2000] contains HB
AX → 1823H

R0 = M[2000] M[2001] M[2002] M[2003]

If [2000] contains LB
R0 = 16042318H

otherwise
R0 = 18230416H.

During the execution of above instructions due to the lack of order of data storage, the instructions generates the multiple outputs.

To avoid the above problem, there is a need of the memory address interpretation (Endian-ness) Mechanisms.

ENDIAN-NESS MECHANISMS:- These mechanisms show the order of data storage. They are 2 types:-

(i) Little endian

(ii) Big endian

Little endian mechanism shows the order of data storage as lower address contains the lower bytes & higher address contains the higher ".

5	4	3	2	1	0	Byte storage
5	4	3	2	1	0	Addr

Big endian mechanism shows the order of data storage as lower address contains the higher byte and higher address contains the lower byte.

0	1	2	3	4	5	Byte storage
5	4	3	2	1	0	Add

NOTE:- Little endian is a default address interpretation which used in processor design.

Therefore the above instructions produce the outputs as

AX

HB	LB
23	18
2001	2000

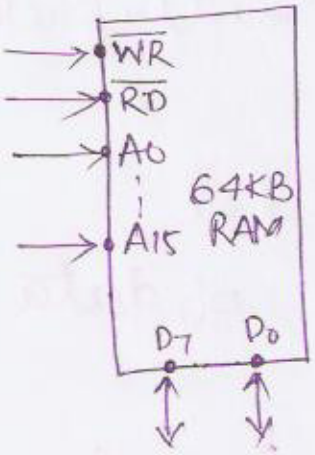
$\Rightarrow$ 2318H.

RD = | 16 | 04 | 23 | 18 | ⇒ 16042318H.

2003 2002 2001 2000

↑ HB ↑ LB

Memory



Processor PIN STRUCTURE:-

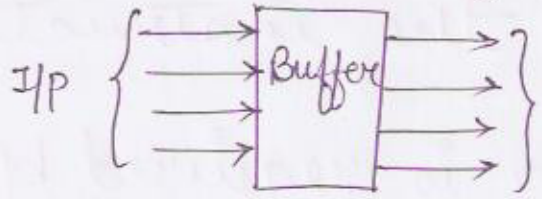
Processor supports 3 catagories of h/w pins to perform the various operations on externally connected components i.e. memory or I/O.

① Active low pins: These pins are enabled when the pin is zero. They are denoted as $\overline{\text{Pinname}}$; Eg → $\overline{RD}$, $\overline{WR}$, $\overline{CS}$, $\overline{INTA}$ etc.

② Active high pins:- These pins are enabled when the input is 1. They are denoted as Pinname. Eg → ALE, INTR, HOLD, HLDA etc. Some component are anode, some cathode (ALP) Enable (AHP) Enable
∴ Above type of pins used in processors.

③ Time multiplexed pins:- This pin carries the multiple meanings but the meaning is defined by another pin. Eg. In 8085, $AD_0 - AD_7$. A → Address, D → data controlled with ALE; [0 → Data is carried by Time Multiplexed Pins, 1 → Address " " " " " "]
$AD_8 - AD_{15}$ in 8086.
The advt of TM pins is that it reduces the no. of h/w pins.

Buffer :- Buffer is a temporary storage component without locking. Input & output lines are always enabled.



component to component comm. takes place using buffer.

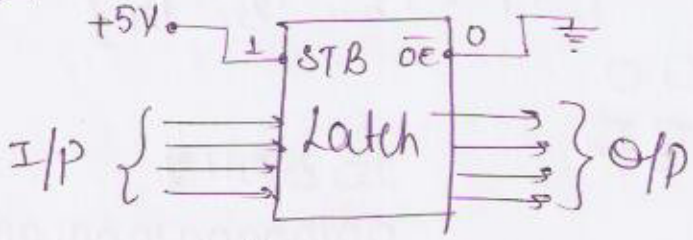
Latch :- Latch is a temporary storage component with locking i.e. i/p & o/p lines ~~and~~ are there in the control of locking pins i.e. strobe (STB) and output enable ($\overline{OE}$) ~~pairs~~ pins respectively.

STB
- 0 input lines disabled
- 1 input lines enabled.

$\overline{OE}$
- 0 output lines enabled
- 1 output lines disabled

NOTE :- Latch may be configured as the buffer but buffer can never be configured as latch.

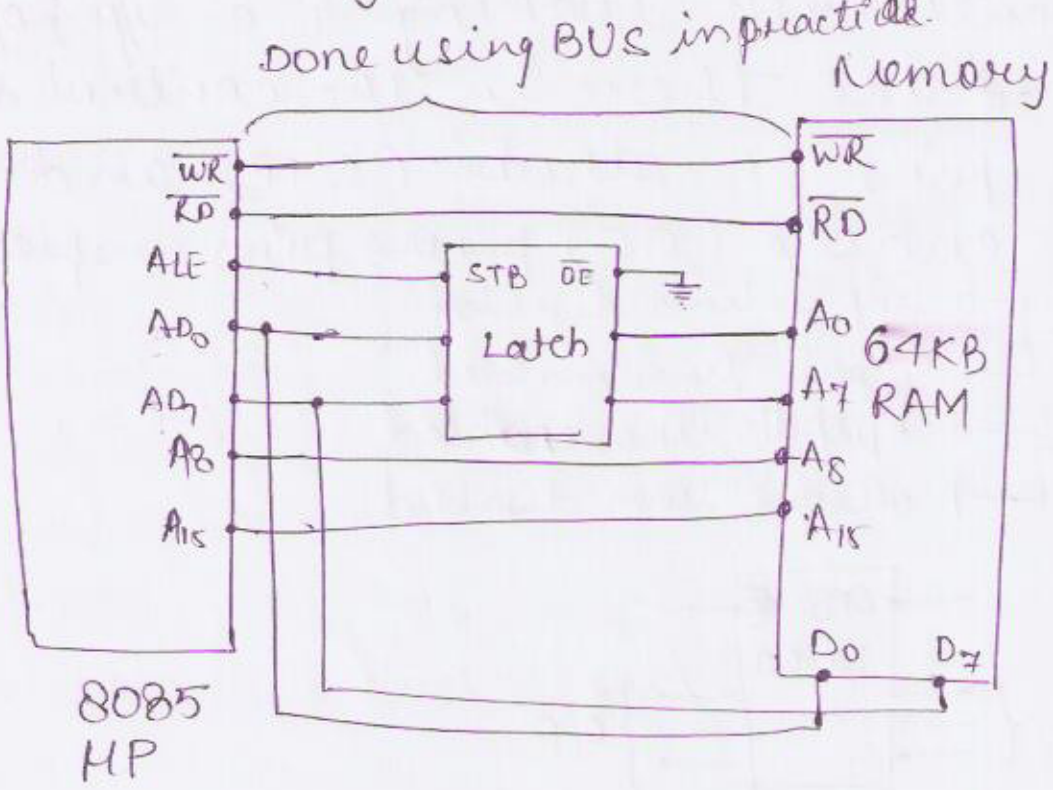
Latch is configured as the buffer by connecting STB pin to +5V and $\overline{OE}$ pin to GND.



Latch as Buffer.

MEMORY INTERFACING :

To execute the program, there is a need of communication required b/w CPU & memory to read the instruction & data.
Therefore, interfacing is required b/w the CPU & memory.



CPU generates the mem. request : 20804 (RD)

M/C cycle :- The no. of clock cycles required to access memory (to read or write) is called machine cycles.

Machine cycles → $\{T_1, T_2, T_3, T_4\}$

ALE → 1, 0, 0, 0
T_1, T_2, T_3, T_4

$\overline{RD}$ → 0 (T_2)

$\overline{WR}$ → 1 (T_2)

2080H

0010000010000000

A_{15} - A_8 | AD_7 - AD_0

T1 ⇒ $AD_0 - AD_7 \rightarrow 80$, $A_8 - A_{15} \rightarrow 20$ ← CPU. ALE → 1
Latch - 80
R $AD_0 - A_7 \rightarrow 80$, $A_8 - A_{15} \rightarrow 20$ ← RAM, STB → 1

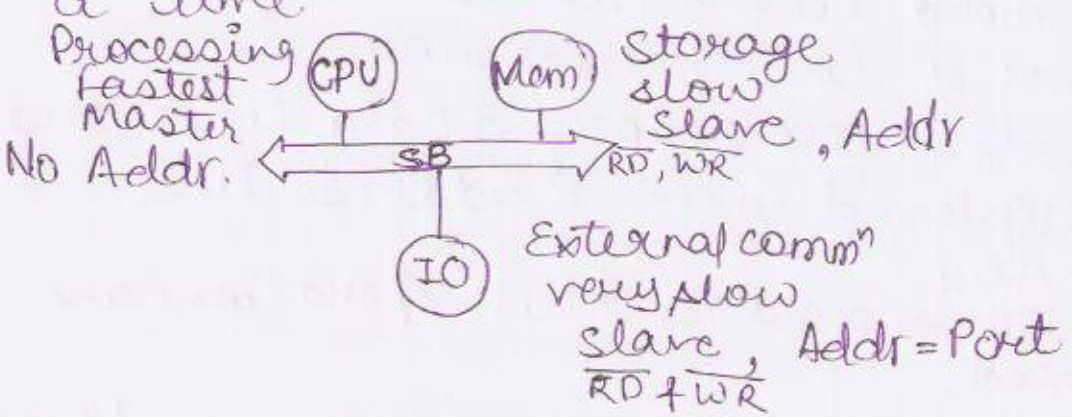
T2 ⇒ $\overline{WR} - 1$, $\overline{RD} - 0$ } CPU, RAM ALE-0, STB-0.

T3, T4 → Binary sequence at $D_0 - D_7$ in RAM to $AD_0 - AD_7$. This sequence is not forwarded to latch. Data is moving between memory and CPU. 2 cycles because memory is slower than CPU.

NOTE: In every m/c cycle, in first time state, ALE pin is in the high/enable state, & remaining time states, ALE is disabled to carry the address & data efficiently over the TM pins.

BUS: Bus is a communication channel used between the major components of the system i.e. CPU, memory and I/O.

- It is also called as the system bus.
- The key characteristic of the bus is shared transmission media. The limitation of the bus is only one transmission at a time



The communication channel contains 3 lines to provide communication.

1) Address lines
2) Data lines
3) Control lines

Address lines

These lines are used to carry the address towards memory & I/O. They are unidirectional.

Based on the no. of address lines, we can determine the capacity of the memory system. Eg. 8085 supports 16 address lines i.e. AD_0-AD_7 & A_8-A_{15}.

$\Rightarrow 2^{16}$ address cells $\Rightarrow$ 64KB (64K cells).

8086 supports 20 address lines i.e. AD_0-AD_{15} $A_{16}-A_{19}$. $\Rightarrow 2^{20}$ address cells $\Rightarrow$ 1M cells $\Rightarrow$ 1MB storage space.

Data Lines:- These lines are used to carry the binary sequence between CPU, memory & I/O. Therefore, data lines are bidirectional.

Based on the no. of data lines, we can determine the word length of the processor. Based on word length, we can determine the performance of the processor.

Eg: In 8085 processor, 8 Data lines are time multiplexed with 8 address lines i.e. AD_0-AD_7.

Word length $\Rightarrow$ 8 bit $\Rightarrow$ 8 bit operations are performed.

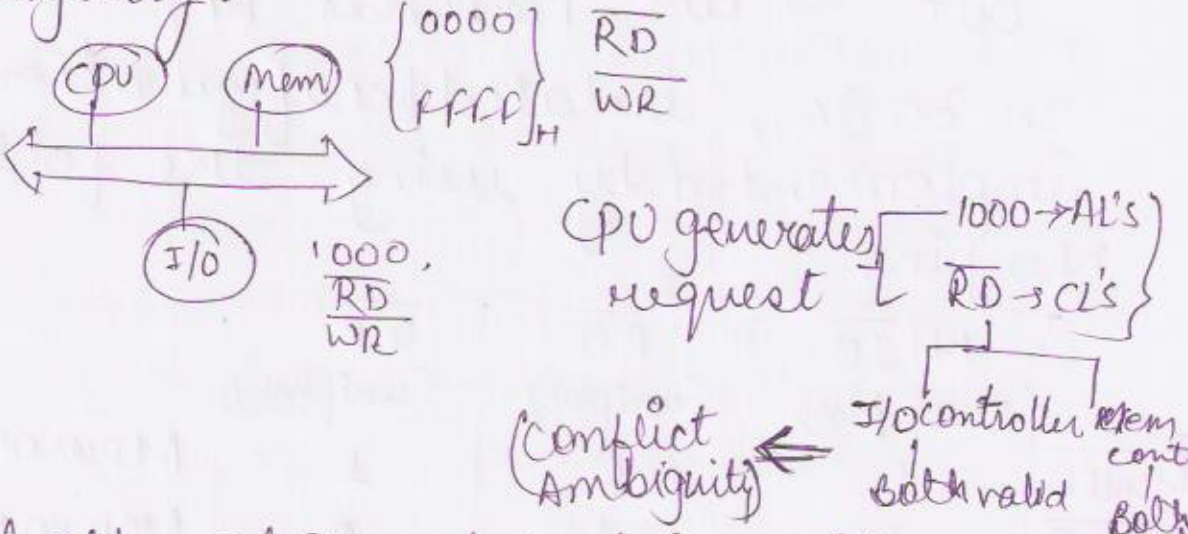
In 8086, 16 data lines are time multiplexed

with 16 Address lines. i.e. $AD_0 - AD_{15}$.
Total word length = 16 bit ⇒ 16 bit operations are performed.

Control lines:-

These lines are used to carry the control signals & timing signals. Control signals are used to indicate the type of the operation and timing signals are used to synchronise the memory & I/O operations with processor clock. Control lines are bidirectional.

NOTE:- When there is a common channel, common address space, and common control ~~sequence~~ signals maintained between the memory and I/O, then there is a possibility of ambiguity.



To avoid the above problem, there is a need of bus configuration

Bus Configuration:- There are 3 bus configuration in system design.

(i) I/O Processor (IOP)
(ii) Isolated processor
(iii) Memory mapped I/O

(1) IOP (Input Output processor):- This configuration uses the separate buses for I/O and memory. Introducing an additional bus is an expensive process. So this config is used in the high performance processor design.

(2) Isolated I/O:- This configuration uses the same bus and common address space but different control signals.

CS → Mem related → MEMRD (LOAD)
CS → Mem related → MEMWR (STORE)
CS → IO related → IOWR (OUT)
CS → IO related → IORD (IN)

8085 & 8086 uses isolated I/O.

LOAD → LOAD R0, [2000]
STORE → STORE [2000], R1
IN → IN R0, Port Addr
OUT → OUT PortAddr, R1

In 8086, isolated I/O configuration is implemented by using the following H/W pins.

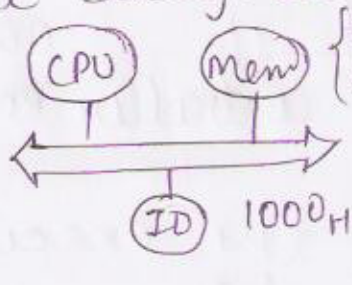
	M/$\overline{IO}$ (dual pin)	$\overline{RD}$ (unipins)	$\overline{WR}$ (unipins)	
$\overline{MEMRD}$	1	0	1	Memory Read
$\overline{MEMWR}$	1	1	0	Memory write
$\overline{IORD}$	0	0	1	IO read
$\overline{IOWR}$	0	1	0	I/O write

In 8085, isolated I/O configuration is implemented by using the following H/W pins:-

different from 8086.

IO/M	RD	WR	
1	0	1	IORD
1	1	0	IOWR
0	0	1	MemRD
0	1	0	MemWR

So, the diagram changes To:



CPU generates Mem req.

1000 → AL's
MemRD → CL's

Mem Controller: 1000 Valid, MemRD Valid

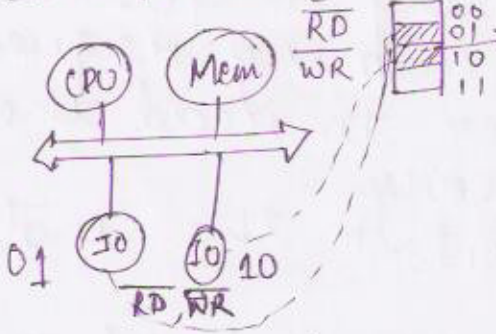
I/O Controller: 1000 valid, MemRD Invalid

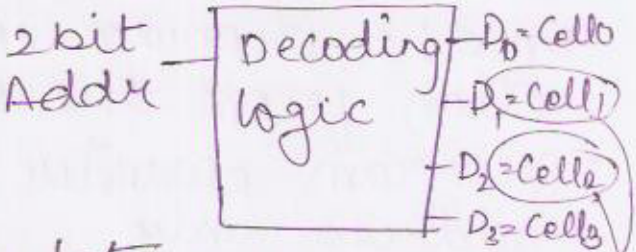
mem opⁿ. NOP.

Memory Mapped I/O

This configuration uses the common bus and common control signal but unique address space.

Unique address space is maintained by taking some of the addresses from the pool of the address space, assigned them to I/o ports. that means, port addresses are the part of the memory address space.





Disadvt → ~~Not~~ complete memory space is not in use. (Not allowed to be used by user. Reserved by processor)

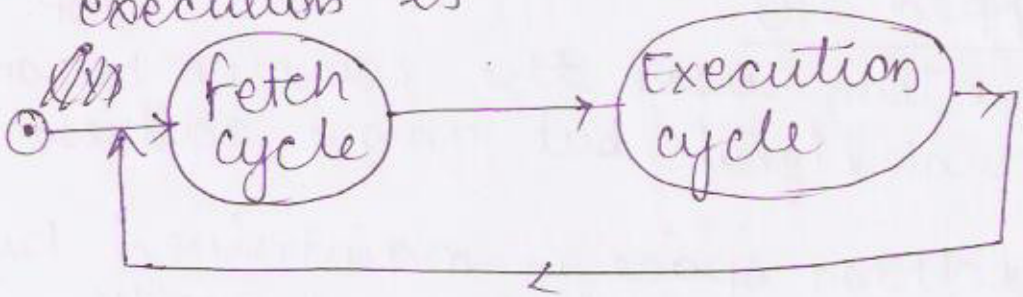
The disadvantage in memory mapped I/O is that the complete memory is not in the use.

CPU generates: 00 → AL's
Mem Req: $\overline{RD}$ → CL's

- Mem controller: 00 valid, $\overline{RD}$ - valid → Mem opⁿ
- I/O controller: 00 - Invalid, $\overline{RD}$ - valid → I/O

Instruction Cycle:-

This concept describes the execution sequence of an instruction.
Instruction cycle consists of 2 subcycles named as fetch & execution cycle.
The sequence diagram of instruction execution is



Fetch cycle:-

- The objective of the fetch cycle is to read the instruction related binary sequence from the memory to CPU.
- The CPU generates the memory request based on the program counter to read the instruction from the memory.
- Program counter points to the instruction address only.
- The functionality of PC is to hold the starting instruction address and immediately points to the next instruction address.
- Here the starting address is provided by the

user and the next instruction address is calculated by incrementing the PC with step size.

- Step size depends on the instruction length i.e. if the instruction size is n byte long, then step size is "n".
- The process of transfer the binary sequence from the memory to CPU based on PC is called as instruction fetch.
- At the end of instruction fetch, PC is incremented to step size to point the next instruction address.

Inst^{n} fetch:
- t = 2000
- trace ↓ starting addr of the prog. → PC → CPU gen Mem Req.
- PC ← PC + step size

Q Consider the program segment executed on a hypothetical processor :-

Address	Instruction	Size (Words)	
1000-1007	I_1	2	+8
1008-1011	I_2	1	+4
1012-1015	I_3	1	+4
1016-1023	I_4	2	+8
1024-1027	I_5	1	+4
1028-1035	I_6	2	+8
1036-1039	I_7	1	+4

Assume the word size is 32 bits. The prog has been loaded in the memory with the starting address of 1000 (decimal) onwards. During the execution of inst^{n} 5. what could be the value present in PC?

ANS → 1028.

Starting Address of inst are valid addrs for PC.

(b) Assume that memory is a word addressable with word size of 32 bits. The program has been loaded in memory with the starting address of 1000 (decimal) onwards. During the execution of I6 what could be the value of PC?

Address:- 1000-1001, 1002, 1003, 1004-5, 6, 7-8, 9.

Soln

NOTE:- When the processor is supported with fixed length instructions then step size is a fixed constant. When the processor is supported with variable length instructions, then step size is also variable.

Q. A computer has 24 bit instruction. The program has been loaded in the memory with the starting address of 300 (decimal) onwards. Which one of the following is the valid program counter values?

(a) 400
(b) 500
(c) 600 ✓
(d) 700

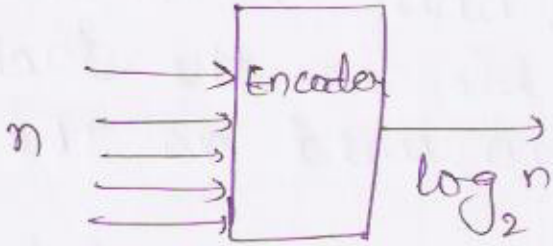
Execution cycle:-

- The objective of the execution cycle, currently fetched instruction undergoes processing.
- To process the instruction there is a need of identifying the type of operat

associated with the binary sequence.
- Opcode indicates the type of operation
- Opcode is present in the instruction but that is defined by using instruction format.
- Instruction format shows the layout of an instruction that means, it describes the internal structure of the instruction.

Encoding (Many to 1 Mapping)

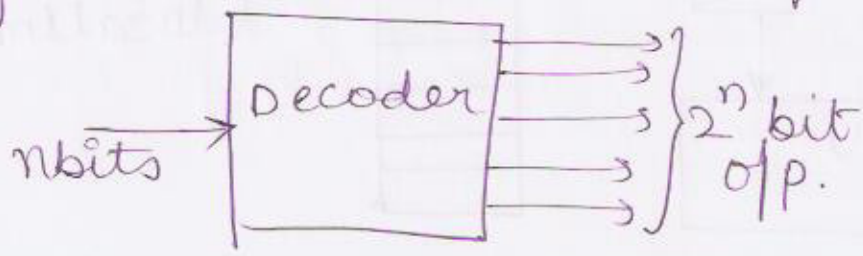
In the encoding process, n different combinations can be represented using $\log_2^n$ bits to indicate only one operation at a time



NOTE:- Instructions are always represented in the form of encoded binary format. This means when the processor supports 16 different operations then 4 bit opcode is required to indicate 1 operation at a time

Decoding (1 to many mapping)

In the decoding process, n bit decoder generates the 2^n output combinations



NOTE: During the execution of the instructions decoder is required to identify the operation associated with the encoded binary format.

Instruction format is classified into 5 types based on the type of CPU organization.

The CPU organization is classified into 3 types based on the availability of ALU operands.

Classification of Instruction set Architecture

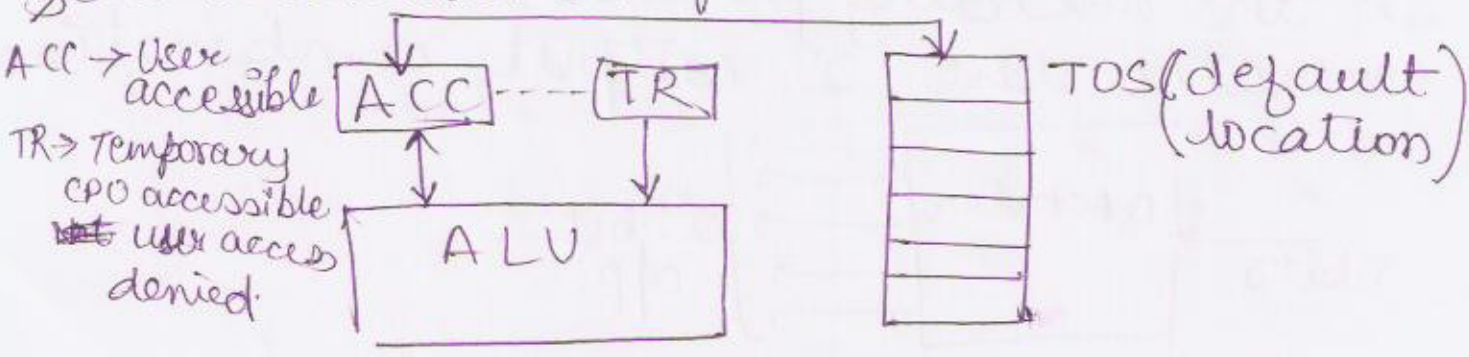
(1) Stack machine
(2) Accumulator machine
(3) General register organization

Stack Machine

In this organization, both the ALU operands are available in the stack. The same stack is also used as the destination.

The stack contains only one entry point and only 1 exit point i.e. top of stack (tos). So tos is default location. Therefore, No need to pass the tos address explicitely in the instruction.

- The instruction format contains only 1 field i.e. [opcode]
- The above instruction format is known as zero instruction format.



OPCODE

↓

Type of opⁿ

↓

ALU opⁿ: src1 src2 Dest
TOS TOS TOS.

↓

POP → POP → operate → push.

Eg. ADD.

Sequence of operations → POP, POP, +, push

NOTE:- In the stack machine ALU operations are the zero address instructions but data transfer operations are not zero addr. instruction

Q. Consider the following program segment used to execute on a stack machine. Each arithmatic operation pops the 2nd operand, pops the 1st operand, operates on them & push back the result onto the stack.

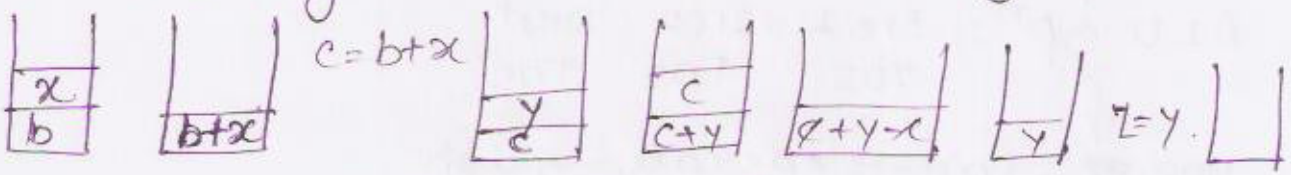
Push b	Push y
push x	ADD
ADD	Push c
Pop c	SUB
push c	POP z

Which of the following statements is/are true?

1. If each push and pop instruction requires 5 bytes of storage and each arithmatic instruction requires 1 byte of storage, then the whole program requires 40 bytes of storage.
2. At the end of execution, z contains the same value as y.
3. At the end of execution stack is empty (Consider stack is initially empty)

a) 1 only (c) 2 & 3 only ✓
b) 2 only (d) 1, 2, 3.

Total storage = 7×5 + 3×1 = 38 bytes.



(2D)

(2) Accumulator Machine

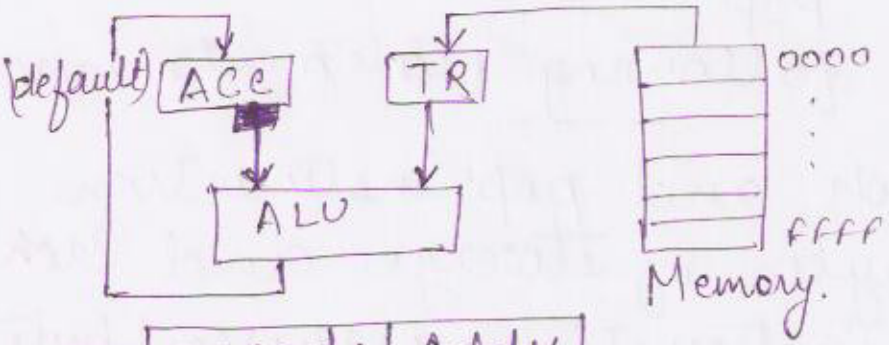
In this organization, the first ALU operand is default available in Accumulator. The same accumulator is used as the destination.

The CPU contains only 1 accumulator register so it becomes the default location.

The 2nd ALU operand may be present in the memory or register. So, memory address or register name is required in the instruction to access the operand 2.

Therefore, the instruction format contains 2 fields i.e. | opcode | Addr |

The above format is known as 1 address inst^{n} format.



| opcode | Addr |

Type of operation

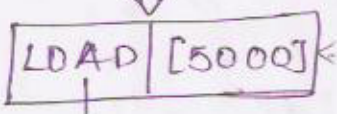
Data Transfer operat^{n}

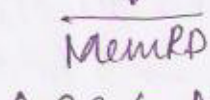
Source	Dest^{n}	
ACC	Mem (write)	STORE
Mem	ACC (Read)	LOAD

ALU OP^{n}:

SRC1	SRC2	DSTN
ACC	Mem/Reg	ACC

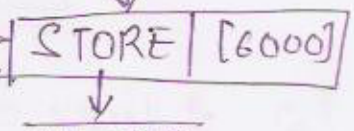
Eg 1: LOAD [5000]

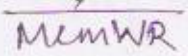




ACC ← M[5000]

Eg 2: STORE [6000]

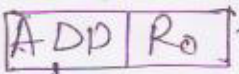




M[6000] ← ACC.

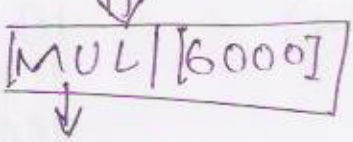
Inst^n format

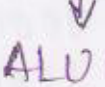
Eg 3: ADD R0.



ACC ← ACC + R0.

Eg 4: MUL [6000]





ACC ← ACC * M[6000]

Q. Consider a hypothetical processor. It supports both 1 addr. & 0 addr instructions. It has 6 bit instructions & 4 bit addresses. If there exists 2 one address inst^n's, how many 0 address instructions can be formulated.

NOTE: 1 Address m/c also supported by 0 address instructions if there is a free space after allocating the 1 address inst^n. This technique is called as expand-opcode technique.

Step 1:- Identify the highest instruction format according to the problem statement. 1 addr

S2: Identify the no. of possible operations. (Instructions)

S3: Identify the no. of free combinations (free opcodes) after allocating the high order instructions.

S4: Calculate the no. of low order Inst^n by multiplying the decoded value of the Addr field with the free opcode.

Sol^n:- (1) ← 6 →
| Opcode | Addr |
← 2 → ← 4 →

(2) N = 2 opcode size
$N = 2^2 = 4 \Rightarrow$ opcode | address
00 → Address
01 "
10 "
11 "

(3) # free opcodes = N − # of existing instructions
= 4 − 2 = 2
00 } → 1 Addr inst
01 }
10
11

Expansion ⇒

(4)
100000
100001
000000
100011
⋮
101111
} 16 + 16 { 110000, 110001, 110010, 110011, ⋮, 111111 } ⇒ 32

# of Zero Addr inst^n = # of free opcodes $* 2^{\text{Addr field size}}$
$= 2 \times 2^4$
$= 32$.

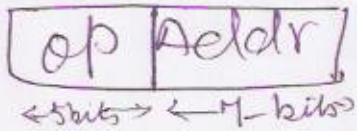
Q Consider a hypothetical processor. It supports both 1 addr & 0 addr inst. A 16 bit instruction placed in the 256W memory. If there exists 100 1 addr instructions, How many 10 addr instructions are possible?

156X2^8

⇒
| op | Addr |
8bits ← → 8bits
← 16 →

256 − 100 = 156 X 2^8

Q. Consider a processor which has 128W memory. It supports both 1addr & 0addr instructions. A 12 bit instruction is placed in 1 memory word. What is the possible range of 1addr & 0 addr inst^n supported by the processor.



$N = 2^5 = 32$

share the 32 op^n's. B/w the 1 Addr & zero add.

1-Addr	0-Addr
1	31
31	1

Range of 1 Address = {1 to 31}

Range of 0 address = {1x2^27 to 31x2^27}

3) General Register Organization :- Based on the no. of registers supported by the processor, the architecture is divided into 2 types :-

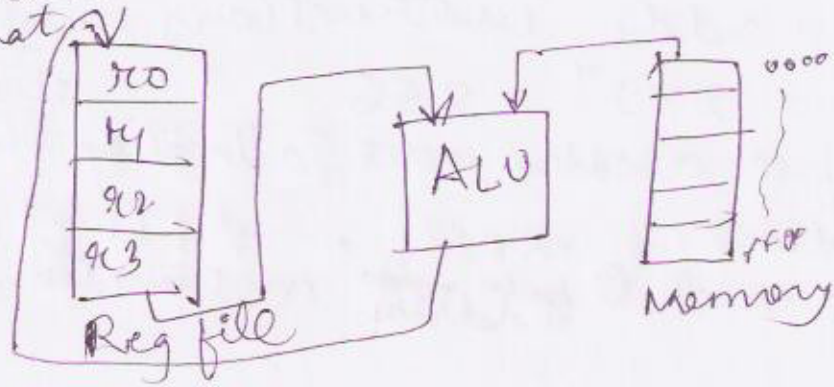
(a) Register to memory reference architecture

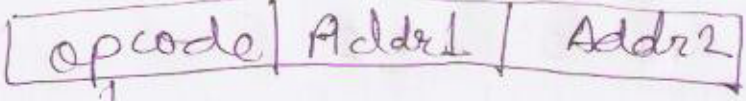
(b) Reg-Reg reference architecture.

Register to memory reference Architecture

In this architecture, processor supports less no. of registers. That means, register file size is small.

In this organization, the first ALU operand is always present in the registers, the same register is also used as the destination. The second ALU operator is available either in memory or in registers. So this architecture supports on instruction format with 3 different fields - opcode | Addr1 | Addr2 The above instruction is called as 2 address Instruction format.





Type of operation

Data transfer operation

Src	dest
reg	reg
Reg	Mem
mem	reg

} MOV

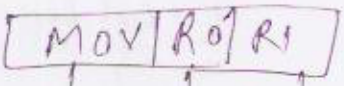
ALU opⁿ:-

Src1	Src2	Destn
reg	mem / reg	source1

Eg 5:- Add R0 [6000]

R0 ← R0 + M[6000]

Eg1:- MOV R0, R1



opⁿ dest src

R0 ← R1

Eg 2: MOV R0, [4000]

R0 ← M[4000].

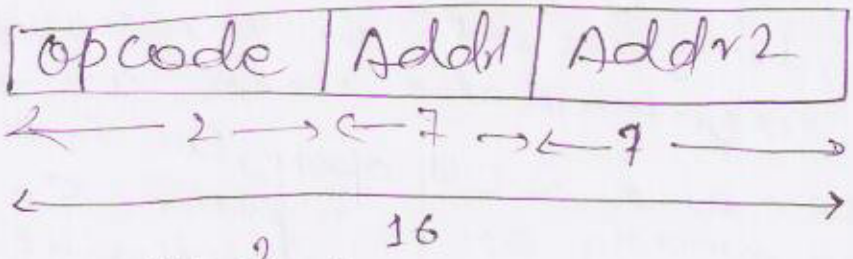
Eg 3: MOV [6000], R0

M[6000] ← R0

Eg 4:- Add R0, R1

R0 ← R0 + R1

Q. Consider a processor that supports 2 addr, 1 addr & 0 addr. A 16 bit instruction is placed in 128 W memory. If there exists 2-2 address instructions, 100-1 address instructions, how many instruction for 0 can be formulated?



$N = 2^2 = 4$.

No of free opcodes after allocating the 2 addr instructions = 4 - 2 = 2.

# of 1 addr instructions = OP | Addr

OP Addr1 Addr2

$= 2 \times 2^7 = 256$

# of free opcodes after allocating the 1 addr instruction = 256 - 10 = 156

∴ No. of 0 address instructions = 156×2^7

Q. A computer has an instruction format with 3 fields. i.e., opcode, register field which indicates 1 of the 60 processor register and memory address. A 32 bit instruction is placed in the 256 KW memory.

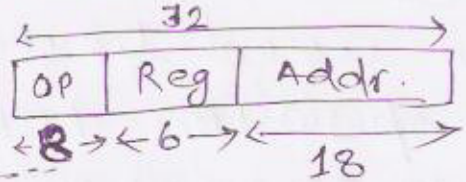
(a) How many no. of operations are possible.

(b) If there exists 2-address instructions which uses both register and memory reference, then how many 1 address memory reference instructions are possible. (register merging with opcodes)

(c) If there exists n-2 address instructions which uses both register & memory reference, m-1 address register references then how many zero address instructions are possible.

Ans:-

(a)



N = 256 ⇒ # free opcodes after allocating 2 addr. instructions = (256-n)

# of 1 address mem ref = $(256-n) \times 2^6$ → Reg merged.

OP	Add (Mem)

# of 1 addr reg ref = $(256-n) \times 2^{18}$

# of free opcodes after allocating 1 addr mem ref

$\text{Inst}^n = [(256-n) \times 2^6] - m$

# of free opcodes after allocating 1 add reg ref

$\text{Inst}^n = [(256-n) \times 2^{18}] - M$

# of free opcodes after 0-address = $(((256-n) \times 2^6) - m) * 2^{18}$

∴ 0 add ⇒ $(((256-n) \times 2^{18}] - M) * 2^6$

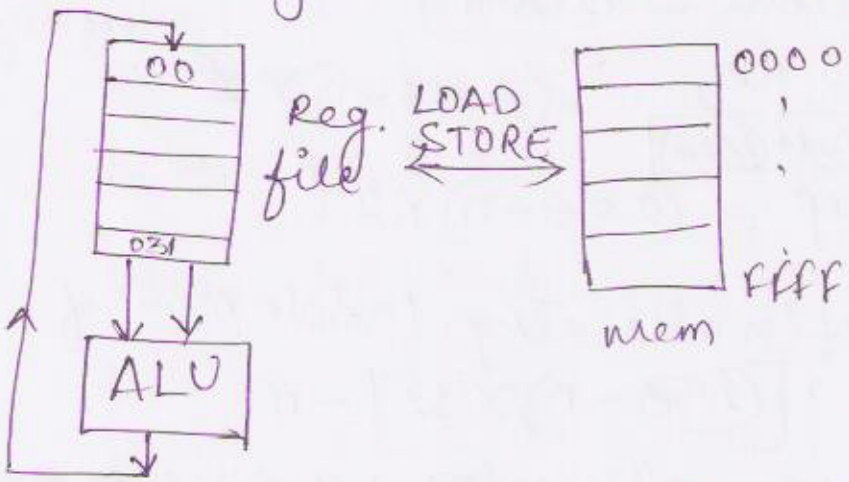
Register to register reference architecture

In this architecture, the processor supports more no. of registers i.e. register file size is large. No possibility of register overwriting.

In this architecture ALU operations are performed only on registers. i.e. Both the ALU operands are available in register. Due to the more no. of registers present in this processors use a separate register to store the output of the ALU operations

Therfore ALU operations require 3 address fields in the instruction. All the 3 address fields contains register names.

In this architecture, 2 address instruction ~~archit~~ format is used to implement the data transfer operation. LOAD and STORE instr are used to transfer the data b/w the registers and memory.



The 3 instr format is | opcode | Addr1 | Addr2 | Addr3 |

opcode → Type of operation → ALU operation

Addr1 — Dstn — Reg

Addr2 — SRC1 — Reg

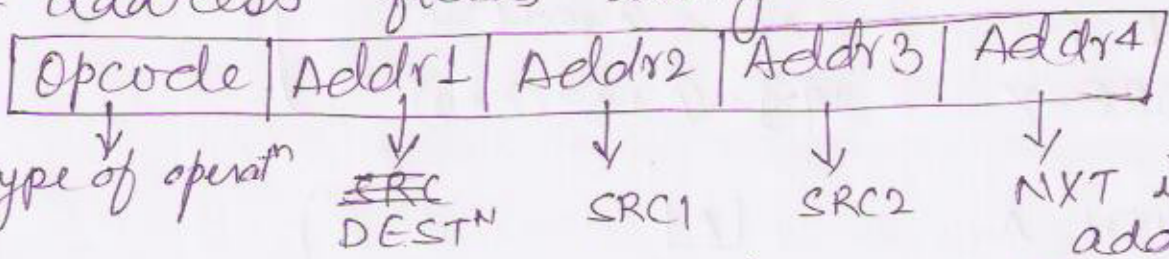
Addr3 — SRC2 — Reg

Eg: ADD r_0, r_4, r_2

$r_0 \leftarrow r_4 + r_2$.

FOUR ADDR INSTRUCTION FORMAT

There is an instruction format which contains 4 address fields along with the opcode.



∵ Addr4 is already implemented using Program counter. ∴ 4-address instruction format is imaginary & Not yet implemented. PC is mandatory register in processor design. It is used to hold the next instr address. So, above concept is not in use.

NOTE:- ⊛ 8085 μP is an example of accumulator machine.

⊛ 8086 μP is an example of the reg to mem reference architecture.

⊛ RISC μP is an example of the reg to reg reference architecture.

Q. Consider the following expression $X=(A+B)*(C+D)$ used to execute on a stack machine, accumulator m/c & reg to mem & reg to reg machine. Here A B C D and X are ~~area~~ present in the memory locations. How many no. of instructions are required to execute the expression on the respective m/cs.

$X = (A+B)*(C+D)$

LOAD A	Acc ← [A]
ADD B	Acc ← [A] + [B]
STORE P	M[P] ← Acc
LOAD C	Acc ← [C]
ADD D	Acc ← [C] + [D]
MUL P	Acc ← (A+B)*(C+D)
STORE X	M[X] ← (A+B)*(C+D)

} Accumulator M/C.

PUSH A	[A]
PUSH B	[B] [A]
ADD	[A+B]
PUSH C	[C] [A+B]
PUSH D	[D] [C] [A+B]
ADD	[C+D] [A+B]
MUL	[(A+B)*(C+D)]
POP X	M[X] ← (A+B)*(C+D)

} Stack Machine.

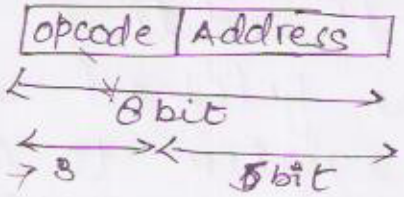
MOV R0, A	$R_0 \leftarrow M[A]$
ADD R0, B	$R_0 \leftarrow R_0 + M[B]$
MOV R1, C	$R_1 \leftarrow M[C]$
ADD R1, D	$R_1 \leftarrow R_1 + M[D]$
MUL R0, R1	$R_0 \leftarrow R_0 * R_1$
MOV X, R0	$M[X] \leftarrow R_0$

} Reg to Mem.

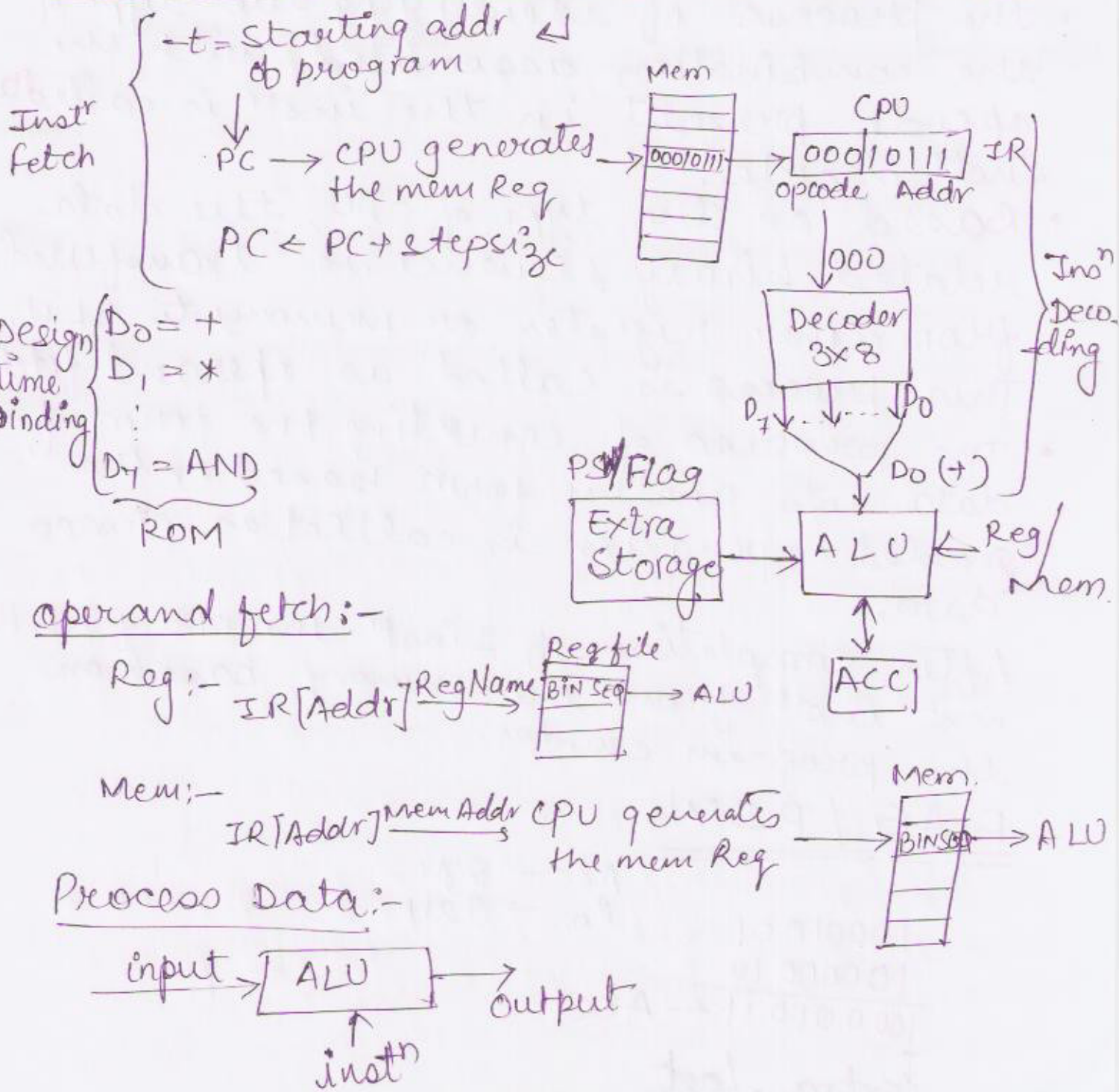
LOAD R0, A	$R_0 \leftarrow M[A]$
LOAD R1, B	$R_1 \leftarrow M[B]$
ADD R2, R0, R1	$R_2 \leftarrow R_0 + R_1$
LOAD R0, C	$R_0 \leftarrow M[C]$
LOAD R1, D	$R_1 \leftarrow M[D]$
~~ADD R3, R0, R1~~ ~~MUL~~ ~~ADD~~	$R_3 \leftarrow R_0 + R_1$
MUL R4, R2, R3	$R_4 \leftarrow R_2 * R_3$
STORE X, R4	$M[X] \leftarrow R_4$

} Reg to Reg

NOTE:- To analyze the execution sequence of the program, let us consider an example CPU 4 its compatible instruct format i.e. accumulator m/c as an eg. CPU. It supports 8-bit instructions with 8 different operations. & inst format is 1-address format.



EXECUTION SEQUENCE OF AN INSTR



Based on the PC, the instr related binary seq is transferred from memory to CPU. This process is called as instruction fetch. At the end of inst fetch, PC is incremented to step size to point to the next instr addr.

- All the fetched instr is placed into the instr register to decode because instruct format is predefined in this register (IR).
- The process of identifying the type of the combination associated with the opcode present in the instr is called as instr decode.
- Based on the type of CPU, the data related binary sequence is transferred from either register or memory to ALU. This process is called as operand fetch.
- The process of converting one form of data into another form based on the enabled operation is called as process data.

After completion of 1 instr the CPU fetch the next instr from the memory based on the program counter.

FLAG / PSW

ACC − 89H
R0 → 82H

```
  10001001
  10000010
 ----------
 100010011 ← ACC
 ↓
 Extra, Lost
```

To store the exception conditions, there exists extra storage provided in processor called as flag register / PSW (Program status word).

Flag - flag is a flip-flop. flip-flop is a bistate device. It is used to store the one bit information.

Flags are classified into 2 types:
→ Conditional flag.
→ Control flags

Conditional flag:-
These flags are set or reset based on the result nature of ALU. There are 6 conditional flags used in processor:-
(i) Carry (ii) Parity (iii) Auxillary carry (iv) Zero (v) Sign flag (vi) Overflow flag.

Carry:- It is a conditional flag set on "Is there an extra bit out of MSB".
- T = set = 1 = C
- F = reset = 0 = NC

Carry flag is used to indicate the range exceeding conditions of the unsigned arithmatic operations.
In the unsigned representation, we can represent only the positive no. only.
n-bit unsigned range is 0 to $2^n - 1$
Eg. 4-bit unsigned range is 0 to 15.

$7 = 0111$
$3 = 0011$
$10 \quad 1010$
CY-0 CY-0
↑ logical

$9 = 1001$
$8 = 1000$
$17 \quad \boxed{1}0001$ ← ACC
CY-1 CY-1

Parity Flag:- "Is the ALU output (ACC content) contain even no. of 1's".
- T = set = 1 = PE (even)
- F = reset = 0 = PO (odd)

This flag is used in the serial communication to prepare the error detection codes.

Auxilary carry:- "Is there an extra bit from the lower nibble to Higher nibble (3rd position to 4th position)
- T = set = 1 = AC
- F = Reset = 0 = NA

Used in BCD arithmatics to indicate the range exceeding conditions.

Zero :- "Is the ALU output (ACC) Zero"
- T = set = 1 = Z
- F = Reset = 0 = NZ

Zero flag is used to implement the control structures

Control Struct
(i) Selection
(ii) Iteration
(iii) Subprograms

Sign:- "Is the MSB of ALU output (ACC) one"
- T = set = 1 = NG (Negative logic)
- F = reset = 0 = PL (Positive logic)

Overflow:- "There is a carry into the MSB & no carry out of the MSB or vice versa.
- T = set = 1 = OV
- F = reset = 0 = NV

It is used in the signed arithmatic operations to indicate the range exceeding conditions.

SIGNED NOS:-
- In the signed format we can represent both positive and negative numbers.
- 2's complement no. representation is used to indicate signed nos in the processor
- when the MSB bit is 1, then the no. is negative. So take the 2's complement to conclude the value.

When the MSB bit is zero, no. is positive. So no need to take the complement to conclude the value.

n-bit 2's complement range is (-2^{n-1}) to $+(2^{n-1}-1)$

Eg 4-bit 2's complement range - {-8 to +7}

2's comple. Binary	Equivalent Decimal
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Eg.

+7	0111
+3	0011
+10	1010 ⇒ 01010 ⇒ (+10)
OV = set	OV = 1

Eg

+7	0111
-3	1101
+4	0100 ⇒ (+4)
OV = 0	OV = 0

Eg

-7	1001
+3	0011
-4	1100 ⇒ (-4)
OV = 0	OV = 0

Eg

-7	1001
-3	1101
-10	10110 ⇒ (10110) ⇒ (-10)
OV = 1	OV = 1

$OV(x,y,z) = \bar{x}\bar{y}z + xy\bar{z}$

x - MSB of OP1, y - MSB of OP2, z - MSB of Result

condition of overflow set

Q. Consider the following 2 two's complement nos. and perform arithmatic opⁿ. What could be status of the carry, sign, and overflow flags after the execution.

```
 1 1 0 0 1 1 0 1
 1 0 1 1 0 0 1 1
-------------------
1| 1 0 0 0 0 0 0 0
```

Carry = 1
Parity = 0
A·C = 1
Zero = 0
Sign = 1
overflow = 0

CONTROL FLAGS:- Based on the status of these flags, the program execution sequence is changed. Instⁿˢ that changed by user. There are 3 control flags used:

(i) Trap → 1 → Trace → Single step execution (-t)
→ 0 → Go → At a time prog " (-g)

(ii) Interrupt → 1: Enable interrupt
→ 0: Disable INTs

(iii) Direction → 1: Auto decrement (STD - Set Direct^n flag)
→ 0: Auto increment (CLD - Clear " ")

ADDRESSING MODES:-

- It shows the way where the required object is present.
- Object may be an instr / data.
- The o/p of the addressing mode is "Effective address" (EA).

- EA is the actual address where the required object is present
  i.e. object = Content of EA = [EA]
- There are different conventions used to indicate different addressing modes i.e.
  - #/I = Indicates Immediate AM
  - Register Name = Register AM
  - [ ] = Direct AM
  - @/( ) = Indirect AM
  - Indexing register name = Indexed AM

#/I → Processor supports different funct's. There are diff AM used in computer. To use diff AM, diff user ~~conventions~~. convenience codes are used during ~~exec~~ creating the program. These are abov-e mentioned.

AM are basically classified into 2 types
(a) Sequential control flow AM (focussed on Data)
(b) Transfer of control flow AM (focussed on Instr)

SEQUENTIAL CONTROL FLOW AM's:-

These ~~adr~~ AMs are focussed on data
- Data present in computer in 2 possible places i.e. processor register & memory
- ∴ sequential control flow AM further divided into 2 Types:
  (i) Register based AM's: These are used only when the data is present in the register ∴ the EA is "register name".
  (ii) Memory based AM's:
  These are used when the data present in the memory. ∴ EA is "Memory Address".

Register based AM:-

(i) Register AM:-

- This mode is used to access local variables.
- In this mode data is present in the register, the reg. name is available in the addr field of inst^{r}.

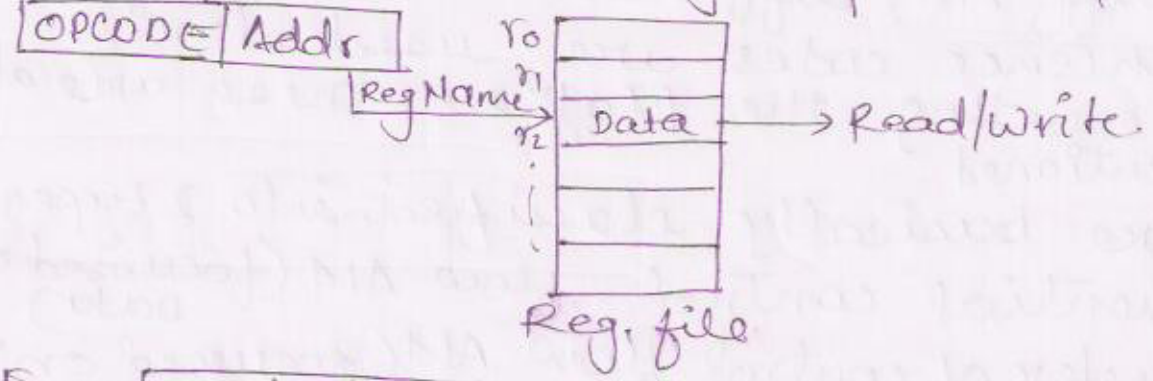
$\therefore$ EA = Addr field ~~of inst^{r}~~ value

↓

Register Name.

Data = [EA] = [Reg. Name]

1 Reg. Ref → read/write.



Eg: ADD R0 R4

↓ ↓ ↓

↑ S1/D S2

↓ ↓

reg reg

1 reg Ref 1 reg Ref

R0 ← R0 + R4

1 reg ref (write) 1 reg ref (read) 1 reg ref (Read)

(ii) Memory based AM:-

There are diff. types of AM employed under this category.

(i) Implied / Implicit AM:- In this mode the data information is present in the opcode itself.

OPCODE

↓ ↓

Type of opⁿ operand info

EA not required

STC Set carry C=1

CLC Clear carry CY=0

ADD Stack M/C. Here data is part of opcode.
POP
POP
+
Push.

NOTE:- All the zero address instr use the implied addressing mode.

(ii) Immediate:-

- This mode is used to access the constants.
- In this mode the data present in the address field of instr.

i.e. [OPCODE | Address] ⇓ Data

Data part of instr.

Eg: [ADD | R_0, #23] $R_0 \leftarrow R_0 + 23$

ADD → +, S_1/D → reg., S_2 → Imme.

NOTE:- The limitation of immediate AM is the range of possible const. are restricted based on the size of the addrs field. i.e. if the addrs field size of the instr is n-bit, then the possible range of unsigned constants are $\{0 \text{ to } 2^n - 1\}$, Possible range of signed constants are $\{-(2^{n-1}) \text{ to } +(2^{n-1} - 1)\}$.

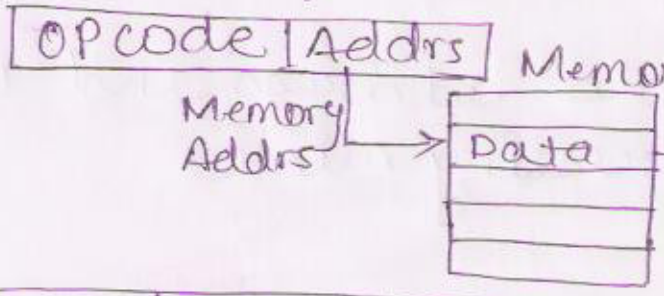
(iii) Direct AM (Absolute AM):-

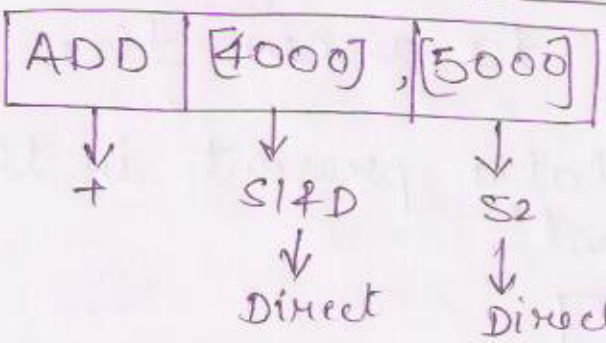
- This AM is used to access static variables.
- In this mode data present in the memory, that memory address available in the addrs field of the instr.

∴ EA = Addrs field value ⇒ Mem. Addrs.

Data = [EA] = [Memory Addrs]

- 1 Mem. ~~ac~~ reference is req. to read/write data to/from the memory by using this AM.





M[4000] ← M[4000] + M[5000]
1 Mem Ref 1 Mem Ref 1 Mem Ref
(3 Mem. Ref)

ADD	[4000] ,	R0

⇒ (2 Mem Ref)

MOV	[5000] ,	[4000]

⇒ (2 Mem Ref)

(iv) Indirect AM :-

- This is used to implement the pointer concept.
- In the indirect AM, data present in memory that memory Addr [EA] may be present in register/memory.

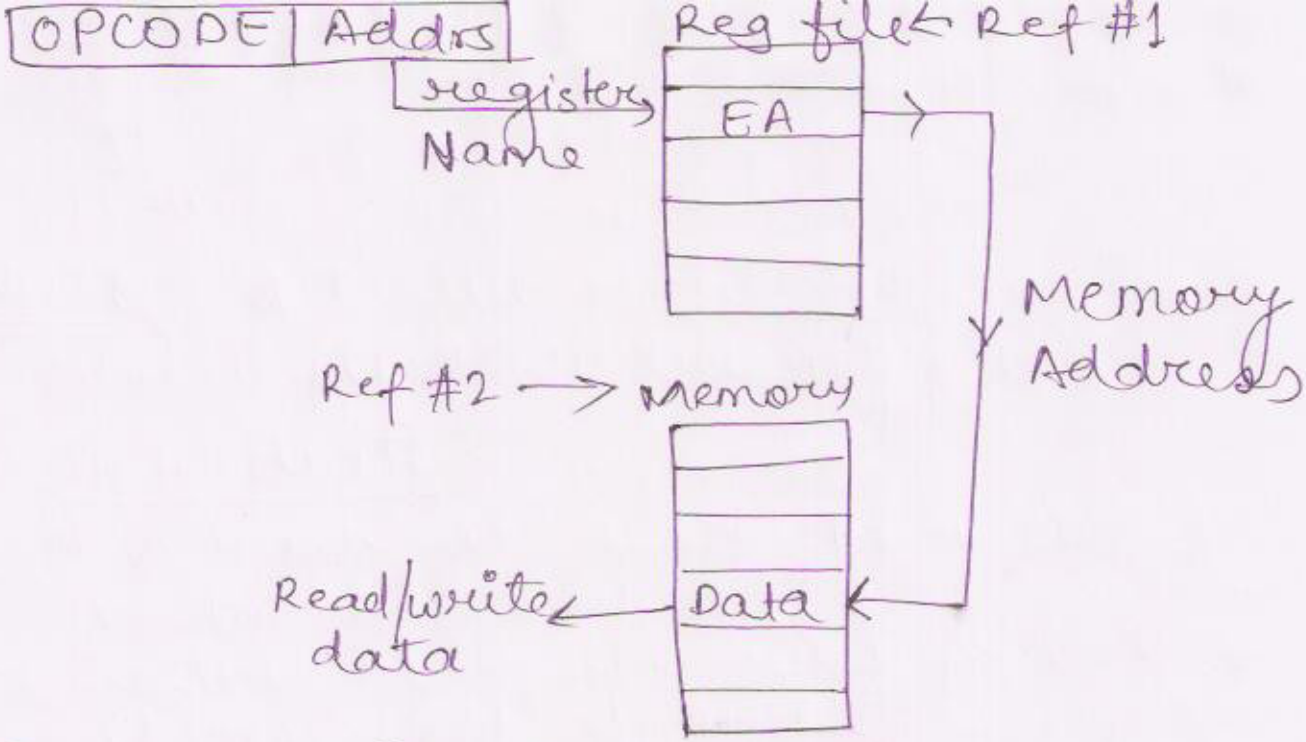
∴ Based on the availability of EA ~~the~~ indirect AM is further divided into two types :-

(iv)(a) Register indirect AM :- In this mode, EA is present in register, that reg. name is available in the addrs field of the instr.

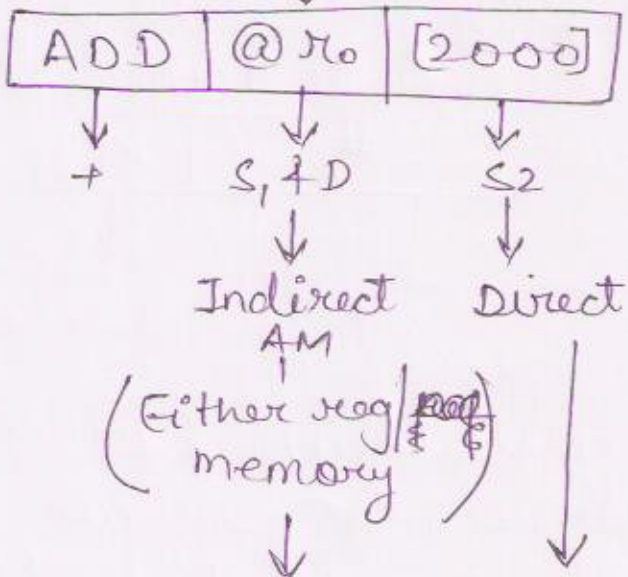
∴ EA = [Addr. field value]
↓
Reg. Name

Data = [EA]
= [[reg. Name]]

2 Ref :-
1 Reg. ref → EA
1 Mem ref → Read/write



Eg:- ADD @ r_0, [2000]



$M[[r_0]] \leftarrow M[[r_0]] + M[2000]$

1 reg. ref = EA
1 mem ref = write

1 reg ref = EA
1 mem ref = read

1 mem. ref (read)

(3 references required)

(3) Memory Indirect Addressing Mode:-

In this mode the effective address is present in the memory, that memory address is available in the address field of the instruction.

$EA = [\text{Address field value}]$
↓
Mem. addr.

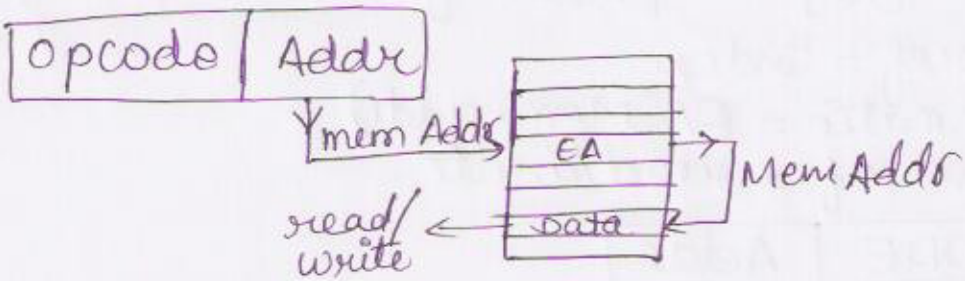
Register indirect is faster.

$data = [EA] = [[\text{mem Add}]]$

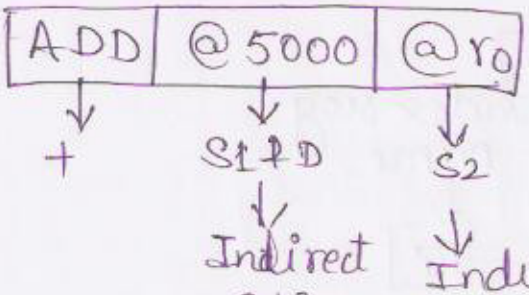
Imem. refer = EA

1 mem. ref = read/write

2 mem ref.



Eg:



⇒ 5 mem. ref.

$M[[5000]] \leftarrow M[[5000]] + M[[r_0]]$
(2) (2) (1)

Indexed Addressing Mode

It is used to implement the arrays. Array is an ordered set of homogeneous data elements. Array is always stored in the sequential memory locations.

To access the array element, there is a need of 2 parameters i.e. (i) Base address (ii) Index value.

In the indexed mode, EA is calculated by adding the index value to the base address. i.e. EA = B.A (+/-) index value.

BA → starting or ending address of the array.

Index → Position of the array element.

Processor supports one special purpose register called as Index register (R_i). It is used to hold the index value.

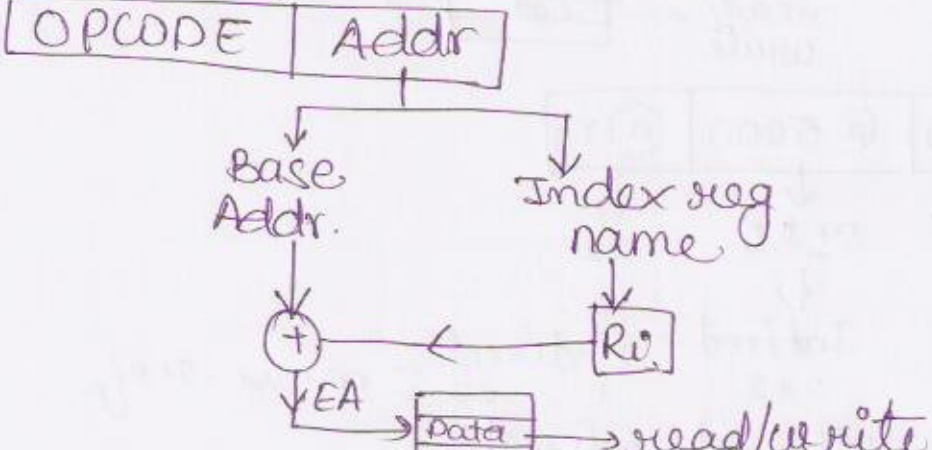
Base address is present in the address field of instr. ∴ EA = Addr. field value + $[R_i]$

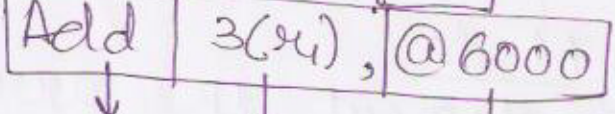
Data = [EA] = [Addr field value + $[R_i]$]

1 reg ref = Index

1 arithmatic = EA (Mem. addr)

1 Mem. ref = read/write.



Eg 

r_1 = Index reg

+ S1 D S2

Indexed Indirect

$M[3+[R_1]] \leftarrow M[3+[R_1]] + M[[6000]]$

$=$

1 reg ref = Indx
1 Arith = EA
1 M ref = ~~read~~ read

1M ref → EA
1M ref → read

Indexed addressing mode is also called as based indexed addressing mode.

NOTE:- Some of the processors support base and index register used to hold the base and index value respectively.

Eg:- 8086 µP supports 2 registers namely

BX → Base reg.

SI/DI → Index reg. (Source/Destination Index)

EA = [Base Reg] + [Indx reg]

↓ Base Addr ↓ Index value

Data = [EA]

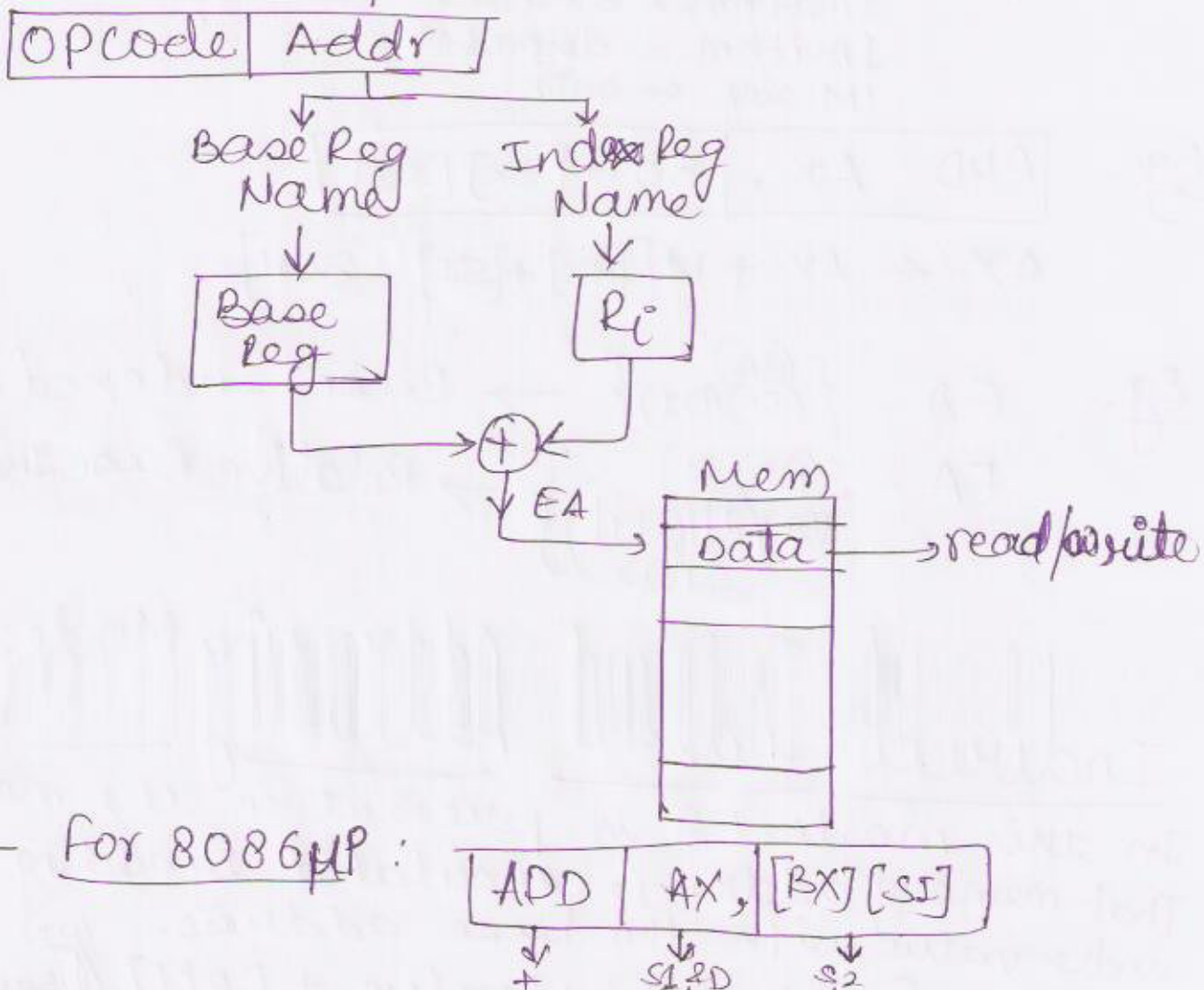
= [[Base Reg] + [Index Reg]]

Ireg ref = Base Addr

Ireg ref = Index

IArithm. = EA

IM ref = read/write



Eg:- For 8086µP :

AX ← AX + M[[BX]+[SI]]

1reg ref (write) 1reg ref (read)

1reg·ref = BA
1Reg·ref = Index
1Arithm· = EA
1M·ref = read

- 8086 processor supports another kind of addressing mode, in which 8 bit or 16 bit displacement is maintained along with the base index & index register names. This mode is called as Relative Base Indexed AM.
- In this Mode the EA is calculated as

EA = [Base reg]+[index reg]+ 8bit/16bit displacement

Data = [EA]

= [[Base]+[Indx]+ 8bit/16bit disp]

1reg ref = Base
1reg ref = Index
1Arithm. = B+Index
1Arithm. = dispadd^n
1M·ref. ⇐ Data

Eg:- | ADD AX, | 80H[BX][SI] |

AX ← AX + M[[BX]+[SI]+80H]

Eg:- EA = {(BA)(SI)(DI)} → Base Indexed AM

EA = {(BA)(SI)(DI)(8/16 bit disp)} → Relative Base Indexed AM

Indirect Indexed Addressing Mode:-

In this mode EA is present in the memory. That memory addr. is calculated by adding the index value with the base address. (or)

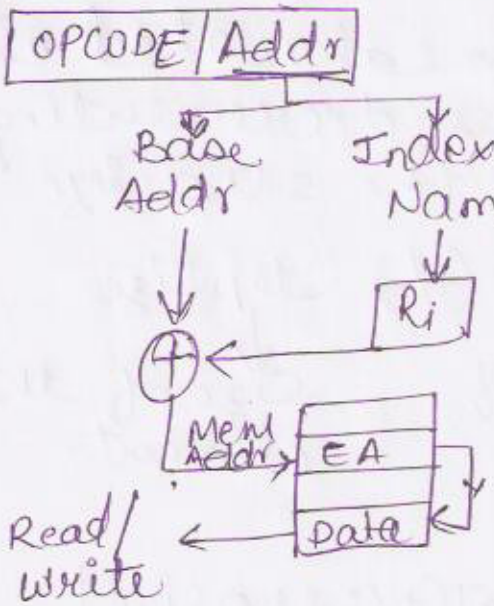
∴ EA = [Address field value + [RI]] / [[Base Addr] + [Ri]]

Data = [EA]

= [[Addr. field + [Ri]]] (or) [[[Base Reg] + [Ri]]]

1 Reg ref = Index
1 Arith. = Mem Addr where EA is present
1 Mem Ref = EA
1 Mem Ref = Read/write

1 reg ref = Base Reg.
1 reg ref = Index val.
1 Arithm = Mem Addr, where the EA is present
1 M. ref = EA
1 M ref = Read/write



Not in use, beoz it takes more time.

Indirect Indexed Addressing mode takes more # cycles to read/write data to/from memory.

Eg: ADD | 3(R1), @6000

+ ← S1 → D S2

S1 → Indirect Indexed AM.
S2 → Indirect AM

M[[3+[R1]]] ← M[[3+[R1]]] + M[[6000]]

1 reg ref = Index
1 Arithm = Mem Addr
1M. ref = EA
1M ref = write

1 reg → R1 (index)
1 arithm → Mem Addr
1M ref → EA
1M ref → read

1 Mem → EA
1 Mem → read

NOTE:- Indexed addressing Mode is used to access the random array element based on the index value.

Auto Indexed AM. It is used to access the linear array elements. To access the linear array elements, there is a need of base address.

- Base address ~~is~~ ~~be~~ indicates the starting or ending of an array.
- Base reg. is used to hold the base addr. In this mode EA is calculated either by incrementing or by decrementing the base address with the step size. i.e.

$$EA = [\text{Base Reg}] \; (+/-) \; \text{Step Size}$$

[Base Reg] → Base addr of an array

Step Size → Size of the data element.

$$\text{Data} = [EA]$$
$$= [[\text{Base reg}] \; (+/-) \; \text{Step Size}]$$

1 reg. ref. = Base Addr.
1 arithm = EA
1 M. ref = read/write

The classification of auto Index AM is

1. MOV r_0, +(r_4)
2. MOV r_0, (r_1)+
3. MOV r_0 -(r_1)
4. MOV r_0 (r_1)+

where R_1 = Base register.

Q. Consider a hypothetical CPU. It uses different operands accessing modes.

operand & accessing mode	Frequency (%)
Register	30
Immediate	20
Direct	22

Mem. Indirect = 17
Indexed = 11.

Assume the 2 cycles required for memory access, One cycle consumes for arithmatic operations & zero cycles consumes when the operand is coming in the register or instruction itself. What is the average operand fetch rate of the processor?

Ans :- Average operand fetch rate = $(0.3 \times 0) + (0.2 \times 0) + (0.22 \times 2) + (0.17 \times 4) + (0.11 \times 3)$

$= 0.44 + 0.68 + 0.33$

$= 1.45$ cycles

Transfer of control flow AM :-

While execution of control structures i.e. selection statements (goto, if-then-else, switch etc), iterative statements (while, do-while, for loop etc) and subprogram concept, the program control is transferred from 1 location to another location.

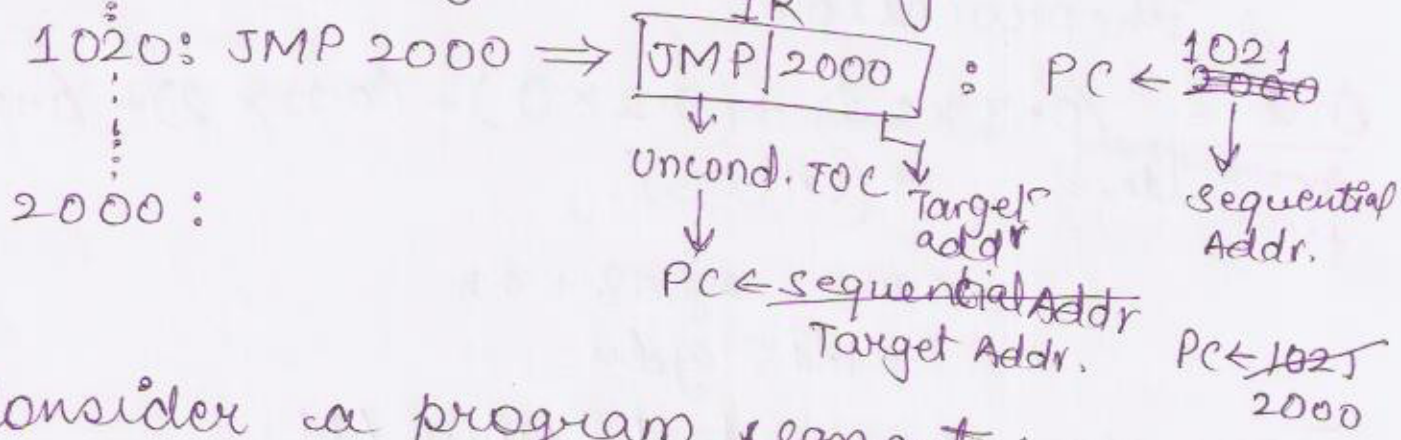
- To implement the control structures, process or supports special type of instructn called as transfer of control flow.
- Transfer of control is associated with the target address.
- While execution of the TOC operatn, the control is tranferred from current location to target location.
- There are 3 possible mneumonics used to implement the TOC opn.
  (i) Branch
  (ii) Jump
  (iii) Skip

TOC op^n classified into 2 types.

(i) Unconditional TOC

(ii) Conditional TOC

Unconditional TOC : While execution of these instr, without checking any cond^n, the control will be transferred from current location to target location. Eg. JMP 2000



Consider a program segment:

1000: I_1
1001: I_2
1002: I_3 (JMP 1020)
1003: I_4
1020: BI_1
1021: BI_2 (JMP 1012)
1022: BI_3

Execution sequence of program is:-

-t=1000

PC = 1000
1001
1002 : JMP 1020
~~1003~~
1020
1021 : JMP 1012
~~1022~~
1012
1013

NOTE: Uncond. TOC operations are used to implement the goto statement halt instr and M/C control instr^n.

HALT: It is implemented using the uncond TOC with starting addr as the target addr.

Eg: 1000: I_1
1001: I_2
1002: I_3 (Halt)
1003: I_4

-t = 1000

PC = 1000
1001
1002 : Halt : JMP 1002
~~1003~~
1002: halt
~~1003~~
1002: halt
⋮

M/C control inst^r:-

Uncond^n TOC op^n are used to control the functionality of some specific M/Cs.

Eg: Traffic light controller.

```
L1: G: ON
    Y: OFF
    R: OFF
    call delay()
    G: OFF
    Y: ON
    R: OFF
    call delay1()
    G: OFF
    Y: OFF
    R: ON
    call delay2()
    Jmp L1.
```

Conditional TOC:

While execution of this instruction, the associated cond^n undergoes evaluation. When it evaluates to true, then the control is transferred to the target location, otherwise the next sequential inst^r is executed.

Eg: JNZ | 2000 | PC ← seq. Addr. cond: NZ

JNZ → Cond^n TOC; 2000 → ~~Target~~ addr

NZ ↓ comp status of previous inst^r

false → True: PC ← ~~Sequential~~ target Addr

PC ← sequential Addr

Consider the program segment:

```
1000 : MOV r0, #4
1001 : ADD r1, r2
1002 : DEC r0
1003 : JNZ 1001
1004 : MOV
```

Execut^n Seq:-

$-t = 1000$

```
PC : 1000 : r0 = 4
     1001 : +
     1002 : r0 = 3 (NZ)
     1003 : NZ (True)
     1004
     1001 : +
     1002 : r0 = 2
```

```
1003 : NZ (True)
1004
1001 = +
1002 = r0 = 1
1003 = NZ (True)
1004
1001 = +
1002 = r0 = 0
1003 = Zero (false)
1004 = MOV.
1005 :
```

NOTE:- Cond^{nal} TOC oper^{ns} are used to implement the cond^{nal} select^n statmnts & iterative ~~select^n~~ stmts.

Subprogram:- Subprogram is a reusable program. That means when the same function-ality is repeatedly occurred in the main app^n, develop the functionality related code only once (Subprogram) and call the functionality many times in the main app^n.

The advtage of subprogram is it reduces the memory space, development time & development cost.

Characteristics of subprogram:-

(i) Single entry point - Single exit points
(ii) Main program is suspended during the execution of the subprogram.
(iii) Control will be transferred to the main program after completion of the subprogram

IMPLEMENTATION:-

To implement the subprogram concept, the processor supports pair instructions.
i.e. CALL - RET

CALL → Entry → Main prog. → Internally PUSH
RET → Exit → subprog. → Internally POP

Consider the program segment

Main prog
1000 : I_1
1001 : I_2
1002 : I_3 (CALL 2000)
1003 : I_4

Subprog
2000 : BI_1
2001 : BI_2
2002 : BI_3
2003 : BI_4 (RET)
2004 : BI_5

t = 1000

without stack unsuccessful implementation

PC : 1000
1001
1002 | CALL | 2000
1003 ← (wait)
2000
2001
2002
2003 | RET |
2004
1003

with stack

1000
1001
1002 | CALL | 2000 → Push 1003
1003 ←
2000
2001
2002
2003 | RET | → pop 1003
2004 ←
1003

In the above execution sequence, due to the lack of ~~return statements~~, RET address during the execution of RET inst^n, CPU enters into the WAIT state. ∴ It is not

a successful implementation of subprogram.
- To make it successful, need of return address..
- Return address is Next instr address after the CALL instr.
- Runtime stack is used to store the return addresses
- During the execution of program, when the CPU encounters the CALL instr, it pushes the P.C. value into the stack Later target address is placed into PC
- During the execution of subprogram, when the CPU encounters the RET instr it invokes the pop operation to restore PC with return address.

1. JMP 2000 PC ← ~~Seq. Addr~~ Target "
2. JNZ 2000 T ⇒ PC ← ~~Seq. Addr~~ target "
   F ⇒ PC ← Seq. Addr.
3. CALL 2000 : PC → stack
   PC ← 2000
4. RET PC ← ~~Seq Addr~~ TOS (Return Addr)

TOC instr further divided into 2 types:-
(i) direct TOC & (ii) Indirect TOC.
In direct TOC, the target addrs. is available in instruction itself. whereas in the indirect TOC, the target addrs is not present in the instr.

TOC	uncond.	cond.
Direct	JMP, CALL, goto Halt	JNZ, CALLZ, CALLNZ, CALLC, for, while, do-while, if, switch
Indirect	RET	RETZ, RETC, RETNC, RETNZ

In the transfer of control flow AM category, two addressing mode are employed:-

(a) Relative / PC relative AM

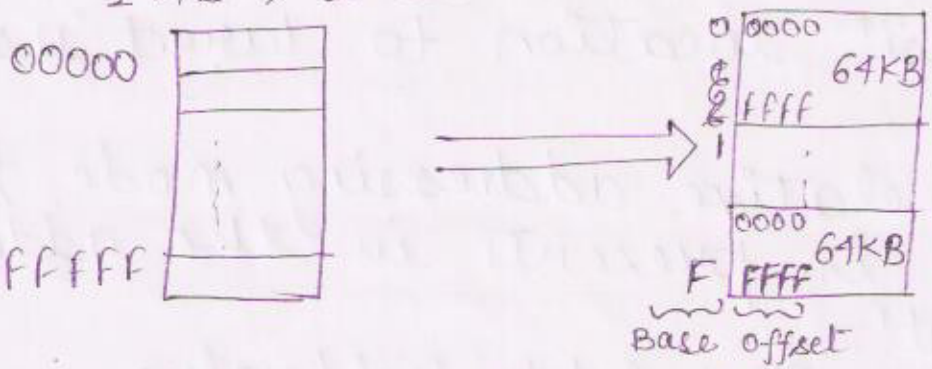
(b) Based / Base reg AM.

→ Inst' are always stored in memory. So to access the instruct's efficiently, there is a need of understanding the memory org.

→ Consider 8086 mem org. It contains 1MB physical memory. The physical memory is divided into 16 logical segments with each segment size of 64 KB. (No change in size)

$1MB \Rightarrow 2^{20}$ cells.

$1MB = 16 \times 64KB = 2^{20} B.$



change in organizat^n

Physical addr = BA + Offset Addr.

In 8086, segment registers are used to hold BA and pointer or Index register is used to hold offset address.

Segment Registers → CS, DS, SS, ES

Pointer/Index Reg → IP, BP, SP } Pointer Reg

→ SI, DI } Index Reg

If CS=0 ⇒ 00000 } Instⁿ
          0FFFF }

If DS=2 ⇒ 20000 } Data
          2FFFF }

To access code ⇒ CS + IP Notation CS:IP
To access data ⇒ DS + BP/SI/DI DS:BP/SI/DI

Relative / PC-relative AM:-

When the target instr is present in the same segment, then the control will be transferred within the segment (Intrasegment TOC)

In the intrasegment TOC the base address is common. So only 1 parameter is required to calculate the target instr addr. ie. relative value.

Relative value is the distance b/w the current instr location to target instr location.

In the PC-relative addressing mode, the relative value is present in the address field of instr.

∴ EA = PC + Addr field value.

Next instr Addr ↓ Current Instr Addr ↓ relative value

PC ← PC + Addr. field value.

PC relative addressing mode is suitable for program relocation at runtime.

Base register AM

→ When the target inst^r is present in different segment, then the control is transferred from current segment to target segment (Intersegment transfer of control)

→ In the intersegment TOC, to calculate the Target inst^n address, 2 parameters are required (i) Base Addrs (ii) relative value.

→ In the base register A.M., the Base Addr is maintained in the base register & relative value is present in the address field of instruction.

$\therefore$ EA ← [Base Reg] + Addr. field value

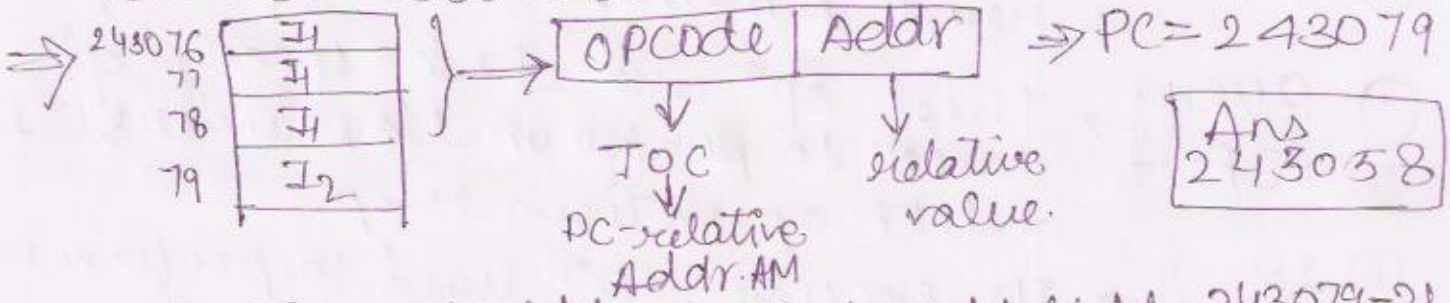
- EA: Nxt ins^n addrs.
- [Base Reg]: Base Addr of the target segment
- Addr. field value: Relative value

$\Downarrow$

PC ← [Base Reg] + Relative value.

Base register AM is suitable for write position independent codes (subprograms).

Q. A 3 byte long PC-relative ins^n is stored in memory with starting address 243076 (decimal). The signed displacement present in the addrs field value is −21. Then what is the branch address?



$\therefore$ Target Addr := PC + Add field = $243079 - 21 = 243058$

INSTRUCTION SETS :-

Processor supports 3 catagories of ins^n.

(i) Data Transfer Ins^n
(ii) Data manipulation Ins^n
(iii) Transfer of control Ins^n

(i) Data Transfer Instruction

While execution of this instr, data is transferred from 1 location to another location. The limitation of data transfer operation is source may or maynot have the storage functionality but the dest^n always has the storage.

~~Immediate addrsing mode may be for~~

There may be different data transfer operations possible:

(1) MOV :- General purpose data transfer instruction.

Source	Dest^n
reg	reg
Reg	Mem
Mem	reg
Mem	mem
Immediate	reg
Immediate	mem.

(2) LOAD
(3) STORE → These are memory specific operat^{n}s used to perform the memory read & memory write op^n resp.

(4) PUSH
(5) POP → These op^n are stack specific op^n used to perform the insert & delete operations resp.

(6) IN
(7) OUT → I/O specific op^n used to perform the I/O read & I/O write op^n resp.

(2) Data Manipulation Instructions

While execution of this inst^n, one form of data is converted into another form

1. Arithmatic (ADD, SUB, MUL, DIV, INC, DEC)
2. Logical (AND, OR, NOT, XOR, CMP)
3. Shift and Rotate Instructions.

NOTE:- During the execution of arithmatic op^n, the carry flag will be used except in the INC and DCR operations. INC 255 $\Rightarrow$ 255, DCR 0 $\Rightarrow$ 0

CMP:- It is used to compare the 2 entities. During the execution of CMP inst^n, it invokes the subtraction operation.

$\rightarrow$ The O/P of the CMP inst^n will be maintained in 0 flag & carry flag.

Eg:- | CMP | r_0, r_1 | ; $r_0 = 80H$, $r_1 = 80H$

Invokes SUB

$r_0 - r_1 = 0 \Rightarrow$ zero flag = 1 (set)

After the CMP execution, if the zero flag is set, i.e. both the operands are equal

Eg:- | CMP | r_0, r_1 | $r_0 = 80H$, $r_1 = 40H$

```
  1000 0000
- 0100 0000
-----------
  0100 0000
```

$\Rightarrow$ Zero flag is reset, carry flag is reset.

After CMP execution, if zero flag & carry flag both are reset, that means 2nd operand is less than the first operand

Eg3: $r_0 = 40H$ $r_1 = 80H$

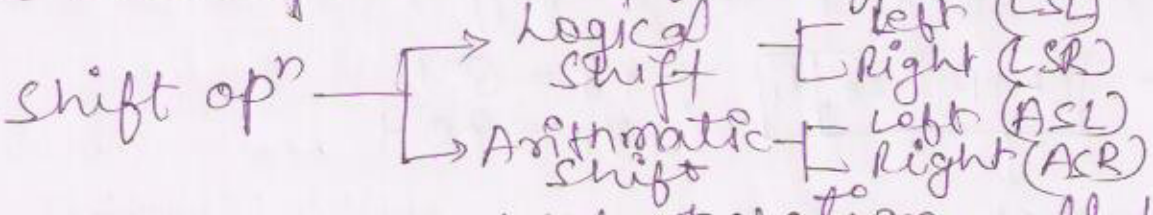
01000000
10000000 (0) zero-reset
11000000 set (1) carry

After CMP execution, if zero flag is reset & carry flag is set. that means 2nd operand is greater than 1st operand.

Shift operation:-

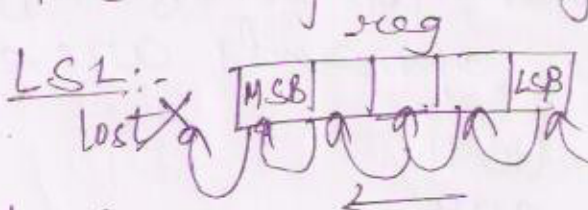
→ while execution of shift operation, data is transferred towards left or right bit by bit with loss of data.

→ The classification of shift operation is

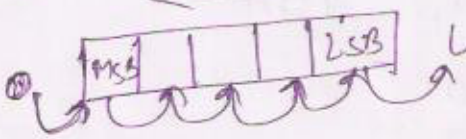


In the logical shift operation all the bits are moving towards left or right including the sign bit.

LSL:-



LSR:-



Eg. LSL r0, #2 $r_0 = 83H$

1000 0011
← 2 times
Ans. ⇒ 0000 1100
0CH

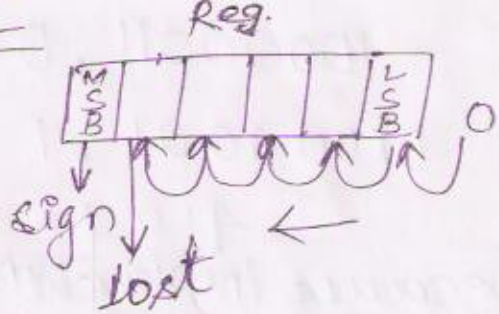
Eg. LSR r0, $r_0 = 83H$

1000 0011
0100 0001 r_0 41H.

Appl^n :- ~~Transfer data into another~~ Used in the serial data comm^n. to transfer the data bit by bit.

In the arithmatic shift operations, all the bit are moving towards left or right except the sign bit.

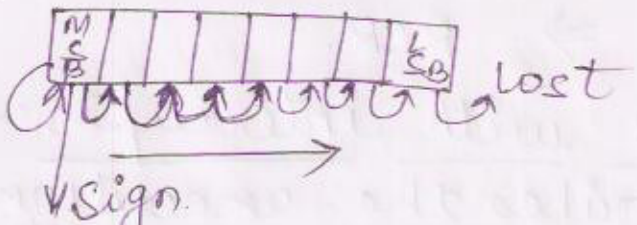
ASL



Eg:- ACL %0 r_0 = 83H

1000 0011
1000 0110
86H

ASR



Eg:- r_0=83

1000 0011
1100 0001
(C1)H.

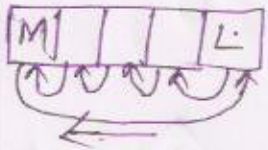
App^n:- Used in signed arithmatic.

Rotate Instruction:- While execution of rotate Inst, all the bits are moving towards left or right without loss.

The classification of rotate operations is

- without carry
  - left (ROL)
  - Right (ROR)
- With carry
  - Left (RCL)
  - Right (RCR)

ROL:-

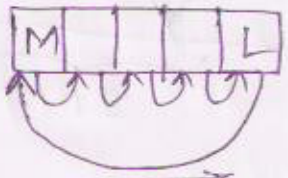


No new data is added
Existing data is not lost.
only position is changed.

Eg:- 83H

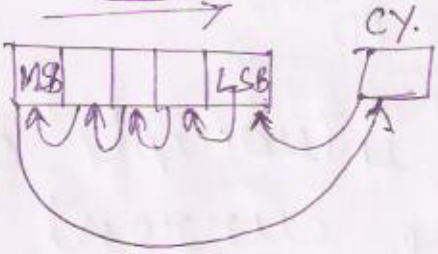
1000 0011 → (07H) 00000111

ROR:-



10000011 → 11000001
(C1)H

RCL:

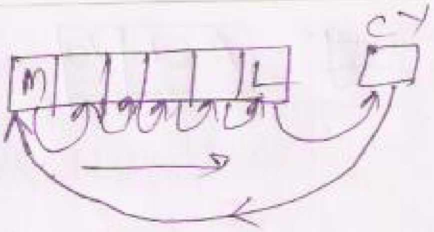


1000 0001 C-0

⇒ 0000 0110 C→1

06H ↓C→1

RCR:-



10000011, 0

01000001 ;1

41H fC 71.

App^n:- optimization because implement^n is simple.

Motor movement ⇒ ROL or ROR.

even odd ⇒ RCR.

Instruction cycle with interrupts:-

This concept describes the execution sequence of instructions along with the interrupts.

Interrupt is an unusual event to disturb the normal flow of execution.

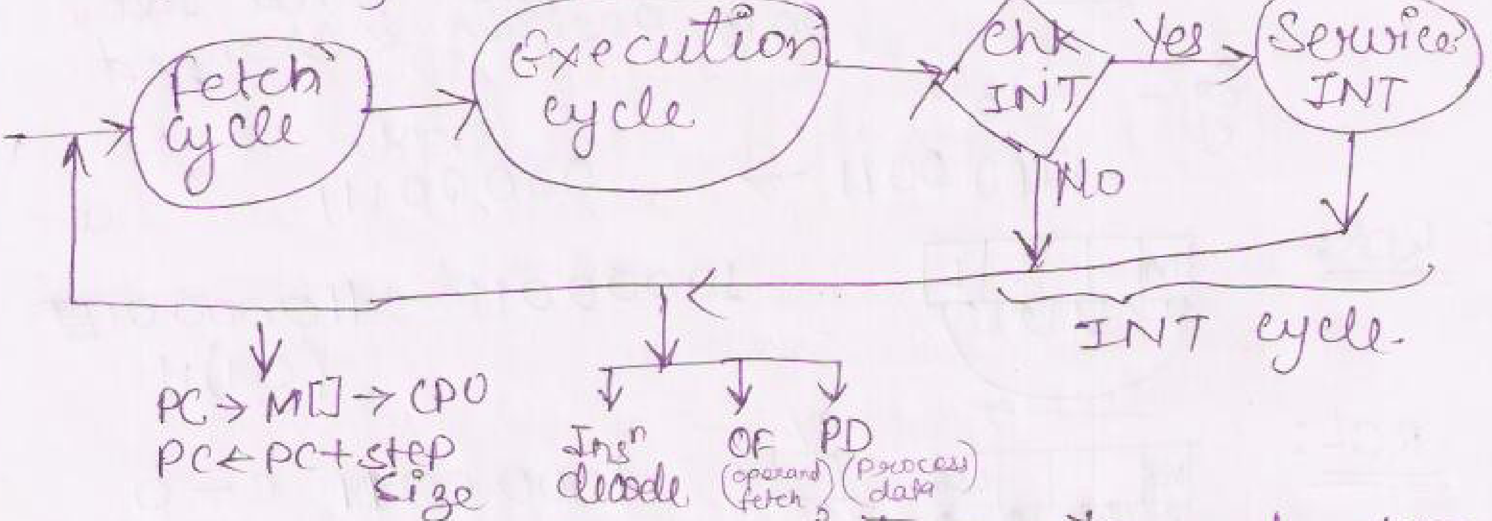
Instruction cycle with interrupt consists of 3 subcycles:

(i) fetch cycle

(ii) Execution cycle

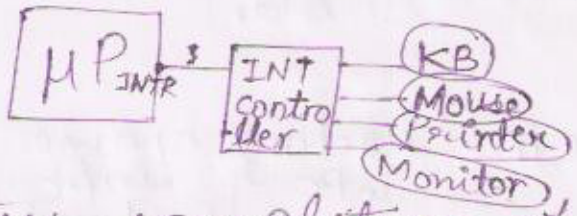
(iii) Interrupt cycle

The sequence diagram of interrupt handling process is



The CPU responds to interrupts only after completion of the current instruction execution.

When the interrupt flag is ENABLED that means interrupt sources are enabled. So interrupt cycle is required to service the interrupts. Otherwise, interrupt sources are disabled, so no need of interrupt cycle.



After completion of every instr execution the CPU reads the status of the interrupt sources to identify whether the interrupt is present or not.

INTR — 1 ; INT is present
0 ; INT is not present.

- When the int are not present, then the CPU fetch the next inst. from the memory.
- When the int is present, the CPU push the program counter value into the stack and transfers the control to the INTERRUPT VECTOR TABLE.
- IVT is a part of memory. It is used to store the interrupt subprograms.
- There are multiple interrupts possible in the processor. So different INT subprograms are required to service the different INT.
- Each INT subprogram must be end with IRET/RETI instruction.
- During execution of INT subprogram, when the CPU encounters the RETI instr it invokes the POP instr to restore the PC with return address.

∴ After service the interrupt, the CPU fetch the next instr based on the PC:

	Data Transfer	ALU	Uncond TOC	Cond^{n} TOC
Fetch cycle	PC ← Seq. Addr.	PC ← Seq. Addr	PC ← Seq.	PC ← Seq.
Execution cycle	NO change in PC	No change in PC	change in PC PC ← target Addr	change or Not PC ← changed or Not
Int Service	Yes RA : Seq. Addr (return Addr.)	Yes RA : Seq. Addr	yes RA : target Addr	Yes RA : Changed or same

(4D) Types of interrupts : [Mainly → H/w & S/w INT — rest are part of these]

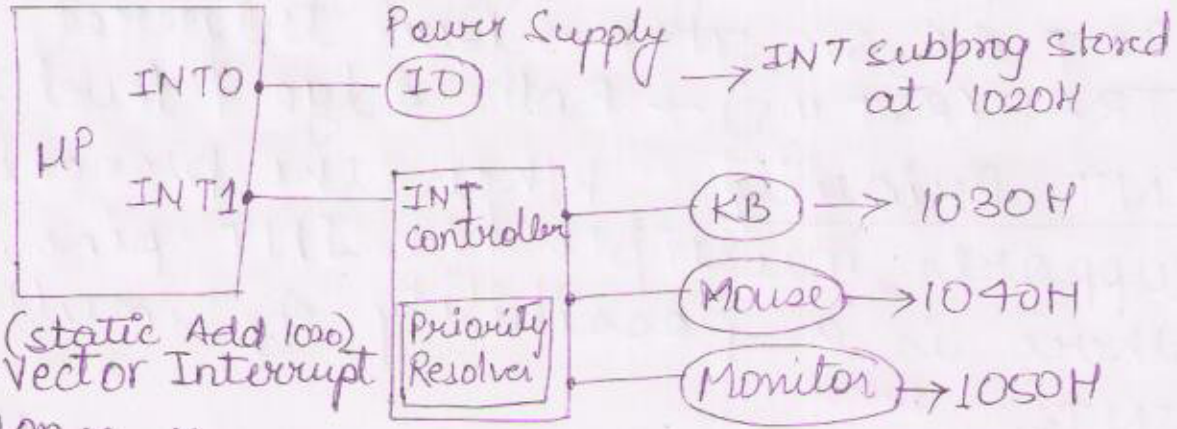
1. Hardware INT :- Hardware int. are present at the H/w pins of the processor. H/w INTs are also called as the ext. int. & internal int.
External INTs are generated by the ext. sources i.e. Power supply, basic IO devices etc.
Internal INTs generated by the internal components of the processor i.e. invalid OPCODE, divide by zero etc.
2. Software INT :- S/w INT is an instr. It is defined in the instr set of the processor. Eg :- Supervisor call.
3. Maskable INT :- Maskable INT pins are enabled or disabled based on the signals. All the basic IO devices are connected to maskable INT pins.

4. Non Maskable INT:- Here, the NMI pin is always present in the enabled state. So critical conditions of the processor i.e. power failure & temp sensor etc. are connected to the NMI pin.

5. Vector Interrupt:- Vector INT is associated with fixed vector point (static vector Addr)

6. Non Vector INT:- It is associated with dynamic vector point (or) means vector ~~point~~ addr is supplied by the source.

Eg.



7. Level triggered INT:- This INT is enabled based on the clock signals

8. Edge triggered INT:- These INT are enabled either with raising edge transition or with falling edge transition.

INT structure:- To analyse the implementation of various INTs, let us consider 8085 processor as a reference component.

8085 H/W INT

1) RST 7.5 (003CH)
2) RST 6.5 (0034H)
3) RST 5.5 (002CH)
4) RST 4.5 (0024H)
5) INTR (TRAP) (NMI)

1, 2, 3, 5 → Maskable

S/W INT – RST 0-7.

For S/W INTs, maskable and non-maskable is not applicable because inst^{n} cannot be enabled or disabled.

INTR – Non vectored INT.

other 4 h/w INTs are vectored.

All the s/w INT are vectored.

RST 0 - 0000 RST 2 - 0010 RST 4 - 0020 RST 6 - 0030
RST 1 - 0008 RST 3 - 0018 RST 5 - 0028 RST 7 - 0038

for S/W INTs, level Triggering and edge Triggering is not applicable.

RST 7.5 → edge Trigged
RST 6.5, 5.5, INTR → level triggered
TRAP (RST 4.5) → Both edge & level triggered

INT Priority: When the processor supports multiple h/w INT pins, then there is a possibility of simultaneous INTs.

- TO service only one INT at a time, there is a need of priority levels.

1. TRAP (RST 4.5)
2. RST 7.5
3. RST 6.5
4. RST 5.5
5. INTR

↓ High Priority ↓ Low

Q. A certain µP supports 4 INTs, I_1, I_2, I_3, I_4.
I_1 → priority decreases. The CPU responds to INT in every 25 µsec. The service time of INTs are 20 µsec, 40 µsec, 30 µsec, 25 µsec resp. The INTs may or may not occur simultan. What is the possible range of time required to execute the INT4.

⇒ Min ⇒ 27.5 MS Max – 105 + 10 = 125 MS.

(*) If many INTs are present simultaneously, after servicing one INT, an instr is executed. Only after execution of an instr, next INT is serviced.

Ans to Ques:-

min ⇒ (without simult) → 2.5μs. with simul → 2.5 μs.

max ⇒ 125 μs with simul → 2.5 μs.

RISC and CISC:-

Chars. of RISC → 5

(i) It supports more no. of registers
(ii) It supports less no. of addressing modes.
(iii) It supports fixed length instructions.
(iv) One instr per cycle - CPI = 1 (cycles per Instr)
(v) It supports successful pipeline implementation.
(vi) Motorola, PowerPC, ARM (Advd. risc M/c) etc are eg. of RISC processors.

Characteristics of CISC:-

(i) It supports less no. of registers.
(ii) It supports more no. of addressing modes.
(iii) It supports variable length instrs
(iv) There is not one instr per cycle. CPI ≠ 1.
(v) Supports unsuccessful pipeline implementation.
(vi) Eg - Pentium.

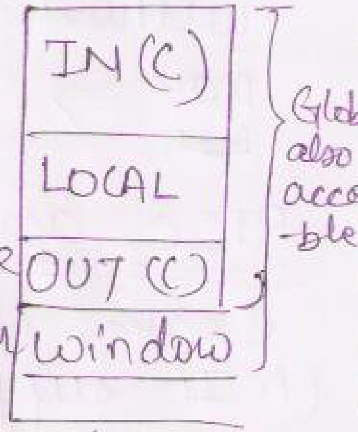
Register organization in RISC CPU:-

- In the RISC CPU, the registers are organized as overlapped register windows
- The risc processor registers are classified

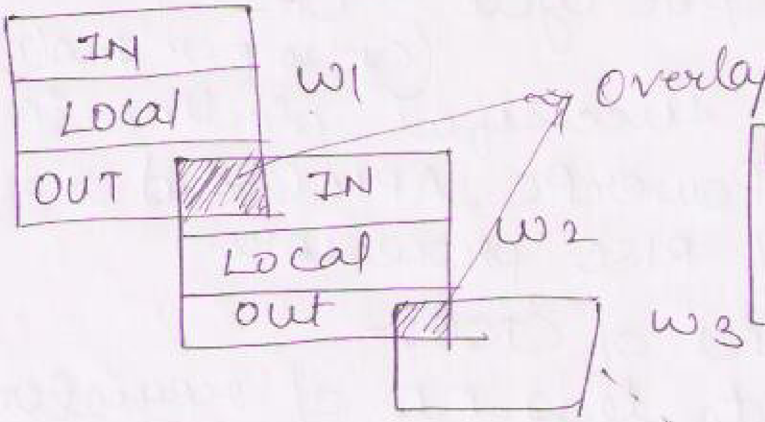
-ed as
→(i) Global reg (G) (iii) IN reg. (C)
(ii) Local reg. (L) (iv) OUT reg. (C)

- The register window contains 3 set of registers i.e. local, IN and OUT reg. The global registers are accessible by all the register window.

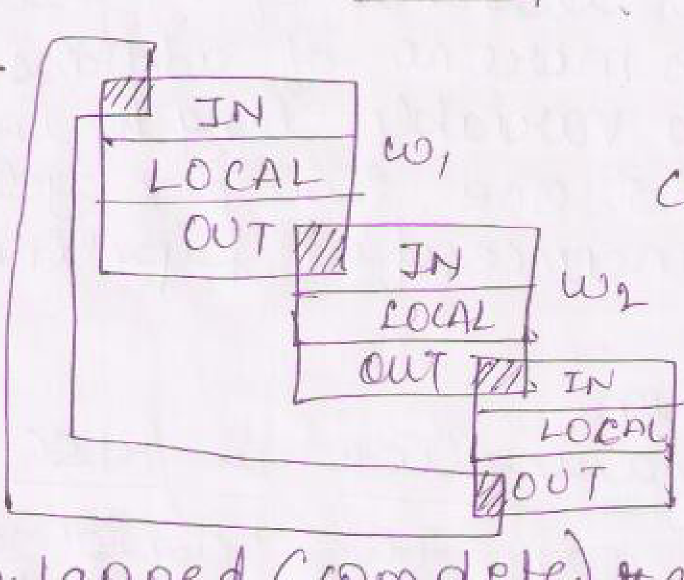
- Window size = L+2C+G.



In the RISC CPU, all the system supported registers are organized as overlapped register windows. Overlapping means 1 window OUT registers are used as IN registers in the another window.



Solⁿ:-



complete overlapping

Here.
W1 (out) = W2 (IN)
W2 (out) = W3 (IN)
W3 (out) = W1 (IN)

In overlapped (complete), register file size = No. of window * (L+C) + G

Components of computer:-

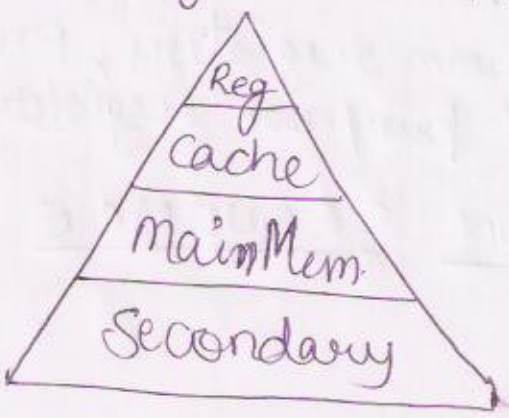
The computer contains 3 basic components.
(i) CPU (ii) Memory (iii) I/O

CPU ORGANIZATION:

CPU functionality is processing the inst'.
It consists 3 basic components (i) Registers (ii) ALU (iii) Control units.

REGISTERS:- Reg. are used to store the frequently referred data.

- During the execution of the program, there is a frequent communicath between CPU and memory.
- Due to slow working of memory component, the CPU performance degrades
- To avoid this problem, memory heirarchy design is used in the system design.
- The objective of memory heirarchy is to balance the bandwidth between the CPU and memory. (or) to minimise the speed gap between CPU & memory. (or) sync the data transfer rate b/w CPU & memory.
- Memory heirarchy describes the way of organization and way of accessing the system supported memories.



Bottom to top approach of memory heirarchy is

(1) Capacity decreases
(2) Access time reduces
(3) Cost (Expensive) increases.
(4) $\left(P\alpha \frac{1}{\text{Exec. Time}}\right)$ (Performance) $\cdot$ $(ET\alpha Acc.Time)$ $(P\alpha 1/AT)$

5) Performance increases.
6) ~~Expensive~~ Cost/Bit increases.

NOTE: 7) During the execution of instruction, data becomes the most imp. So use the smallest storage (register) component to hold the frequently accessed data. ∴ The processor maintains minimum set of registers.

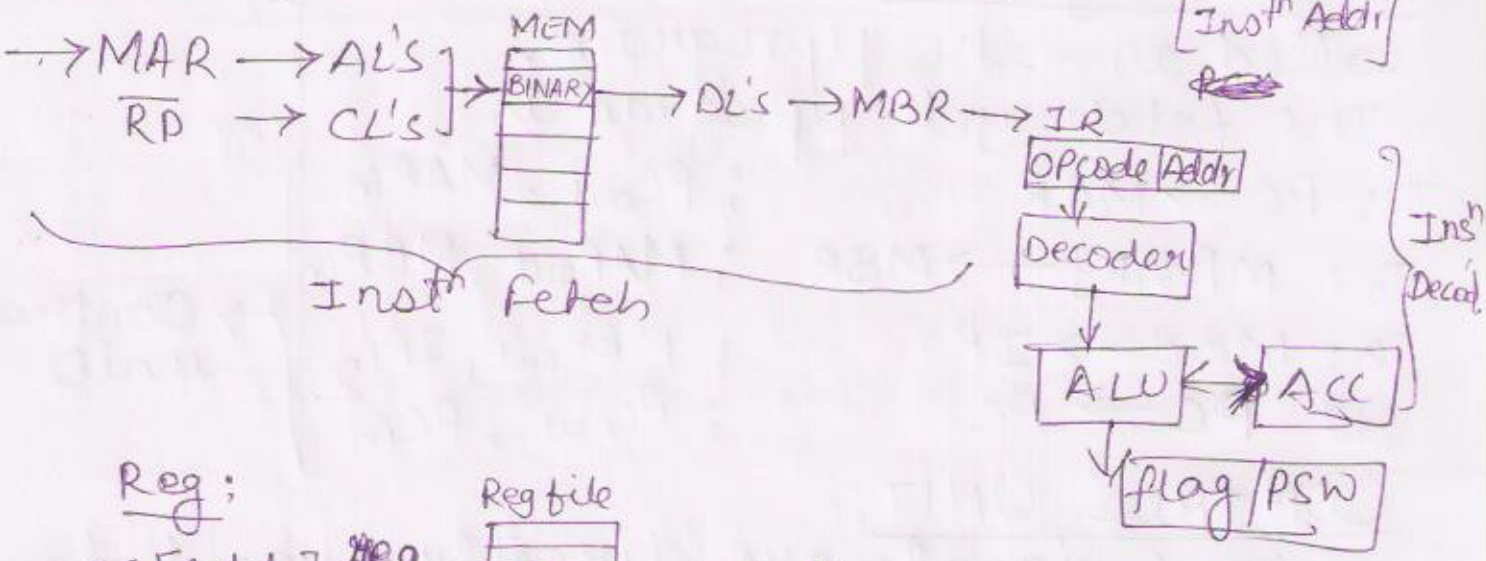
Mandatory Register:

1. PC: It is used to hold the starting address & immediately points to the next instr addrs.
2. IR:- It is used to hold the currently fetched instr to be decoded because instr format is predefined in this register, so it is not user accessible register.
3. ACC:- It is user accessible register, used to hold the ALU input & output.
4. TR:- It is used to hold the 2nd ALU operand. It is not a user accessible register.
5. MAR:- It is used to carry memory addrs. MAR register is directly connected to address lines of system bus.
6. MBR/MDR:- It is used to carry the binary sequence. MDR register is directly connected to data lines of system bus.

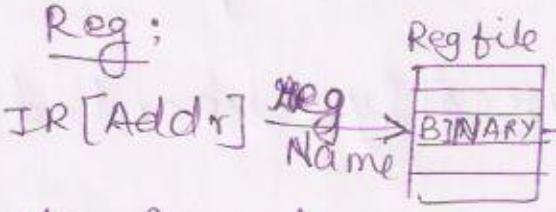
NOTE:- Apart from the above registers, processor supports set of the general purpose registers.

INSTRUCTION EXECUTION SEQUENCE USING MICROOPERATIONS:-

Executable command ⇒ -t = startingAddr → PC



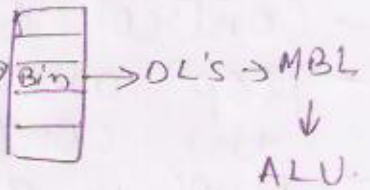
Reg:



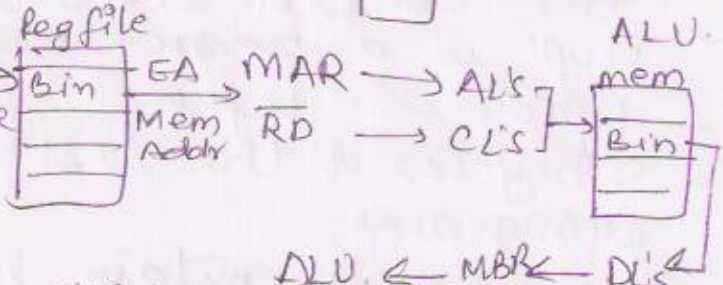
Mem Based:

Immed:- IR[Addr] $\xrightarrow{data}$ ALU

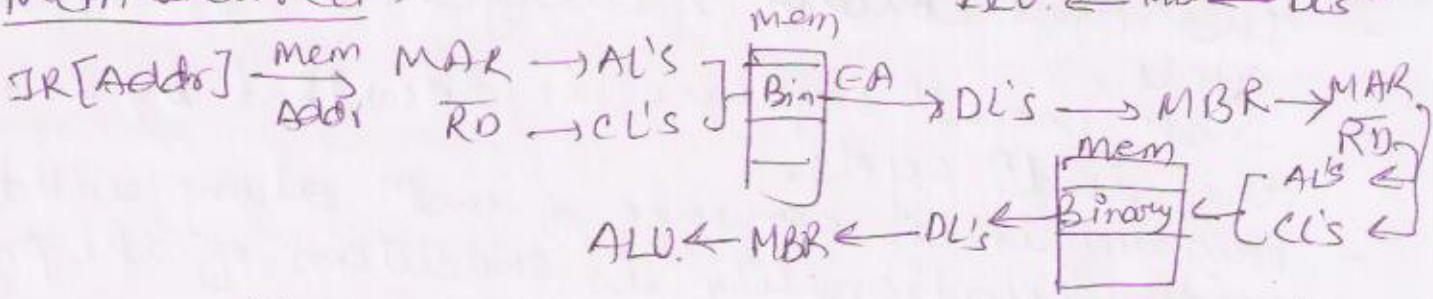
Direct:- IR[Addr] $\xrightarrow{Mem\ Addr}$ MAR → AL's, RD → CL's



Reg Indirect: IR[Addr] $\xrightarrow{reg\ Name}$



Mem Indirect:-



———×———

Microoperation is a basic reg. to reg. transfer opⁿ. Each μinstⁿ can be executed in 1 cycle. μOpⁿ is also called as μinstⁿ or control word or Atomic opⁿ.

Control signals are req. to executed the μinstⁿ.

Eg. (a) T_1 : $R_1 \rightarrow R_2$; R_{1out}, R_{2in}.

(b) R_{1out}, R_{2in}, R_{3in} ⇒ $R_1 \rightarrow R_2$, $R_1 \rightarrow R_3$ } In same cycle.

MicroProgram : The sequence of μop's is called as the μprogram.

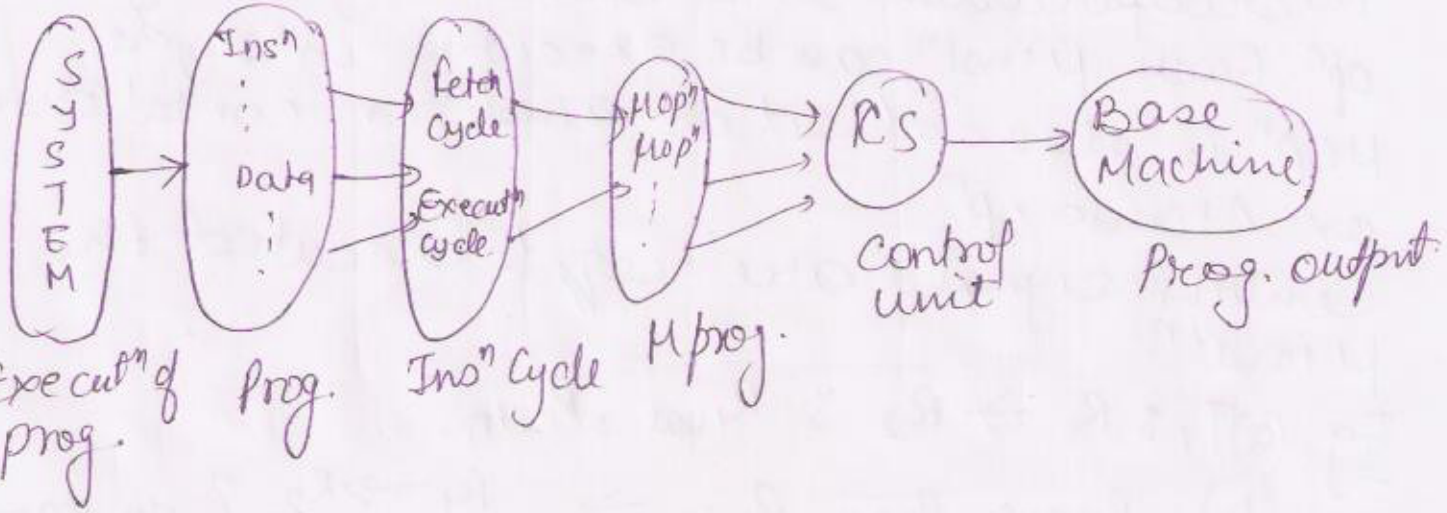
- The fetch cycle μprogram is

T_1: PC → MAR ; PCout, MARin
T_2: M[MAR] → MBR ; MARout, MBRin
T3: MBR → IR ; MBRout, IRin
PC → PC ; PCout, PCin

→ Control unit

CONTROL UNIT :-

- Control signals are directly executed on base machine.
- Control unit generates the control signals
- Control signals are implemented in CU.
- μop's are executed based on the sequenced the control signals.
- μop is a basic reg to reg transfer op.
- μprog is a sequence of μop's.
- Subcycles of the inst cycle invokes their own μprograms.
- Inst cycle contain 2 subcycles i.e fetch & execute cycle.
- Inst execut sequence is described by using the inst cycle.
- Program is a sequence of inst along with data
- System functionality is execution of the program



Control unit

To design the control unit the designer must identify following 3 parameters:

a) The no. of op^n (Inst^n) possible in the processor.
b) The no. of μop^n required to execute each inst^n.
(c) The no. of control signals required to execute each μop^n.

After identifying the above 3 parameters designer maintains the database.

Eg: # inst^n $\{I_0, I_2, I_3\}$ $\{$ADD, SOB, MUL$\}$

opcode = 2bits
00 - ADD
01 - SOB
10 - MUL
11 - Undef.

# of control signals recognized by h/w = $\{S_0, S_1, S_2\}$
μop^n req to execute each inst^n: $\{T1, T2, T3\}$

Database:

Ins^n/μop^n:	00 I_1 ADD	01 I_2 SOB	10 I_3 MUL
T_1	S_0, S_1	S_0, S_2	S_2, S_1
T_2	S_1, S_0	S_2, S_1	S_0, S_2
T_3	S_1, S_2	S_1, S_0	S_1, S_0

$\Rightarrow$ implemented in CU

Inst^n sel: ADD, SOB, MUL $\rightarrow$ ADD $\rightarrow$ 00 $\rightarrow$ T_1, T_2, T_3 $\rightarrow$ T1: S_0S_1, T2: S_2S_0, T3: S_1S_2 $\rightarrow$ Execution of T1: S_0S_1, T2: S_2S_0, T3: S_1S_2

Match to previous circle diagram

- After defining database, designer uses any one of the following approaches to imp. the database
(a) Hardwired Approach
(b) Microprogrammed Approach

Hardwired control unit design :-

- In this design, control signals are expressed as sum of product expression; they are directly implemented on the independent h/w.
- It is the fastest control unit design when control signals are implemented on independent hardware.
- It is used in real time appn. Eg:- aircraft simulation & weatherforecasting, etc.
- Even a minor modification requires redesign and reconnection of the control signals. ∴ It is not flexible.
- It is not suitable in the designing and testing places.
- Risc control unit is hardwired CU.

Q. Consider a hypothetical control unit. It supports 3 instn (I_1, I_2, I_3) and 4 control signals (S_0, S_1, S_2, S_3). Each instn requires 4 μopns (T_1, T_2, T_3, T_4) to complete the execution. The following table indicates what are the control signals required for each μopn, for each instn.

Instn/μopn	I_1	I_2	I_3
T_1	S_0S_1	S_2S_0	S_3S_1
T_2	S_1S_2	S_3S_2	S_0S_2
T_3	S_3S_2	S_1S_0	S_3S_0
T_4	S_0S_3	S_2S_3	S_2S_0

Get the control exp. for S_0, S_1, S_2 & S_3.

$S_0 = T_0(I_1 + I_2) + T_2 I_3 + T_3(I_2 + I_3) + T_4(I_1 + I_3)$

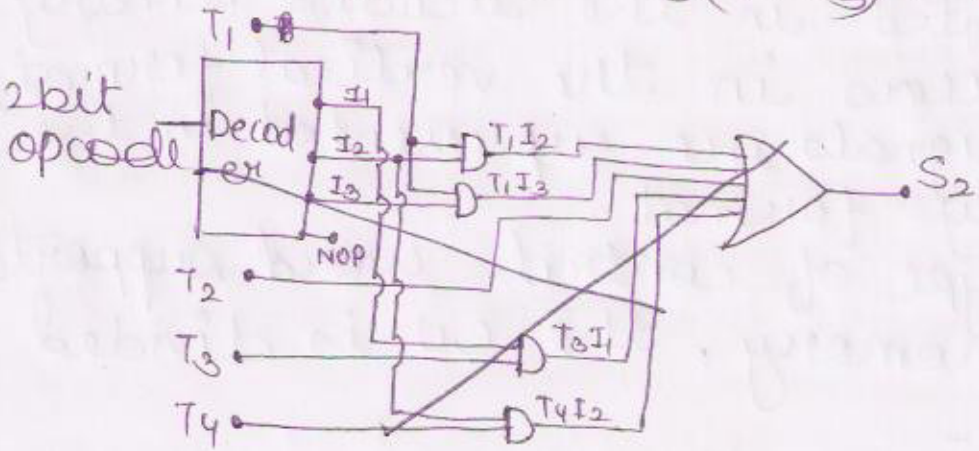
$S_1 = T_1 I_1 + T_2 I_1 + T_3 I_2$

$S_2 = T_1(I_2 + I_3) + T_2(I_1 + I_2 + I_3) + T_3 I_1 + T_4 I_2$

$S_3 = T_1 I_3 + T_2 I_2 + T_3(I_1 + I_3) + T_4(I_1 + I_2 + I_3)$

$\rightarrow T_1(I_2 + I_3) + T_2 + T_3 I_1 + T_4 I_2$

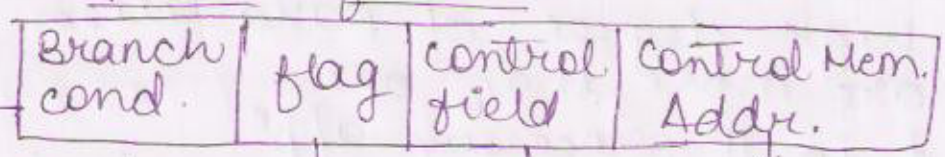
$\rightarrow T_1 I_3 + T_2 I_2 + T_3(I_1 + I_3) + T_4$



Microprogram Control Unit Design:-

- In this design, control memory is used to store the Mprog.. The control memory is permanent memory ie. ROM.
- MProg is a sequence of μoperations.
- μinstr are stored in the control memory by using the following format.

μinstn format:-



- One of system supported branch cond's → $\log_2$(# Branch cond'n)
- One of System supported flags → $\log_2$(no. of flags)
- control signals
  - Decoded → 1 bit/cs → Horizontal
  - Encoded → 'N' cs → $\log_2 N$ bits → Vertical
- Next μInstn Addr. → $\log_2$(Control mem size)

If control format is Horizontal → Horizontal μprog.

Based on way of representing control signals, μinstⁿ is divided into 2 types. i.e. (i) Horizontal μinstⁿ (ii) Vertical μinstⁿ. In Horizontal format, the control signals are represented in the decoded binary format, whereas in the vertical format, the control signals are represented in the encoded binary format.

- Based on type of control word supported by control memory, the CU is divided into 2 types:-

(i) Horizontal μprogram CU
(ii) Vertical μprogram CU.

Horizontal Programming:-

1. In this design, the CS are expressed in decoded binary format i.e. 1 bit/CS.
2. It supports longer control word
3. NO need of the external decoder to generate the CS, so it is faster than vertical.
4. It allows high degree of parallelism. i.e. none are more than one.
5. It is used in IIˢᵗ processing appⁿ.
6. It is bit flexible compared with hardwired approach.

Vertical Programming:-

1. In this design, the CS are expressed in encoded binary format, i.e. N CS require $\log_2 N$ bits.
2. It supports shorter control word.
3. There is a need of external decoder to generate the control signals. So it is slower

than horizontal.

4. It allows low degree of parallelism i.e. none or one.
5. It allows easy implementation of new instr. So it is more flexible.

NOTE:-

The ascending order of the CU design in terms of speed is Vertical - Horizontal - Hardwired

low ⟶ high

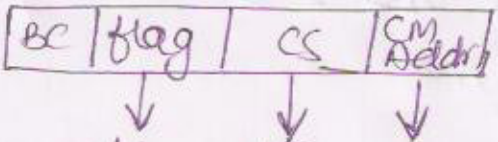
The ascending order of CU design in terms of flexibility

low ⟶ high

Hardwire - Horizontal - Vertical.

Q Consider a hypothetical CU. It supports 1024 control word memory. It has 48 CSs and 16 flags. What is the size of CU word in bits & control memory in bytes using the horizontal programming?

Ans:-



= 62 bits

CW size = 62 bits

= 1024 CW

= 1024 × 62 bits

$= \left(\frac{1024 \times 62}{8}\right)$ bytes

Q A hypothetical CU supports 8 mutually exclusive groups of CS that is tabulated below:-

Groups	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8
CS	1	3	7	6	12	14	20	2

How many #bits saved using the vertical programmed over horizontal programming?

Ans:- H:= 1+3+7+6+12+14+20+2 = 65 bits

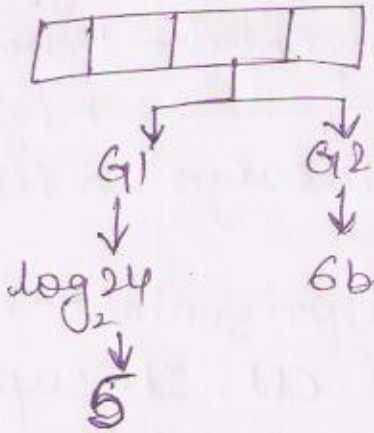
V = $\log_2 1 + \log_2 3 + \log_2 7 + \log_2 6 + \log_2 12 + \log_2 14 + \log_2 20 + \log_2 2$

= 23 bits

∴ No. of bits saved = (65 − 23) = 42 bits

Q. A field of μinstr supports 2 groups of CS. Group 1 indicates any 1 of 24 CSs and grp 2 indicates at most 6 from remaining. What is the size of control field?

⇒



⇒ 11 bits.

Performance Evaluation of Processing:-

Performance is an indirect measured parameter. It is depending on 2 directly measured parameters

a) Execution time

(b) Throughput

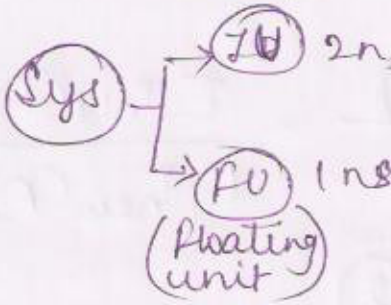
- The time required to complete the program is called execution time or elapsed time or response time.
- Consider 2 systems x and y. Both are used to execute the same task in which X system takes 5 ns & Y system takes 10 ns time to complete task. $P \propto \frac{1}{ET}$
- The no. of tasks completed in a unit of time is called as throughput. $P \propto T$
- The performance gain can be measured by using the speedup factor. Speedup factor compares the 2 different entities.

i.e. $\text{Speedup} = \frac{\text{Performance}_x}{\text{Performance}_y}$

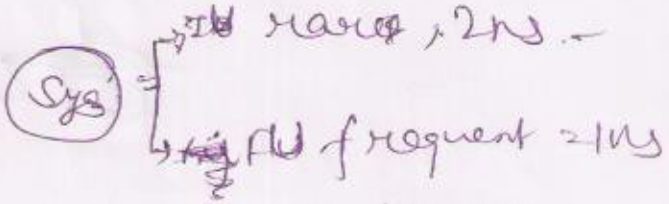
$S = \frac{ET_y}{ET_x}$; $S = 'n'$

System 'X' runs 'n' times faster than system 'y'

- Quantitative principles are used to improve the performance.
- These principles states that improve the performance of processor within the cost & price limits while making the common case fast.
- This principle is called as "Amdhals law"



$ET = 2 \times 10 + 1 \times 90 = 100 ns.$



Amdhal's law is focussed on performance gain after enhance the frequent components.

i.e.

$$\text{Soverall} = \frac{\text{Performance of the Sys with enhancement}}{\text{Performance of the sys without enhancement.}}$$

$$\text{Soverall} = \frac{1/ET_{new}}{1/ET_{old}} = \frac{ET_{old}}{ET_{new}} \quad (1)$$

- After enhance the system, it contains 2 portions i.e. enhanced portion and unenhanced portion.

$$\Rightarrow ET_{new} = ET_{\text{unenhanced portion}} + ET_{\text{enhanced portion}} \quad (3)$$

2 parameters are needed to calculate the ET_{new}

(i) $\text{Fraction}_{enhanced}$ (f).

(ii) $\text{Speedup}_{enhanced}$ (S).

Fraction enhanced indicates how much portion of the system is enhanced.

f = enhanced

(1-f) = unenhanced

⇒ Speedup enhanced indicates, how many times the enhanced portion is running faster that the old portion.

$$S = \frac{\text{Performance of enhanced portion}}{\text{Performance of old portion}}.$$

$$= \frac{1/ET_{new}(F)}{1/ET_{old}(F)} = \frac{ET_{old}(F)}{ET_{new}(F)}$$

$$ET_{new}(F) = \frac{ET_{old}(F)}{S} \quad -(2)$$

②nd in ③rd

$$ET_{new} = ET_{old}(1-F) + \frac{ET_{old}(F)}{S}$$

Substitute ② in ①,

$$S_{overall} = \frac{ET_{old}}{ET_{old}(1-F) + \frac{ET_{old}(F)}{S}}$$

Let $ET_{old} = 1$.

$$S_{overall} = \frac{1}{(1-F) + \frac{F}{S}} \quad \left[(1-F) + \left(\frac{F}{S}\right)\right]^{-1}$$

If $ET_{old} = 1$, then ET_{new} is always less than 1 & $S_{overall}$ is always greater than 1.

NOTE:- When the system contains multiple enhancements then $S_{overall} = \left[(1-\Sigma F_i) + \sum \frac{F_i}{S_i}\right]^{-1}$

i = # enhancements.

Q. Consider a computer which is used in the mathematical model simulations It consists of 2 components :- Integer & floating point units. FPU is enhanced then it runs 2 times faster but only 10% of the instr are floating point. What is the performance gain.

Ans:- $S_{overall} = \left((1-f) + \frac{f}{s}\right)^{-1} = \left[(1-0.1) + \frac{0.1}{2}\right]^{-1}$

$= 1.05$

≈ 1

According to the problem statement, instead of frequent case the rare case goes on enhancement. ∴ there is no considerable performance gain.

The corrective measurement is enhance the frequent case i.e. Integer unit.

$S_{overall} = 1.82 \simeq 2$ ∴ considerable gain

Q. Suppose the cache memory is 10 times faster than main memory & it is used 70% of the time. How much speedup do we gain by using this cache.

$$\left[(1-0.7) + \frac{0.7}{10}\right]^{-1} = \left(0.3 + \frac{0.7}{10}\right)^{-1}$$

$$= \left(\frac{3.7}{10}\right)^{-1} = \frac{10}{3.7} = 2.7.$$

⇒ Introducing cache improves performance by 2.7 times.

Q 3 enhancements are proposed for a new architecture with the following speedups ⇒ $S_1 = 30$, $S_2 = 20$, $S_3 = 15$.

If the enhancement 1 & 2 both will

be used 25% of the time, what fraction of the time enhancement 3 will be used to gain the overall of 10

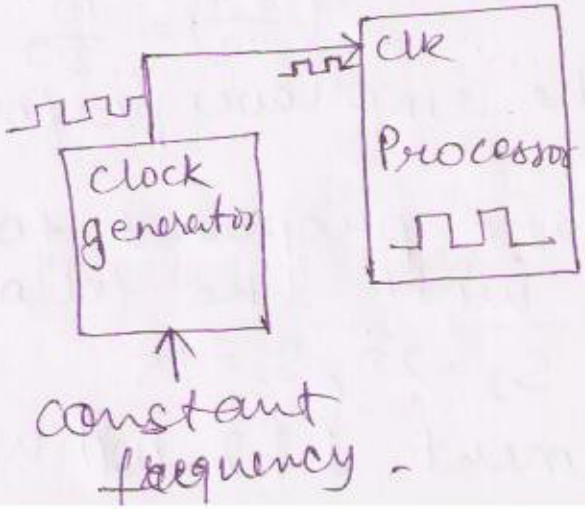
$$10 = \left[[1-(0.5+x)] + \frac{0.25}{30} + \frac{0.25}{20} + \frac{x}{15}\right]^{-1}$$

$$10 = \left[0.5 + x + \frac{0.5+0.75}{60} + \frac{x}{15}\right]^{-1}$$

$$10 = \left(0.5 + x + \frac{0.25}{60} + \frac{x}{15}\right)^{-1}$$

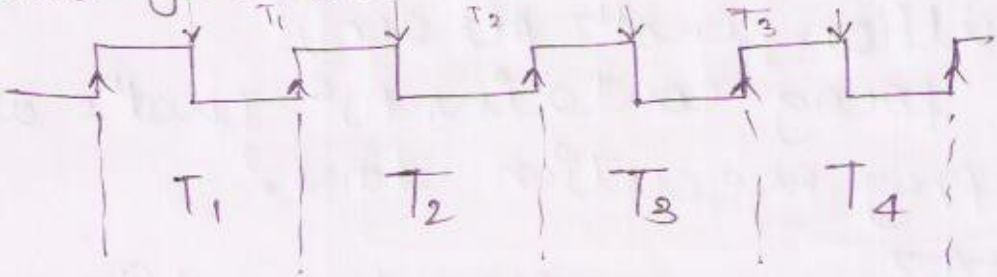
Calculation of CPU time :-

- CPU time means program execution time.
- Processor is having clock pin as an input pin.
- The clock pin is externally connected with the clock generator.
- Clock generator is operating with the constant frequency to generate the clock pulses.
- These clock pulses are carried into the CPU through the clk pins. ∴ The CPU operations are controlled by the clock.



The program execution time can be calculated based on cycle & cycle time.

- Cycle is defined as raising edge to raising edge Transition or falling edge or falling edge Transition.
- It is also called as the 'tick'/clk tick/cycle/ clock cycle etc.



- The time required to transfer the pulse either from raising edge to raising edge or from falling edge to falling edge is called as cycle time.
- The cycle time is depending on the clock frequency.

$1\,GHz = frequency$

$\Rightarrow t = \frac{1}{1GHz} = 10^{-9}\,sec. = 1ns$

The program execution time = # sec per prog

= # inst^n/prog * # cycles/ins^n * # seconds/cycle.

~~= # inst^n/prog * # cycles/inst~~

$$ET = Inst^n\,Count * CPI * cycletime$$

CPI: cycles/inst

Program is a mixing of ~~3~~ 3 categories of inst^n. So different inst^n take different # cycles. ∴ The Prog. execut^n time / CPU time

$$= \sum (IC_i * CPI_i) * Cycletime$$

i = type of inst^n.

Q. Consider 1ns clock cycle processor. It consumes 4 cycles for load & store opⁿ, 6 cycles for ALU oprs and 5 cycles for branch opⁿ.

The relative frequencies of these instn are 40%, 40% & 20% resp.

(a) What is the average instn executing time?

(b) What is the performance in terms of MIPS. (Million instns per sec)?

(c) If the prog contains 10^6 Instns what is the prog execution time?

Ans:-

(a) Avg instn execution time $= \sum (IC_i * CPI_i) * t_{cycle}$

$= [(0.4*4) + (0.4*6) + (0.2*5)] * 1\,ns$

$= 5\,ns.$

(b) 1 instn $\rightarrow$ 5 ns.

? $\rightarrow$ 1 sec.

$\frac{1}{5 \times 10^{-9}} = 2 \times 10^8$ Ins/sec $= 200$ MIPS.

(c) $10^6 \rightarrow 5 \times 10^{-9} \times 10^6$

$= 5 \times 10^{-3}$

$= 5$ ms.

(5D) HIGH PERFORMANCE ARCHITECTURE

High performance architecture exhibits the concurrency.

→ Concurrency means more than one event execution.

→ Event may be program, subprogram, instn or stage (Intrainstn)

Concurrency implies parallelism, simultaneous, pipelining.

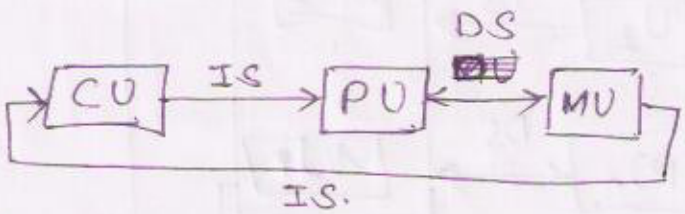
→ Parallelism means 2 or more event execution

at the same time period.
→ Simultaneous means 2/more event execution at the same time instance.
→ Pipelining means 2 or more event executed in overlapping span.
According to the Flynn's ~~archi~~ classification, the parallel architecture is divided into 4 cata:-

(i) SISD → Single inst^r single data stream
(ii) SIMD → Single " multiple "
(iii) MISD → Multiple " single "
(iv) MIMD → ~~Mul~~ " " multiple "

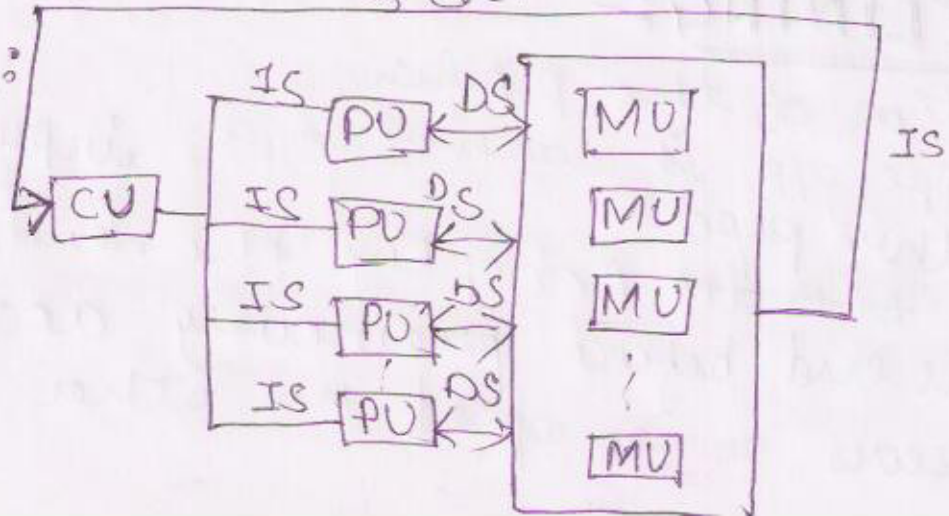
PU: Processing Unit
CU: Control Unit
MU: Memory Unit
IS: Inst^r stream
DS: Data stream

SISD:



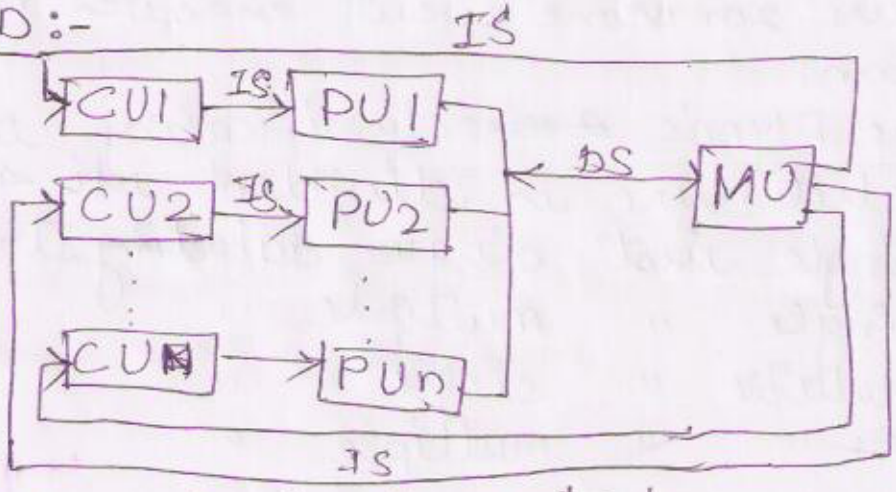
Inst^n & data both are stored in same memory. After performing the operation, the result is also stored in same memory. This concept is known as stored program structure.
→ Von Neumann Archi. was designed based on stored program concept.
→ Uniprocessor is using the SISD architecture.
Eg → VAX, IBM 360, ~~etc~~.

SIMD:



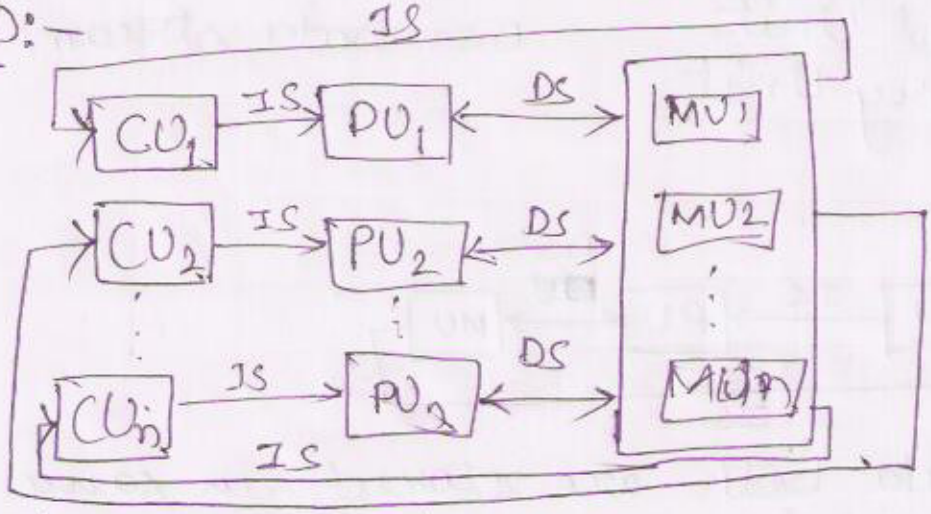
→ Array processors use the SIMD archi.
eg → STERAN, PEPE, MPP etc.

MISD:-



Not yet implemented.

MIMD:-



~~Multiprogramming~~ Multiprocessor system uses MIMD architect
→ ex: Eg. Cray, Cyber etc.

→ To improve the performance of uniprocessor system pipelining is used.

PIPELINING:-

Function of the pipeline
→ I pipe o/p is connected as input to the another pipe.

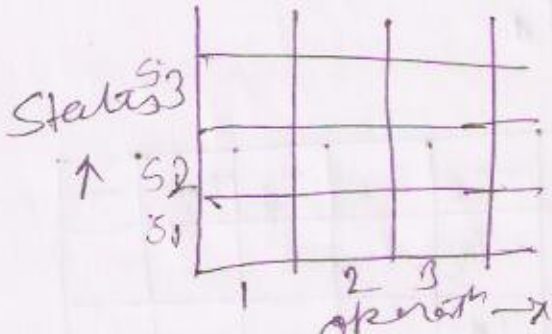
Def^n:- ~~Access~~ Accepting the new inputs at one end before previously accepted input appears as an op at the other end.

Defⁿ states that insert the new i/p before completion of old inputs. That means new ips are executed alongwith the old inputs. So pipeline allows the overlapping execution.

Non pipeline allows the non-overlapping execution i.e. new inputs are inserted only after completion of old inputs.

The overlapping execution sequence can be represented using space time diagram

- In this diagram x → cycles, y → stages.



The successful char. of pipelines. Is that in every new cycle, new input must be inserted into pipeline.

∴ cor=1.

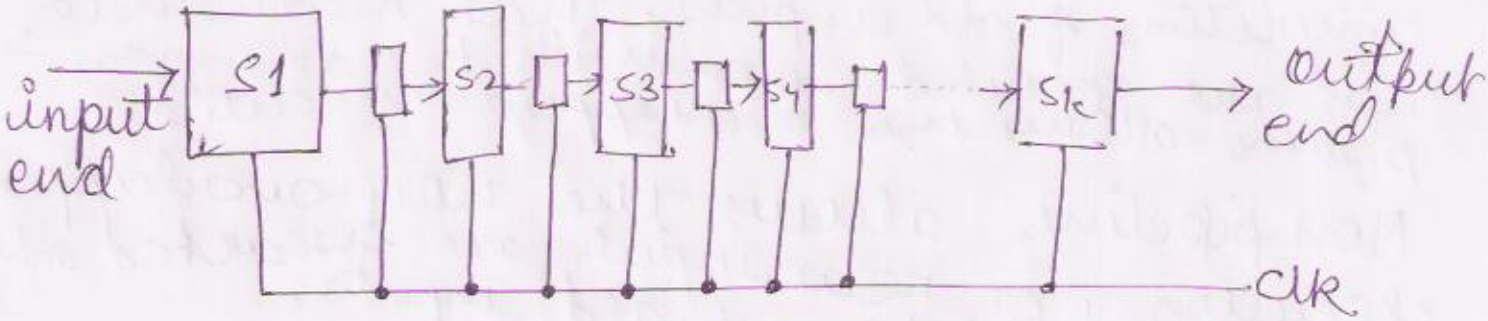
Designing of the pipeline:

1. Pipe has 2 ends. — I/p end
   — O/p end.

Between the inp & outputs ends, i.e input & output pipe, is interconnected.

→ Interface registers are used b/w the stages to hold the intermediate results, they are also known as buffer / latch, pipeline register.

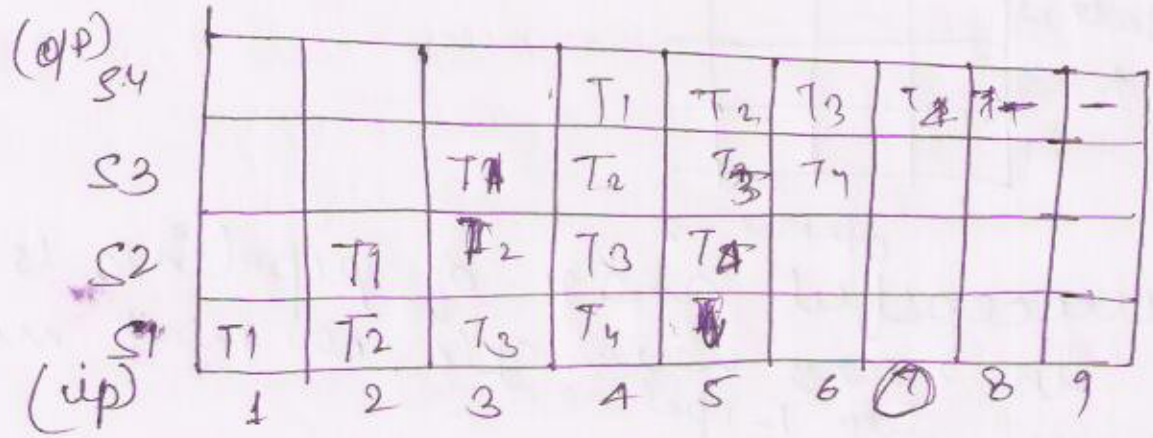
All the stages along with interface registers are controlled by the common clock.



Consider 4 segment pipeline used to process the 4 tasks.

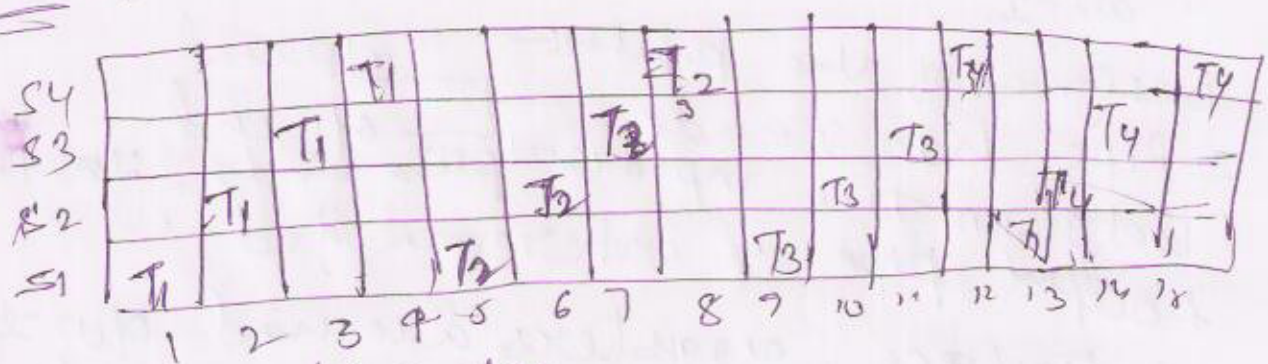
Stages: S1, S2, S3, S4

I/p : T1, T2, T3, T4



4 tasks = 7 cycles

Sequential:-



4 tasks = 16 cycles.

Performance evaluation of pipeline processor:-

→ Consider k segment pipeline used with the clock cycle time time of tp used to execute n-tasks.

→ The very first task is executed in the pipeline is non-overlapping order ...

It takes k cycles to complete the operation. The remaining n-1 tasks are emerged from the pipe at the rate of 1 cycle per task. ∴ n-1 tasks consumes n-1 cycles to complete the operation.

→ So the total time required to process the n tasks using the k segment pipeline is

$$ET_{pipe} = (k+n-1) \text{ cycles.}$$
$$= (k+n-1)\, t_p.$$

Consider non pipeline processor used to execute the n-tasks in which each task takes t_n time to complete the operation.

∴ The total time required to complete the n tasks in non pipeline is

$$ET_{non\text{-}pipe} = n\, t_n$$

The performance gain of the pipeline processor over the non pipeline is

$$\text{Speedup } (S) = \frac{\text{Performance}_{pipe}}{\text{Performance}_{non\text{-}pipe}}$$

$$S = \frac{1/ET_{pipe}}{1/ET_{nonpipe}} = \frac{ET_{nonpipe}}{ET_{pipe}}$$

$$= \frac{n\, t_n}{(k+n-1)\, t_p}$$

When the no. of tasks increases, that n becomes too large than k-n. then k+n-1 value is approached to n

When $n \gg k$. then

$$k+n-1 \simeq n.$$

Now $$S = \frac{n\, t_p}{(k+n-1)\, t_p} = \frac{n\, t_p}{n\, t_p}$$

$$\boxed{S = t_n / t_p}$$

When all the tasks take same no. of cycles then 1 task execution time is also equal to the no. of stages in the pipeline.

$t_n = k$ cycles.

$t_n = k \cdot t_p$

Under this condn, $S = \frac{k t_p}{t_p}$

$$\Rightarrow \boxed{S = k}$$

$k \rightarrow$ no. of stages in pipeline

Also called as pipeline depth.

When the processor is operating with 100% efficiency then maximum speedup depth the pipeline depth.

100% $\eta \rightarrow S_{max}$

?, $\eta \rightarrow S$

$$\boxed{\eta_{pipe} = \frac{S}{S_{max}} = \frac{S}{k}}$$

The through put of the pipeline is # task processed per total time taken to process the task.

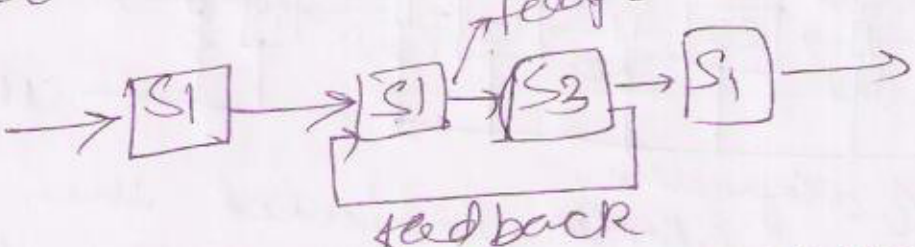
$$T_{Pipe} = \frac{\#\text{tasks processed}}{\text{total time taken to process the tasks.}}$$

$$= n / (k + n - 1) t_p.$$

Types of the pipeline:-

(1) Linear pipeline:- This pipeline performs only a specific functionality. It consists only the feed forward connections.

Non-linear pipeline: This pipeline can perform multiple functionality. It consists of feed forward & feed backward connections. (Asynchronous)



Reservation table is required to use the pipeline stages.

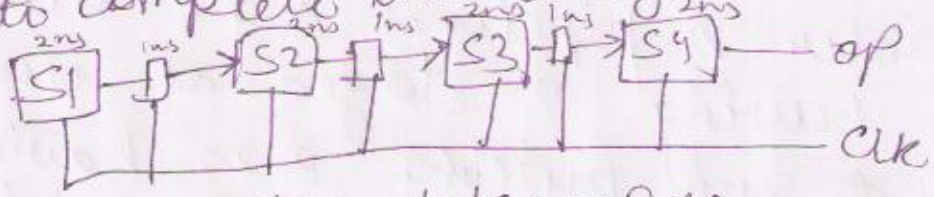
- Synchronous pipeline:-

In this pipeline operations are initiated based on the clock.

- Asynchronous pipeline:-

→ In this pipeline, operations are initiated based on the hand shaking signals.

Uniform delay pipeline:- In this pipeline all the stages maintains the same amt of delay to complete the assigned operation.



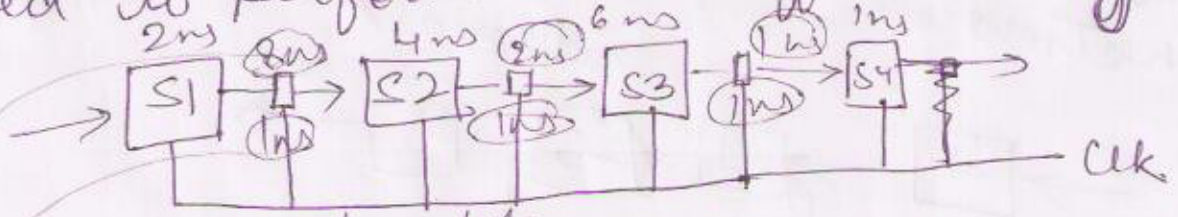
(i) t_p = stage delay = 2ns.

(ii) If buffer delay is included then
t_p = stage delay + buffer delay
= 3ns

(iii) Skew or setup time overhead is included. then
t_p = t_p + overhead time.

Non-uniform delay :-

In this pipeline, different stage delays are used to perform the different stage operations.



1) t_p = max of stage delay

2) t_p - if buffer delay is included then

t_p = max (stage delay) + Buffer delay.

3) In nonuniform buffer delay is included,

t_p = max (stage delay + buffer delay).

Q. Consider an inst^n pipeline which has the speedup factor 10 ~~vehicle~~ while operating with 80% efficiency. What would be # stages present in the pipeline?

$\Rightarrow$ $\eta = 80\%$, $S = 10$

$\eta = \frac{S}{K}$ $K = \frac{10}{0.8} = 12.5 \approx (13)$

Q Consider 2 pipelines A & B where pipeline a is having 8 stages of uniform delay of 2ns. pipeline B is having 5 stages with the respective stage delays of 2ns, 3ns, 4ns & 1ns, 2ns. How much time is saved when 100 tasks are pipelined using A instead of B.

$\Rightarrow$ Pipe A: $K = 8$

$n = 100$

$t_p = 2$ ns.

$ETA = (K + n - 1) t_p$

$= (8 + 100 - 1) 2 = 214$ ns.

Pipe b: (5+100-1)·tp = 104 × 4 = 416 ns.

Time saved = 416 - 214 = 202 ns.

Q Consider 4 segment pipeling with respective stage delays of 10, 20, 5, 15 ns. What is the approx speedup when very large no. of instr are pipelined?

⇒ $S = \frac{t_n}{t_p}$

$t_p = \max(10, 20, 5, 15)$

$t_p > 20$ ns

(Non overlapping execution time) $t_n = S_1 + S_2 + S_3 + S_4 = 10 + 20 + 5 + 15$

$= 50$ ns

$S = \frac{50}{20} = 2.5$

$\eta = \frac{S}{K} = \frac{2.5}{4} = 0.625$

Q: Consider 4 stage pipeline with the following time delays:

$t_1 = 20$ ns $\quad t_4 = 25$ ns.

$t_2 = 40$ ns

$t_3 = 15$ ns

The interface regis. used btw the stages have the delay of 5 ns, what is the speedup?

⇒ $S = \frac{t_n}{t_p} = \frac{100}{45}$

$= \underline{2.2}$

[No need of adding buffered delay because there is no need to save intermediate results in non overlapping execution.

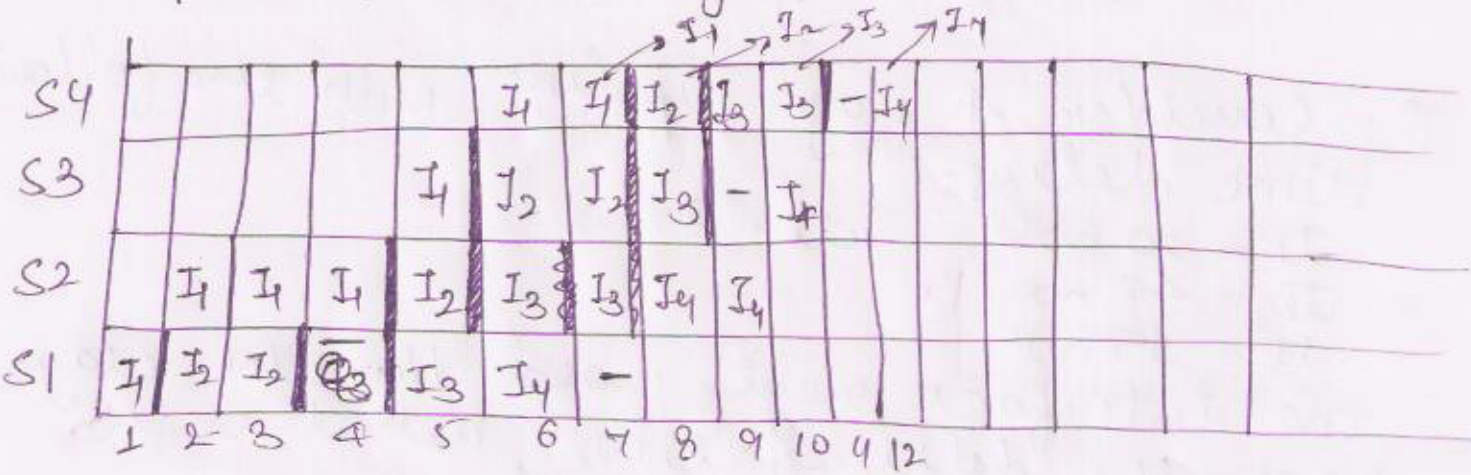
Q. Consider 4 stage instruction pipeline where different instructions are sending different amount of time at different stages that is shown below:-

	S_1	S_2	S_3	S_4	
I_1	1	3	1	2	$\Rightarrow 7$
I_2	2	1	2	1	$\Rightarrow 6$
I_3	1	2	1	2	$= 6$
I_4	1	2	2	1	$= 6$
					(25)

(i) How many cycles required to complete the above inst^n?

(ii) what is the speedup?

⇒ Not uniform delay, not non-uniform delay. Special case. Solved using space time diagram.



(i) ⇒ 12 cycles.

(ii) ⇒ $S = \frac{ET_{nonpipe}}{ET_{pipe}} = \frac{I_1 + I_2 + I_3 + I_4}{12}$

$= \frac{25}{12} = 2.08$

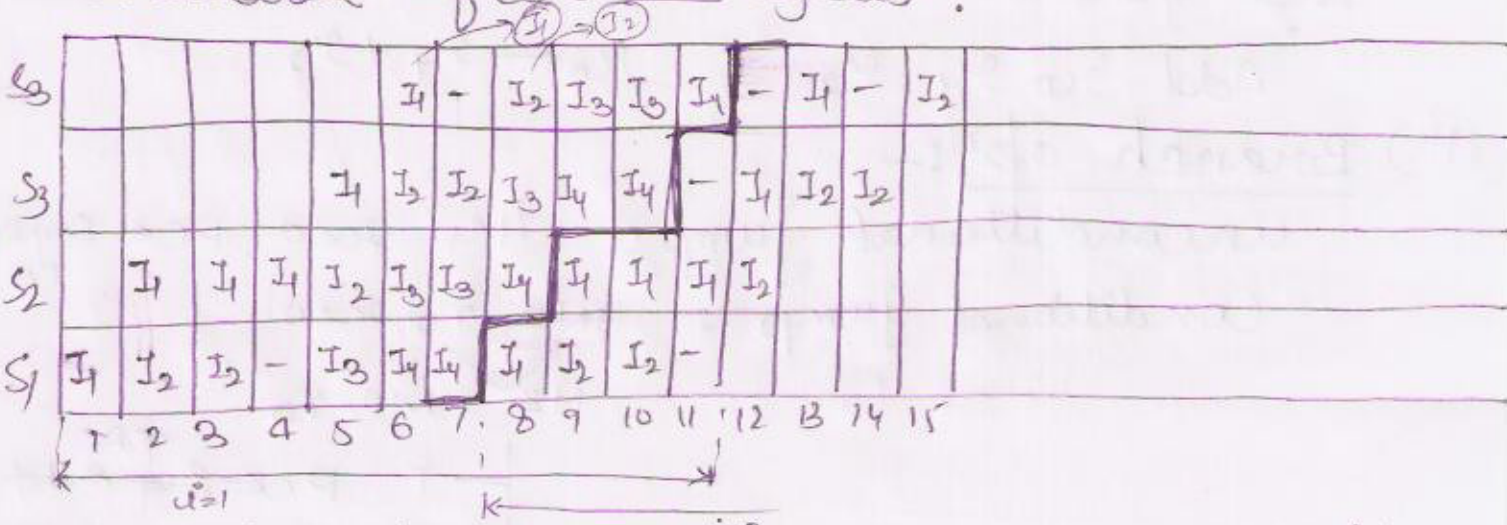
Q. Consider 4 stage instr pipeline where different instn are spending different amount of time at different stages that is shown below:-

	S1	S2	S3	S4
I1	1	3	1	1
I2	2	1	2	1
I3	1	2	1	2
I4	2	1	2	1

The following loop is executed on the pipeline

```
for (i=1; i<=1000; i++)
  { I1; I2; I3; I4; }
```

The output of the I_2 for $i=2$ will be available after ___ cycles.



(overlapping cycles) ← overlapping because it is pipelining (called loop level parallelism).

Ans ⇒ 15

RISC PIPELINE

To analyse the implementation issues of pipelining, let us consider a successful pipelining as reference.

RISC ϕn is a successful pipelining.

RISC processor supports 5 stage instr

pipeline. It is used to execute the inst^n. RISC processor supports 3 catagories of instructⁿ.

(i) Data transfer Instⁿ. LOAD & STORE instⁿ are used to transfer the data between CPU & memory, in which to access the memory Indexed addressing mode is used.

The instr format is

LOAD r_0, 3(r_4) ; $r_0 \leftarrow M[3+[r_4]]$

STORE 4(r_2), r_3 ; $M[4+[r_2]] \leftarrow r_3$

(ii) ALU operations: ALU operations are performed only on registers.

Instr format is

Add r_0 r_4 r_2 ; $r_0 \leftarrow r_4 + r_2$

(iii) Branch opⁿ:-

unconditional jump: JMP 2000 ; PC ← ~~Seq Adr~~ TA

conditional jump : JNZ r_0, 2000

NZ CMP r_0

T ; PC ← ~~Seq Add~~ TA

F ; PC ← seq Addr.

To execute the above instⁿ, RISC processor supports 5 stage instⁿ pipeline.

STAGE 1 (Instⁿ fetch):- In this stage instⁿ are fetched from memory based on the program counter. At the end of this stage, the program counter value is incremented to fixed constant

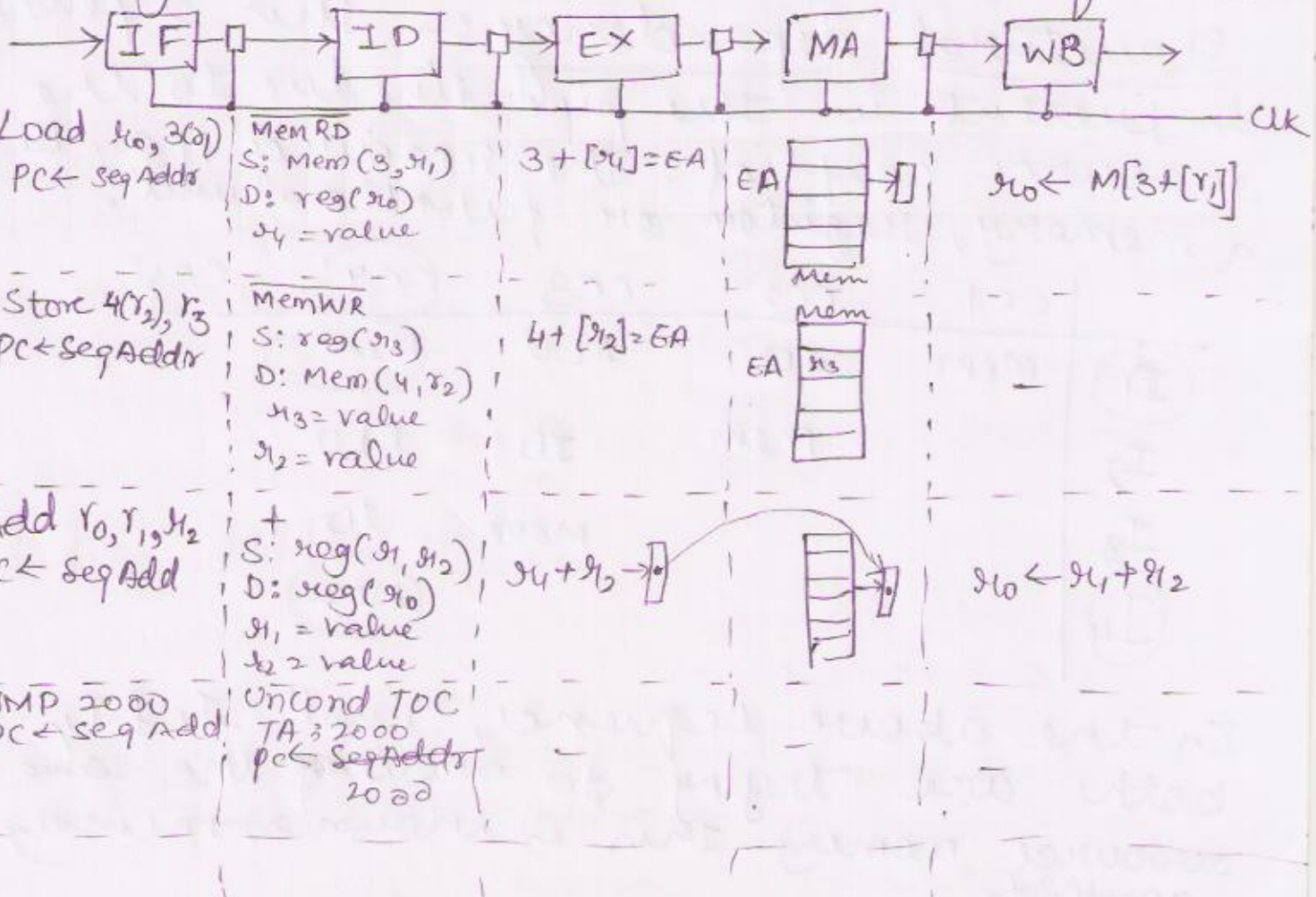
STAGE 2 (INST DECODE) :- In this stage 2 operations are performed i.e decode instⁿ & accessing the register file. This stage also contains the comparator logic to evaluate the branch conditions.

STAGE 3 (EXECUTION): In this stage ALU operations are performed

STAGE 4 (Memory Access): In this, memory read & memory write operations are performed

STAGE 5 (Write Back): This stage opⁿ is divided into 2 halves.

In the first half, register write operation is performed, in the 2nd half, register register read operation is performed.



JNZ R0, 2000 PC ← Reg Addr	cond TOC TA : 2000 cond: NZ, R0 R0 = value NZCMP R0 T / F PC ← seq 2000 / PC ← seq.	–	–	–

Branch insth ⇒ 2nd stage
ALU + reg. load ⇒ 5th stage
Reg ↔ reg store ⇒ 4th stage

Depend

There are 3 types of dependencies possible in pipeline.

(i) structural dependency.
(ii) data dependency
(iii) control dependency.

Dependency creates the extra cycles in the pipeline. Extra cycle without operation (New i/p initiation) called as stall.

Structural dependency :- This dependency is present in the pipeline due to the resource conflict. The resource may be a memory, register or functional unit.

	CC1	CC2	CC3	CC4	CC5
I1	Mem	ID	ALU	Mem	
I2		Mem	ID	ALU	
I3			Mem	ID	
I4				Mem	

In the above sequence, insth I_1 & I_4 both are trying to access the same resource memory. This is known as memory conflicts

Conflict is an unsuccessful operation. So to make it successful, keep instr 4 into the wait until the resource becomes available. i.e. this waiting creates the stalls in the pipeline.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
I_1	mem	ID	ALU	mem	WB			
I_2		mem	ID	ALU	mem	WB		
I_3			mem	ID	ALU	mem	WB	
I_4								mem

stalls

To minimise the no. of stalls in pipeline due to the structural dependency, renaming concept is used.

Renaming states that use the independend memory modules to store the instr & data separately called as code memory & data memory.

Use the code memory in stage 1 & data memory is stage 4. ∴ without conflict, instr 1 accessing the data from data memory and instr 4 is fetched from code memory.

	CC1	CC2	CC3	CC4	CC5	CC6
I_1	CM	ID	ALU	DM	WB	
I_2	—	CM	ID	ALU	DM	WB
I_3	—	—	CM	ID	ALU	DM
I_4	—	—	—	CM	ID	ALU
I_5	—	—	—	—	CM	ID
I_6						CM

Q1 ~~L=04~~

1. H=02
2. L=05
3. L=04 (NZ) L=03, 02, 01, 00
   (NX) 2 3 4
   ↑ false
   ↓
   H=01 00
   ↓ false
   (1)

Ans ⇒ 4,1.

Q2. L=04, 3, 2, 1, 00
↓
H=0 → L=255 ···· 0
255 True.

∴ 255+4=(259 ≠ 1)

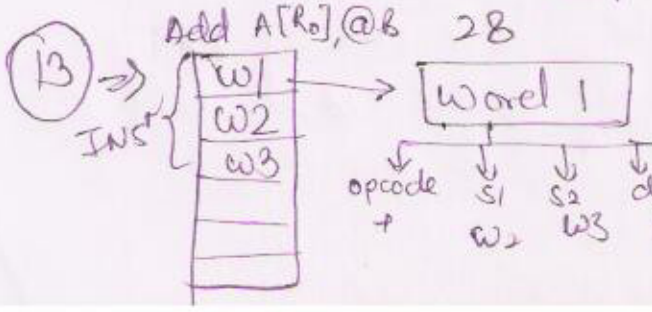
Q3: TOCAM = base + relative.

Q4
```
MOV B,#0     B←0
MOV C,#8     C←8
Z: CMP C,#0  8cmp0(NZ)
   JZ X      (false)
   SUB C,#1  C←7
   RRC A,#1  CA7A6A5A4A3A2A1, A0   after 8 ⇒ A0 A7A6A5A4A3A2A1C A7
   JC Y;
   JMP Z
Y: Add B,#1
   JMP 2;
X:
```
0 cmp0(NZ) false C←0

← To see original A, 1 more RRCA,#1

Q6 → TOC → relative + Base.

Q7. →
I1: 1000 - 7
I2: 8 - 11
I3: 12 - 15
I4: 16 - 23
I5: 24 - 27
Add A[R0],@B 28



Q9 →

F+D	OF+PD
2w×2c	3
1×2	3
1×2	1
2×2	3
1×2	–
14	10

= 24 cycles.

A[R0] ← A[R0] + @B

M[A+[R0]] ← M[A+[R0]] + M[[B]]

1 reg ref - Index
1 arithm EA
1 mem ref → write

1 reg. ref = Index
1 arith - EA
1 mem ref → read

1 M ref ⇒ EA
1 M ref → read

1 M ref → A
1 M ref → B
} 6 M ref.

(20) opcode | | |

128 → 7 bits
256M → 28 bits

5 bits → opcode
→ 4

→ rm
→ 1

~~7+28~~ 35
(5)

(23) $R_s \leftarrow 1$

$[R_d \leftarrow 1000 + R_s = 1001 = (1)$
$R_d \leftarrow 1]$

$R_d \leftarrow 1001$

$[0 + 1001] \leftarrow 20$

20
1001

(31) Before replacing the register, take the existing value and compare the remaining program instruction to check whether it is used as i/p or not.

- If it is not used at input, replace register content otherwise verifies the o/p as old i/p or new i/p.
- Replace register when it is a new i/p and donot replace register when it is old output.

(31) a = 1
b = 10
c = 20
d = a + b = 11
e = c + d = 31
f = c + e
b = c + e

	reg1	reg2	reg3
	a	b	c
	d	f ~~b~~	~~c~~ b
	e		
	d		

Chap 2 Q 20

CISC

1
1
40
MC

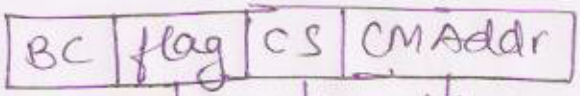
1
1
1
6 times
1
1
MC

RISC → 1CPI cycles per inst^n.

$$S = \frac{ET_{CISC}}{ET_{RISC}} = \frac{42}{15} = 2.8$$

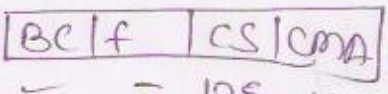
12. #inst^n = 256
# μop^n = 8
Total = 2048 (256×8)
= (M)
CM size = 2048 μop^n

HCW



4 48 11 = 63 bits

11 →



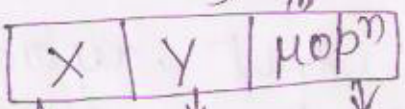
– – 125 10 ⇒ 135.

10 → PC_{out}, R_{in}, MAR_{in}
MAR_{out}, MBR_{in}
MBR_{out}, PC_{in}

change $PC \leftarrow M[R_0]$

T_1: PC_{out}, R_{in}
T2: R_{out}, MAR_{in}
T3: MAR_{out}, MBR_{in}
T4: MBR_{out}, PC_{in}

8 →



⇒ 26 bits

Nxt μop^n Addr
8

mux select
3 bit

B

⇒ X + 3 + 13 = 26 ⇒ X = 10

Y = 3

CM = 2^{10} cells

= 1024 CW
= 1024 × 26 Bits
= $\frac{1024 \times 26}{8}$ bits

To implement the BBS inst^n by using the alternative stmt, make sure that mask must contain only 1 bit as 1 that itself based on the position & reset bits must be zero. During AND op^n, corresponding bit value is maintained intermediate

(6D) Data Dependency :-

- Consider the program segment where the instruction j follows instⁿ i in the program order i.e.

  i : instⁿ
  j : instⁿ
  ⋮

- Data dependency exists when Instⁿ j tries to read data before instⁿ i writes it.
- Data dependency existing between adjacent instⁿ is called true data dependency.

Eg: I_1 : Add r_0, r_1, r_2
I_2 : Mul r_3, r_0, r_4

- When the above instⁿ are executed in non overlapping order, there is no dataloss problem because I_2 is inserted after completion of I_1.
- When above instⁿ are executed in pipeline, instⁿ I_2 is inserted before completion of I_1. So I_2 tries to read r_0 before I_1 writes it.
  ∴ old data is injected into the process.
- To make it correct, keep I_2 in waiting until completion of I_1
- This waiting creates "stalls" in the ~~program~~ pipeline.
- To detect the data dependency condition, some kind of database will be maintained in the decoding stage.
- The dB contains following fields :-

(i) Serial No Func° unit Destⁿ Indep1 Indep S2 dep S1 (I) Dep S2 (I) Dep S1 (D) Dep S2 (D) Status

SNO	Fun° Unit	Destⁿ	Independent Source 1 (I)	Ind. S2 (I)	dependent Sources (D)	Dep S2 (D)	Status
I_1	ADD	r_0	r_1	r_2	–	–	0 ⇒ EX Stage
I_2	MUL	r_3	–	r_4	r_0 (I_1)	–	0 ⇒ NOEX Stage

No further fetch are performed because I_2 is not passed to EX stage & there's a stall. The above algo is called "TOMASULO Algo."

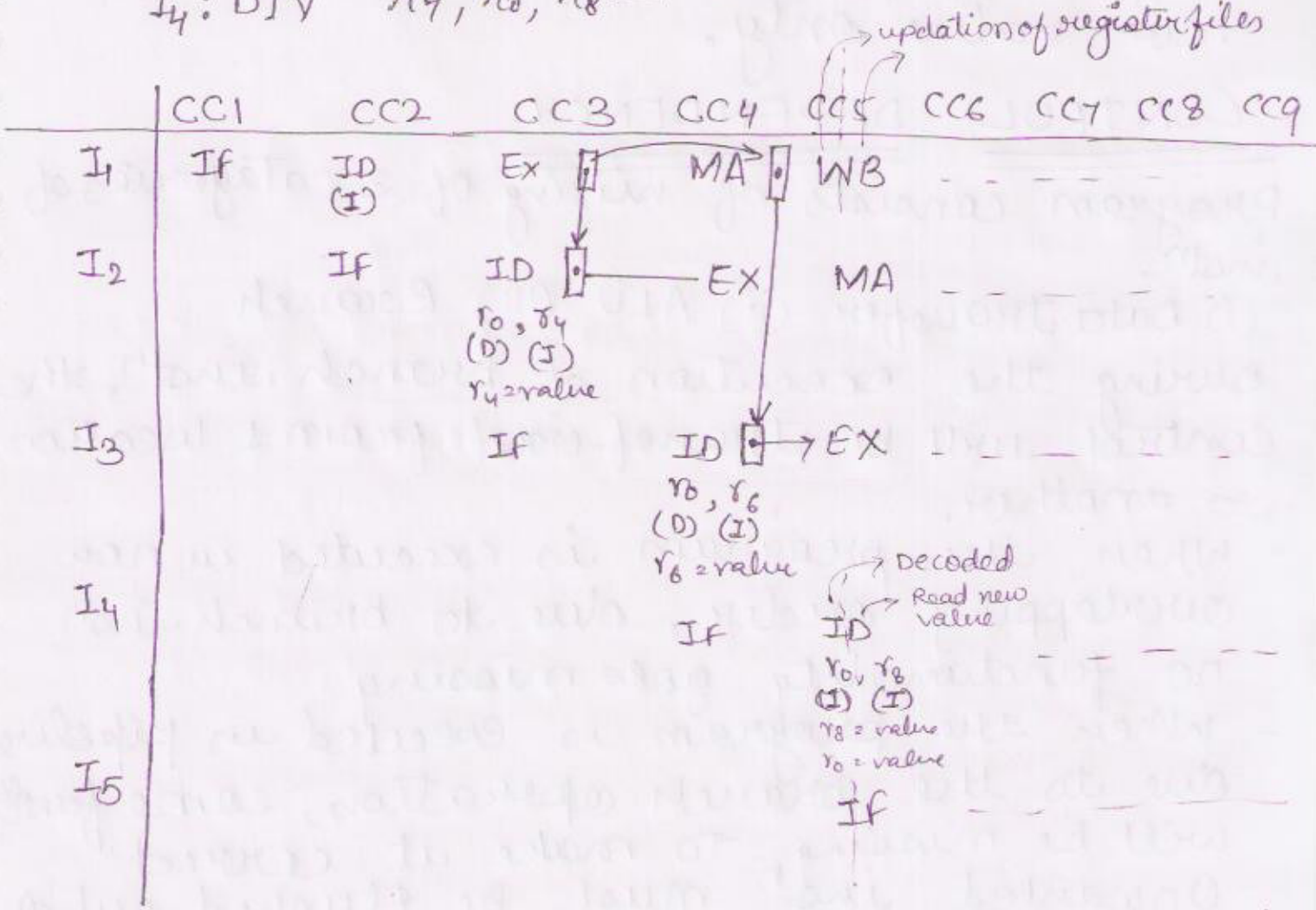
Status 0 means the execution is not yet completed. Execution completed ⇒ Status 1.

	CC1	CC2	CC3	CC4	CC5	CC6
I_1	IF	ID + S: r_1, r_2 (I) (I) D: r_0 r_1 = value r_2 = value	EX	MA	WB	
I_2		IF	ID * S: R_0, r_4 (0) (I) D: r_3 r_4 : value	X	X	EX
I_3			IF	X	X	ID
I_4						IF

CC4 & CC5 are called "stall cycles" because no new input is accepted due to presence of I_3 in IF stage.

- To minimise the no. of stalls, due to data dependencies, one H/w technique is used called as operand forwarding. It is also called as bypassing or short-circuiting.
- Operand forwarding states that access the new data from interface register before updating the register file.

Eg. I_1 : ADD r_0, r_4, r_2 ← True data dependency
I_2 : SUB r_3, r_0, r_4 ←
I_3 : MUL r_5, r_0, r_6 ← dependency but not special name.
I_4 : DIV r_7, r_0, r_8 ←



CC5 divided into two halves. 1st half → Decoding of I_4 & updation of register file as I_1.
2nd half I_4 → reading updated value from reg.

- In the above sequence diagram due to operand forwarding technique, the dependent inst^n (I_2 & I_3) are accessing the data from interface registers before updating the register file.
∴ <u>No wait No stall.</u>

- During execution of I_4, it reads r_0 value from register file. At that time, r_0 contains the new value because CC_5 cycle is divided into 2 halves. In the first half of CC5

I_4 updates r_0 & I_4 decodes ins^n.
In the second half of CC5 I_4 reads the register file. At that time r_0 contains new value only.

CONTROL DEPENDENCY

Program consists of mixing of 3 catagories of inst^n.
(i) Data transfer (ii) ALU (iii) Branch
During the execution of branch inst^n, the control will be transferred from 1 location to another.

- When the program is executed in non overlapping order, due to branch inst^n, no functionality goes missing.
- When the program is executed in pipeline due to the branch operation, some functionality will be missing. To make it correct.
- Unwanted inst^n must be flushed out of the pipeline. This operation creates the stalls in the ~~memory~~ pipeline.

UNCONDITIONAL BRANCH:

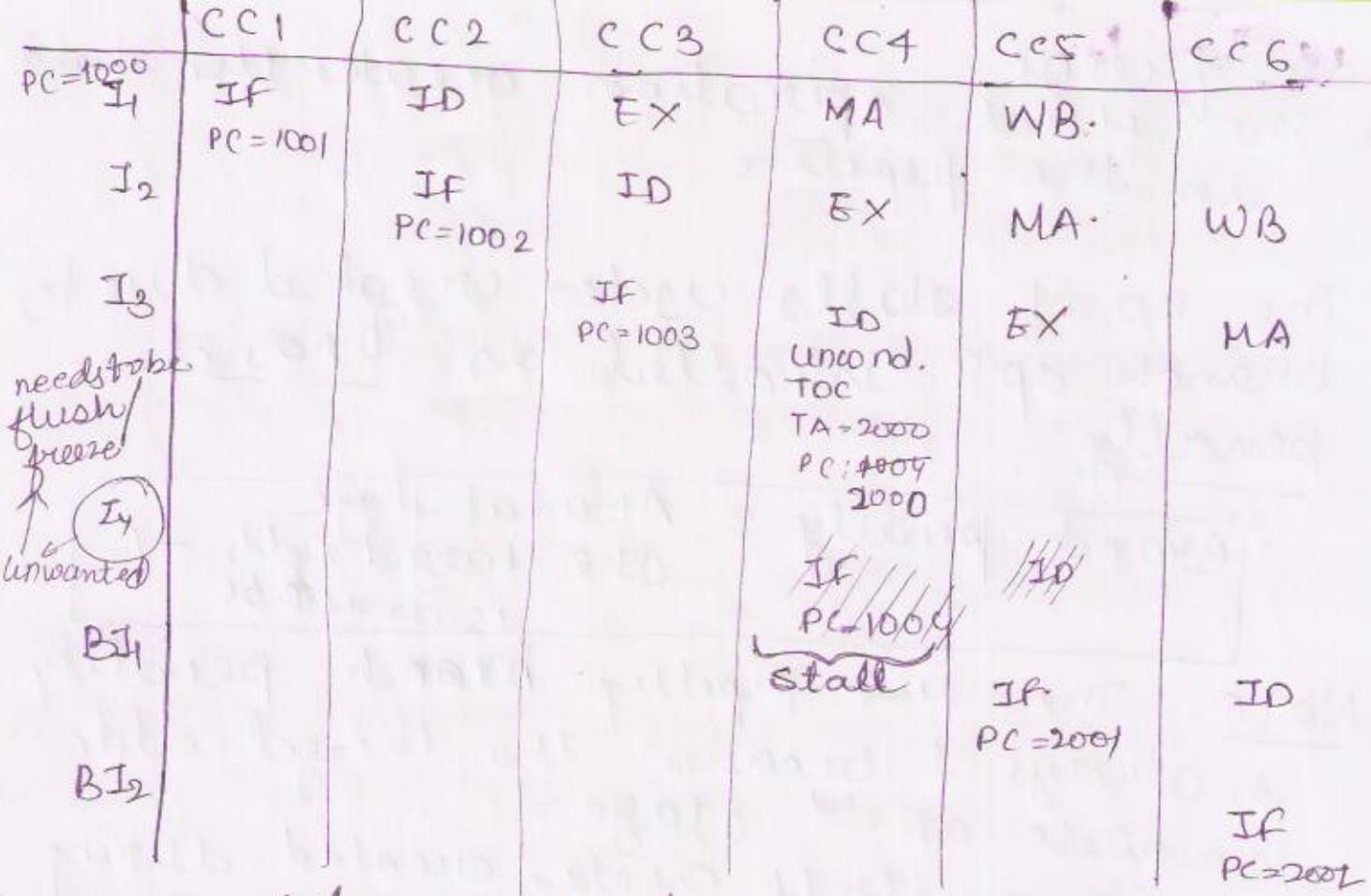
Consider the prog segment.

1000: I_1
1001: I_2
1002: I_3 (Jump 2000)
1003: I_4
:
2000: BI_1
2001: BI_2

↓ Falling Path

↓ Taken path.

Execution seq.
$I_1 - I_2 - I_3 - BI_1 - BI_2 - \ldots$



I_3 completes at CC4 Because uncond. TOC completes at ID stage.

1. Instⁿ feteh stage is overlapping with Instⁿ decodes stage. Before decoding fetched instⁿ the next sequential instⁿ is already inserted into the pipeline.

If decoding stage decodes the fetched instⁿ as data tranfer or data manipulation then next sequential addr becomes the wanted instruction.

When decoding stage decodes the instⁿ as uncond. TOC then the next sequential instⁿ becomes the "unwanted" instⁿ.

To maintain the correct functionality while executing the program, there is a need of removing the unwanted instⁿ from the pipeline.

The process of removing the unwanted instⁿ from the pipeline is called as flush

or freeze.
- The flush operation creates the stalls in the pipeline.

- The no of stalls cycles created due to branch opⁿ is called as "Branch penalty"

> Branch penalty = At what stage the target Addr - 1 is available

NOTE:- The risk pipeline branch penality is always 1 because the target addr is available at 2nd stage.

The no. of stall cycles created during the execution of program, due to branch operation is calculated as

> #stalls from Branches = branch frequency * branch penalty.
> (During Prog. ET).

CONDITIONAL BRANCH:-

Consider the program segment.

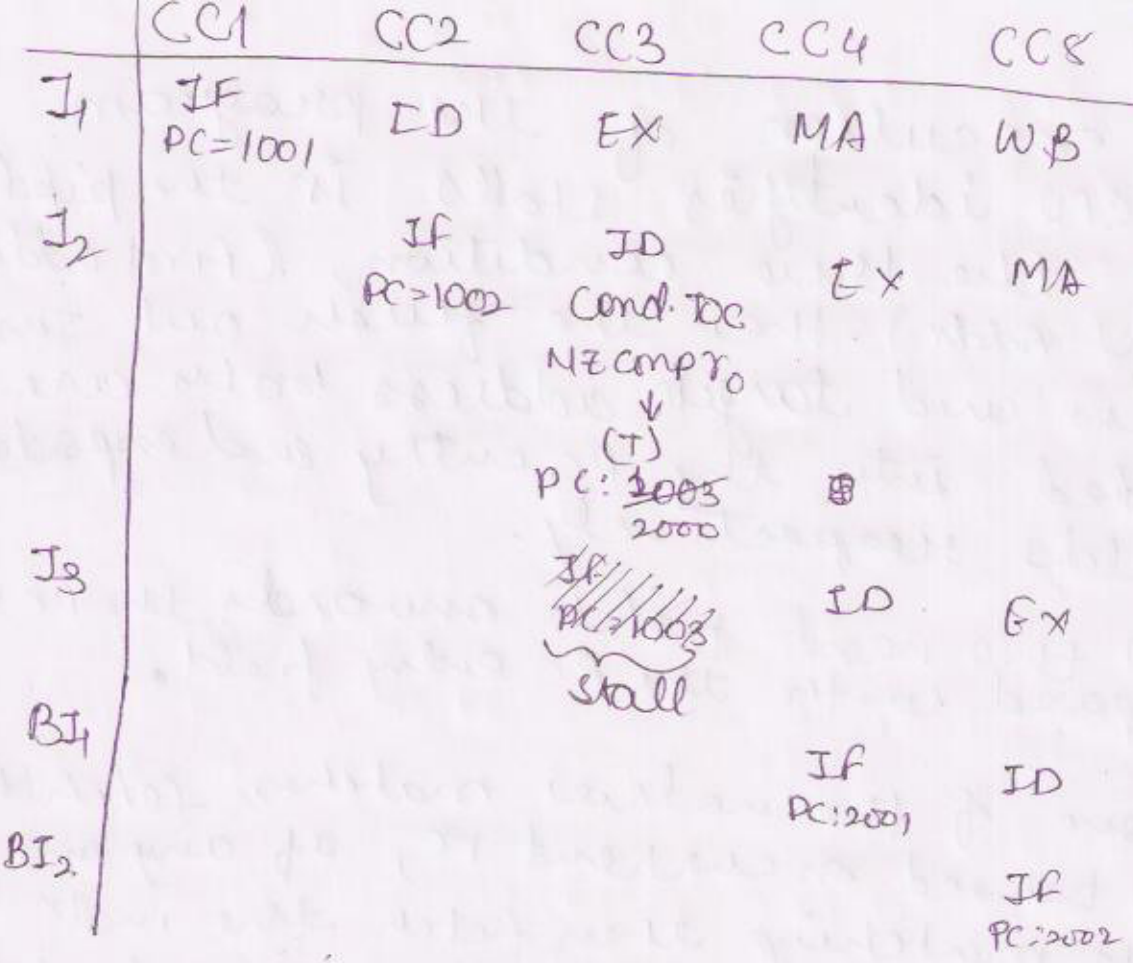
```
1000 : I1
1001 : I2 (JN2 r0, 2000)
1002 : I3
1003 : I4

2000 : BI1
2001 : BI2
```

with true:-

$I_1 I_2 BI_1 BI_2$

with false

$I_1 I_2 I_3 I_4 \ldots\ldots$



- When decoding stage decodes fetched instn as cond. TOC with true condition then next sequential instn becomes unwanted instn. So, flushout the "unwanted" instn. It creates stalls.
- If cond^n evaluates to false, the next seq. instn becomes "wanted" instn, so no flush no stall.

To minimise the no of stalls in pipeline due to branch operations, one h/w technique is used i.e. branch prediction buffer or Branch target buffer or loop buffer. (BPB / BTB / LB).

BTB:- BTB is a high speed buffer maintained at the instn fetch stage to predict the target addresses. Branch target buffer consists of following fields.

PC entry	Expected PC

- During execution of the program when CPU identifies stalls in the pipeline due to the true condition, (first addr is target Addr). then the flush out Instn address and target address both are inserted into the PC entry and expected PC fields respectively.

⇒ from the next fetch onwards the PC value is compared with the PC entry field.

→ If none of the values matches, fetch the instn based on current PC, if any one entry is matching then fetch the instn based on the corresponding expected PC value.

```
1000: MOV r0, #4
1001: ADD r1, r2
1002: DJNZ r0, 1001
1003: MOV
   :
```

PC: 1000 : $r_0 = 4$
1001 : +
1002 : $r_0 = 3 (N2) \Rightarrow T$
flush [1003] 1001 : +
1002 : $r_0 = 2 (N2) \Rightarrow T$
1003
No flush, No stall → 1001 : +
1002 : $r_0 = 1 (N2) \Rightarrow T$
No flush [1003] 1001 : +
1002 : $r_0 = 0$ (Z) ⇒ F
No flush [1003]
flush [1001] 1003

insertn → / ← deletn

PC entry	PC (Expected)
1003	1001

Useful only in loops.

1st flush → insertion } in expected-entry table.
last flush → deletion }

- When there is a stall due to the false condn delete the ~~entry~~ expected PC value from the buffer & fetch the next instn based on the ~~current~~

corresponding PC value.

DELAYED BRANCH :-

- It is a compiler technique. It is used to preserve the execution part in the pipeline
- When there is an unwanted instⁿ present in the pipeline that slot is fixed as the delayed slot.
- When the slot is fixed as delayed then it substitutes the NOP operation.

	CC1	CC2	CC3	CC4	CC5	CC6
I_1	If	ID *	EX	MA	WB	
I_2		If	ID load	EX	MA	WB
I_3			If	ID Uncond TOC	EX	MA
I_4				If (NOP)	ID	EX
BI_1					If	ID
BI_2						If

CC1 is never delayed slot (1st instⁿ in the falling path)

CC2 is a delayed slot

CC3 " " " " → Forwarded because of I1, ID

CC4 " " " " → (fixed) → uncond TOC

CC5 " never delayed slot (1st instⁿ in taken path)

CC6 " delayed slot (depends on decoding of ID)

(wanted or unwanted depends on decoding unit)

INSTRUCTION SCHEDULE :-

The processor executes the program in sequential order called as inorder execution.

In the inorder execution sequence, if any instⁿ is data dependent on other instⁿ, the

remaining instⁿ are also sharing the stall cycles of the dependent instⁿ.

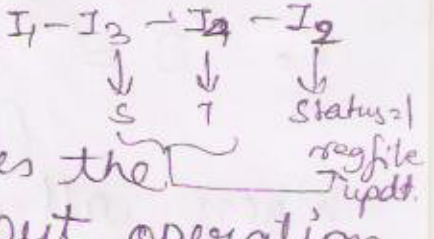
Eg: I_1: Add r_0, r_4, r_2 — Add r_0, r_1, r_2
I_2: Mul r_3, r_0, r_4 ⇒ Mul r_3, r_0, r_4
I_3: Sub r_4, r_5, r_6 — Sub r_5, r_5, r_6
I_4: Div r_3, r_7, r_8 — Div r_4, r_4, r_8

In order Execution sequence :- $I_1 - I_2 - I_3 - I_4$

- In the above example I_2 is data dependent on I_1, so it is undergoes wait until completion of I_1. Due to this, I_3 & I_4 are also sharing stall cycles even they are independent.
- To avoid the above problem, there is a need of schedular instructions.
Scheduling causes the out of order execution (Reorder exeⁿ). $I_1 - I_3 - I_4 - I_2$ (Out of order exeⁿ).
- The out of order exeⁿ creates 2 more dependencies in the pipeline
(i) Antidependency
(ii) Outputdependency
- Antidependency is existed when the instⁿ j tries to write the data before instⁿ i reads it.
eg:- I_3 modifies r_4 before I_2 reads it.
- Output dependency exists when instⁿ j tries to write the data before instⁿ i writes it.
eg:- I_4 executed before I_3, so I_4 writes r_3 before I_2 writes it.

To handle the anti and output dependencies register renaming is used. Register renaming states that use some temp. ~~regis~~ storage (reorder buffer) to store the out of order exec^n outputs. After receive the ~~except~~ acception signal (status=1) from the dependent inst^n, update the reg. file with reorder buffer contents.

HAZARDS:-



- Hazard is a delay. Delay creates the extra cycles. Extra cycle without operation is called stall.

~~Dat~~ There are 3 types of data hazards.

(i) RAW Hazard (Read after Write)
(ii) WAR Hazard (Write after read)
(iii) WAW Hazard (Write after write)

RAW hazard is created when the inst^n 'j' tries to read the data before inst^n 'i' writes it (true data dependency).

WAR H is created when inst^n j tries to write the data before inst^n i reads it (antidependency).

WAWH is created when inst^n j tries to write data before inst^n i write it (output dependency).

Performance evaluation of pipelining with stalls:-

$$S = \frac{\text{Avg ins}^n \text{ exec}^n \text{ of non-pipe}}{\text{Avg inst exec of pipe}} = \frac{\text{Avg Ins}^n \text{ E T}_{np}}{\text{Avg Ins}^n \text{ E T}_p}$$

$$= \frac{CPI_{Np} * \text{cycletime}_{np}}{CPI_{ap} * \text{cycletime}_p}$$

The ideal CPI_p is almost 1 (always).

$$S = \frac{CPI_{np} * cycletime_{np}}{(1 + \#stalls/ins^n) * ct_p}$$

(1 → Ideal, # stalls/ins^n → due to dependencies)

When all stages are perfectly balanced, then both cycle times are equal.

$$\therefore S = \frac{CPI_{np}}{(1 + \#stalls/ins^n)} \quad [\because ct_{np} = ct_p]$$

When all inst^n take same # cycles, then 1 ins^n exe^n is also equal to # stages in pipeline.

$$\therefore S = \frac{\text{pipeline depth}}{1 + \#stalls/ins^n}$$

When system operates with 100% efficiency then no stalls exist.

$$\Rightarrow \boxed{S = \text{pipeline depth}}$$

Pg 17 Q 2

cycle time = 10ns.

Branch freq = 20%

Branch penalty = 5 − 1 = 4 cycles.

Avg inst^n ET = (1 + # stalls/ins) × cycletime

$$= \left[1 + (\text{Branch freq} * \text{Branch penalty})\right] \times \text{cycletime}$$

$$= (1 + (0.2 \times 4)) \times 10\,ns$$

$$= 18\,ns.$$

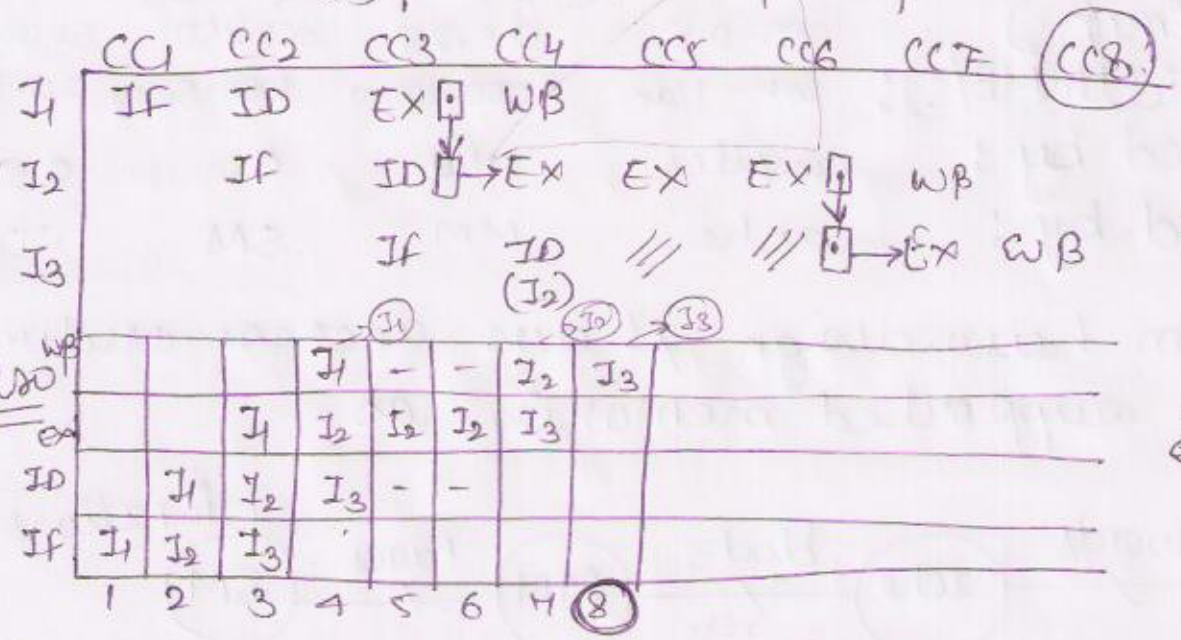
1 inst^n → 18 ns

$$IS = \frac{1}{18} \times 10^9 \text{ ins/sec}$$

$$= 55 \text{ MIPS}$$

Q3 Pg17 :-

	IF	ID	EX	WB
I_1	1	1	1	1
I_2	1	1	3	1
I_3	1	1	1	1



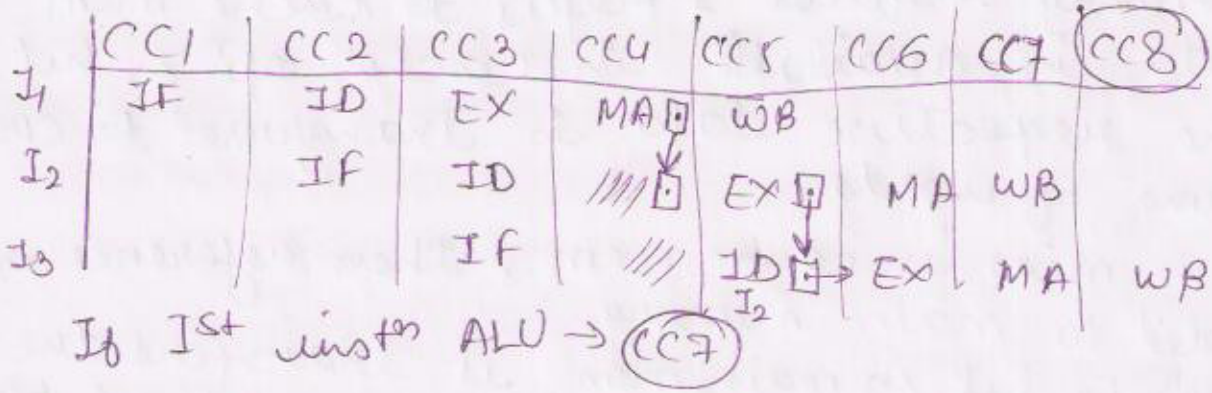
Cycle diagram

Also

space time diagram

(7) Pg18.

	IF	ID	EX	MA	WB
I_1	1	1	1	1	1
I_2	1	1	1	1	1
I_3	1	1	1	1	1



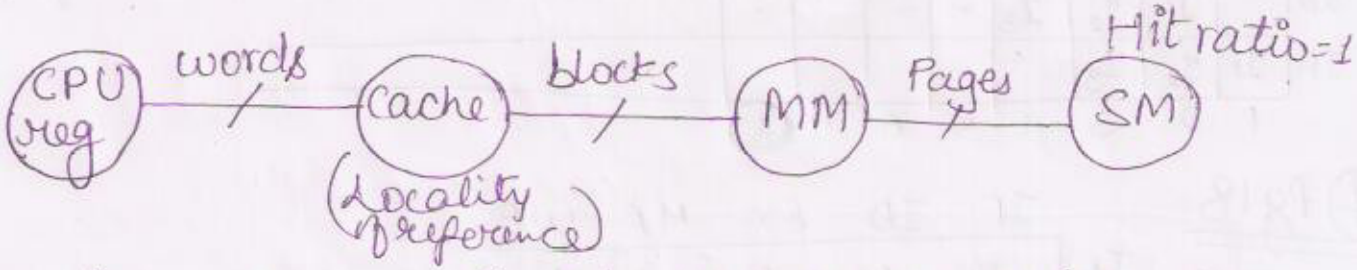
If Ist instn ALU → (CC7)

MEMORY ORGANIZATION :-

Memory org° shows effective utilization of system supported memories.
To satisfy that obj. memory heirarchy design is used.
Memory Heirarchy uses the following specifications :-

level:	1	2	3	4
Name:	Reg.	Cache	MM	SM
Typical Size:	<1KB	<16MB	<16GB	>100GB
Implementat':	custom multiport	SRAM (flip-flop)	DRAM (capacitor)	magnetic
Access time(ns)	0.25-0.5	0.5-25	80-250	50,000,000
Bandwidth(MB/s):	20K-1lac	5000-10000	1000-5000	20-150
Managed by:	compiler	H/W	OS	OS
Backed by:	cache	MM	SM	CD

The mem heirar. design shows access order of system supported memories as



- During the exe^n of program, when CPU encounters memory reference ins^n, it generates memory request & refers to cache mem.
- If data is available in cache, op^n is hit. So, the respective data is transferred to CPU in terms of words
- If op^n is miss in cache mem., then reference is forwarded to main memory.
- when op^n is hit in main mem. it transfers the data to the cache memory in the form of blocks and cache to CPU in form of words. Otherwise, ref. is forwarded to SM.
- The operation is always hit in SM because it is the last level of memory in heirarchy structure. ∴ hit ratio is always 1 in SM.
- ∴ data is transferred from SM to MM in terms of pages; MM to cache in blocks & from cache to CPU in the form of words.
- According to heirarchy design, the data is Xfered from higher to lower. ∴ lower level data is always

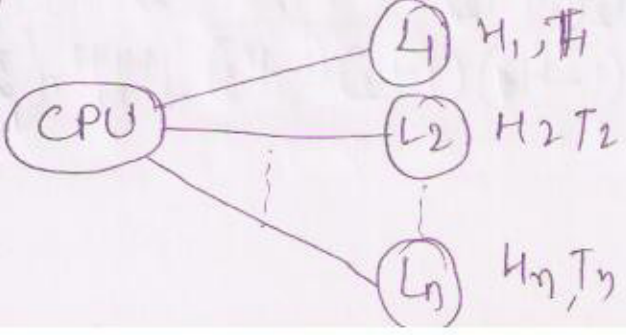
a subset of higher level data.
→ This property is also called as inclusion.
According to hierarchical design, the CPU can perform the read & write op" only the cache memory.
- cache memory is a reusable storage which holds the image of main memory. So CPU performs op" on reusable space. This property is called as locality of reference.

Types of memory organizations:-

Based on way of accessing the system supported memories, the memory org is divided into 2 types:-
(i) Simultaneous Access memory orgn.
(ii) Hierarchical access " ".

Simultaneous access:-

In this org. the CPU is able to access all the levels of memories. In this org. level to level comm. is not allowed.
- CPU refers the level 1 memory to read or write the data.
If there is a miss operation, the reference is forwarded to level 2. If op" is hit in level 2 i.e. directly transferred to the CPU without the involvement of level 1. otherwise reference is forwarded to next level & so on.



where $H_1, H_2, \ldots H_n$ are hit ratios of respec. memories.

Hit ratio = $\frac{\text{no. of hits}}{\text{Total \# accesses}}$

$T_1, T_2 \ldots T_n$ are access time of respec. mem.

- The time required to access 1 operand from memory is called as average access time.

$$T_{avg} = H_1T_1 + (1-H_1)H_2T_2 + (1-H_1)(1-H_2)H_3T_3 + \ldots$$
$$+ (1-H_1)(1-H_2)(1-H_3)\ldots(1-H_{n-1})H_nT_n.$$

1 word = T_{avg}.

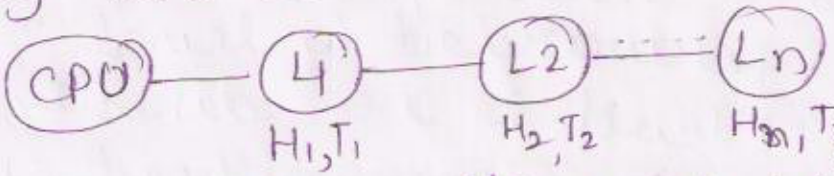
? words = 1 sec.

$$\Rightarrow \boxed{\eta, \text{efficiency} = \frac{1}{T_{avg}} \text{ words/sec}}$$

latency ≈ 0

(Always when cache is used)

Heirarchical Access :- In this org. the CPU can perform read & write op^n only on level 1 memory. If there is a miss in level 1, transfer the data from higher level to level 1, then only CPU can ~~ac~~ read or write.



The avg access time of heirarchical access org. is

$$T_{avg} = H_1T_1 + (1-H_1)H_2(T_1+T_2) + (1-H_1)(1-H_2)H_3(T_1+T_2+T_3) + \ldots$$
$$\ldots + (1-H_{n-1})(1-H_2)(1-H_3)\ldots(1-H_{n-1})H_n\left(\sum_{i=1}^{n} T_i\right)$$

Q. In a 2 level memory, (*) level 1 memory is 5 times faster than level 2 memory. And (!) its access time is 10ns less than avg access time. Let level 1 access time be 20 ns. What is the hit ratio?

⇒ (Simultaneous)

L_1 & L_2.

(*) ⇒ Speed up factor is given

$$S = \frac{T_2}{T_1} = 5 \quad \leftarrow \text{faster}$$

$$\Rightarrow \boxed{T_2 = 5T_1}$$

(!) ⇒ $T_1 = T_{avg} - 10$

$$= \boxed{T_{avg} = T_1 + 10}$$

$T_1 = 20$ ns

$\therefore T_{avg} = 30$ ns

& $T_2 = 100$ ns.

$$T_{avg} = H_1 T_1 + (1-H_1) H_2 T_2$$

$$= H_1 T_1 + (1-H_1) T_2$$

$$30 = H_1 * 20 + (1-H_1) 100$$

$H_1 = 0.87$ Ans

last level Hit ratio = 1 (always)

Q. 3 level memory has following specifications.

level	Access time/word	Block size (word)	Hit ratio
1	20	—	0.8
2	50	2	0.9
3	100	1	1

If the required block is not present in L_1, transfer it from L_2 to L_1, if not available in L_2, transfer from L_3 to L_2 & L_2 to L_1. How long will it take to transfer the blocks

Ans ⇒ $T_1 = 20 \times - = 20$ & $T_3 = 100 \times 4 = 400$

$T_2 = 50 \times 2 = 100$

$$T_{avg} = H_1 T_1 + (1-H_1) H_2 (T_1+T_2) + (1-H_1)(1-H_2) H_3 (T_1+T_2+T_3)$$

$$= 0.8 \times 20 + 0.2 \times 0.9 (120) + (0.2)(0.1)\,1\,(520)$$

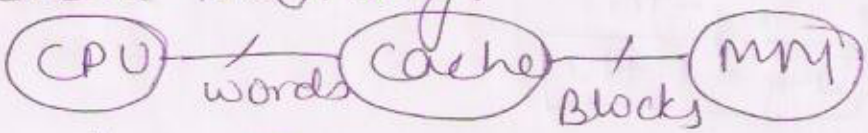
$$= 16 + 18 (120) + 0.02 \times 520$$

$$= 16 + 21.6 + 10.40$$

$$= 48 \text{ ns}.$$

CACHE MEMORY:-

Cache memory is used as the intermediate memory between CPU & main memory.
∴ CPU perform read & write op only on the cache memory.



Basic elements are
(i) Memory organization
(ii) Mapping techniques
(iii) Replacement algorithms
(iv) Updating techniques
(v) Multilevel caches

Memory organization: The data is transferred from main memory to cache memory in form of blocks. So both memories are divided into equal parts based on block size. Each part is called as cache mem. block & main memory block respectively.

$$\#\text{cache mem. blocks (lines)} = \frac{\text{cache mem size}}{\text{block size}}$$

$$\#\text{ main memory blocks} = \frac{\text{MM size}}{\text{Block size}}$$

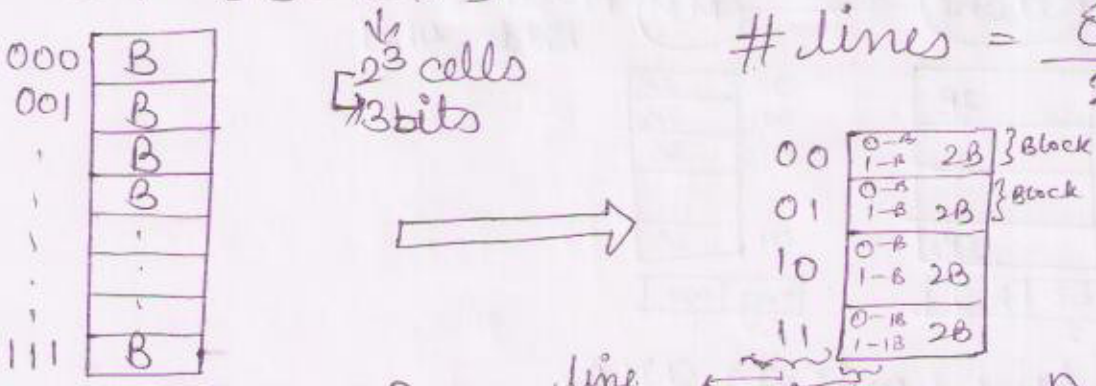
Consider 8 byte cache memory & 2 byte block size.

CM = 8B = 8 X B

2^3 cells

3 bits

$$\#\,lines = \frac{8B}{2B} = 4$$



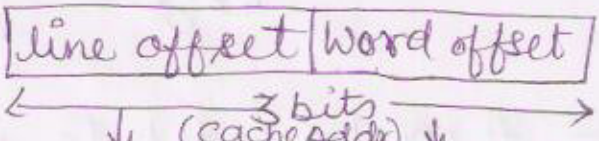
Before org^n

After org^n

line offset

word offset

Before and after organization of cache memory there is no change in capacity but the internal structure differs ∴ After cache mem. org., address can be interpreted as



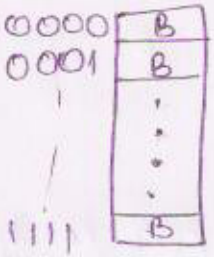
line no in cache → $\log_2$ # lines → $\log_2 4 = 2$ bits

word addr in the block → $\log_2$ Block size → $\log_2 2 = 1$ bit

Consider 16 byte main memory & 2B block size.

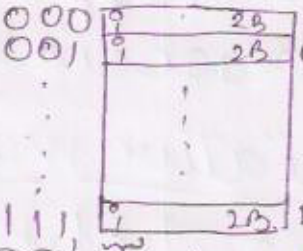
Before org^n

MM = 16B = 4 bit addr

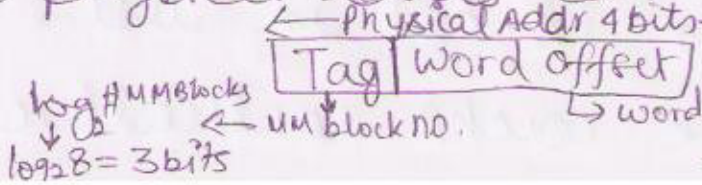


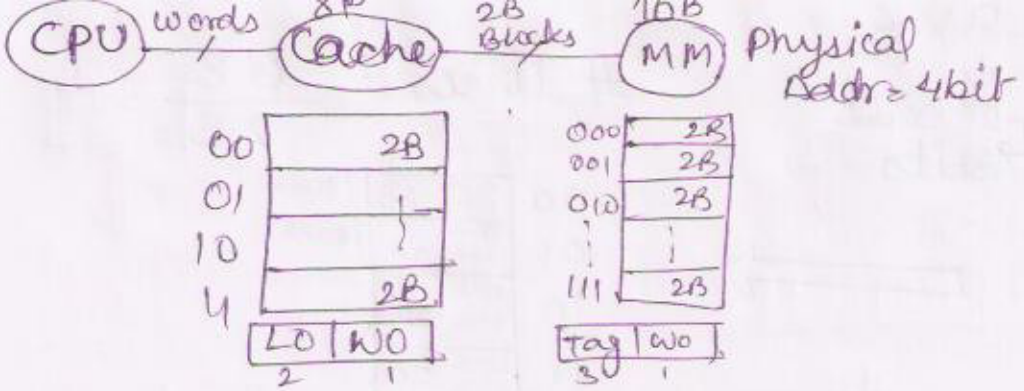
After org^n

MM Blocks = $\frac{16}{2} = 8$

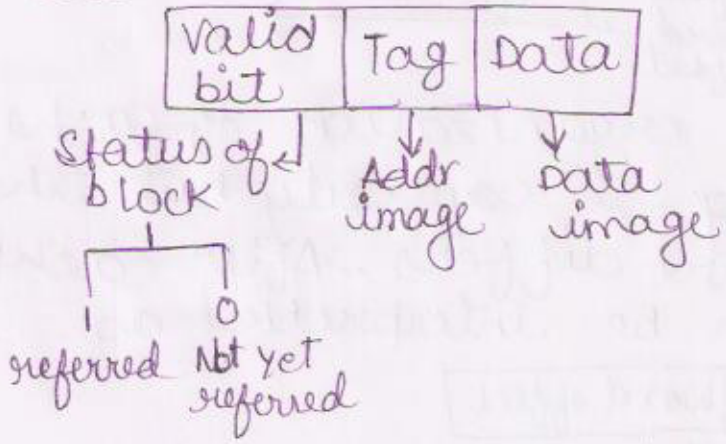


Before & after org^n of the MM, there is no change in capacity but the internal structure differs so, physical addr can be interpreted as





The cache line contents are



Cache mem size = tag mem size + Data mem size

Tag mem size = # cache lines * # tag bits in each line

Data mem size = # cache lines * # bits in each line

MAPPING TECHNIQUES

The process of transferring data from MM to cache memory is known as mapping.

There are 3 mapping techniques used:-

(i) Associative mapping
(ii) Direct mapping
(iii) Set associative mapping.

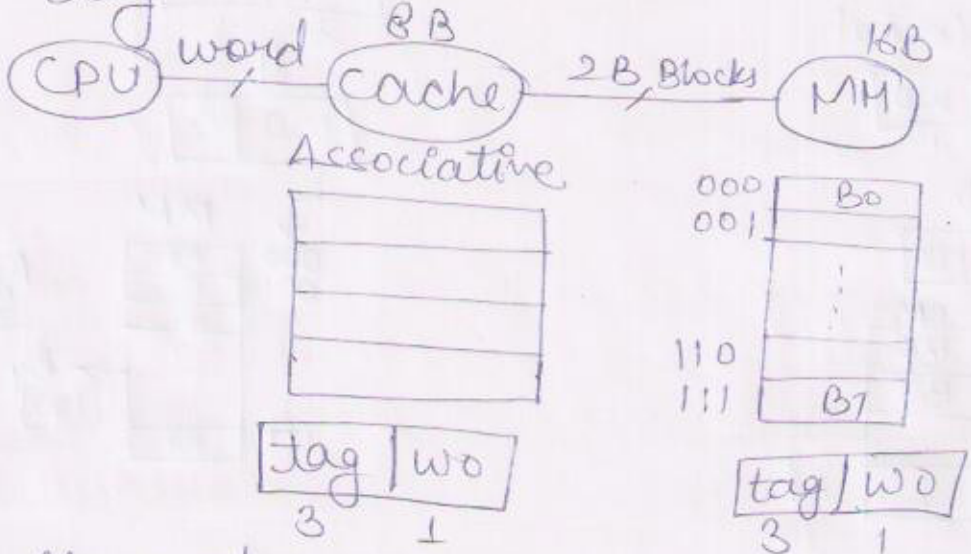
Associative mapping:-

In this technique, there is no mapping funct^n used to transfer the data. That means any MM block can be mapped to any cache line

3. Cache address is not in use.

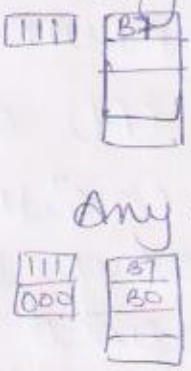
- The associative cache controller inter-

- prets the CPU requests based on MM address format.
- During the mapping, the complete data block is transferred to CM along with Tag.



(i) initially empty cache.

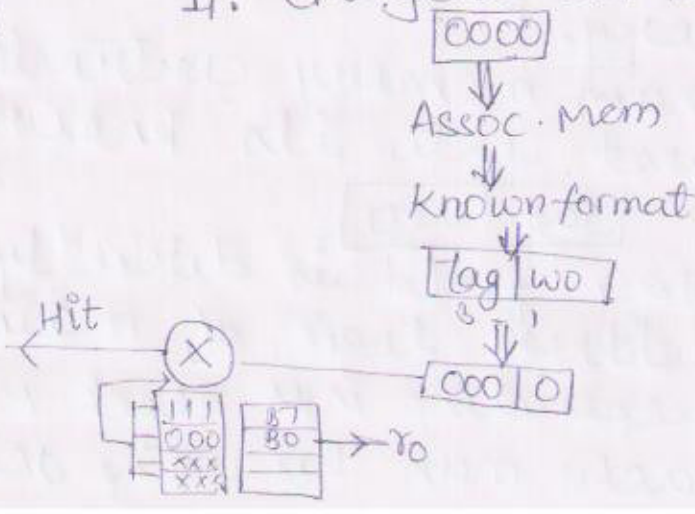
	MM Block	Associative Mapping	CM Line
(ii)	B7 [111]	No mapping fun^n	Any 1 line
(iii)	B0 [000]	No mapping fun^n	Any one line



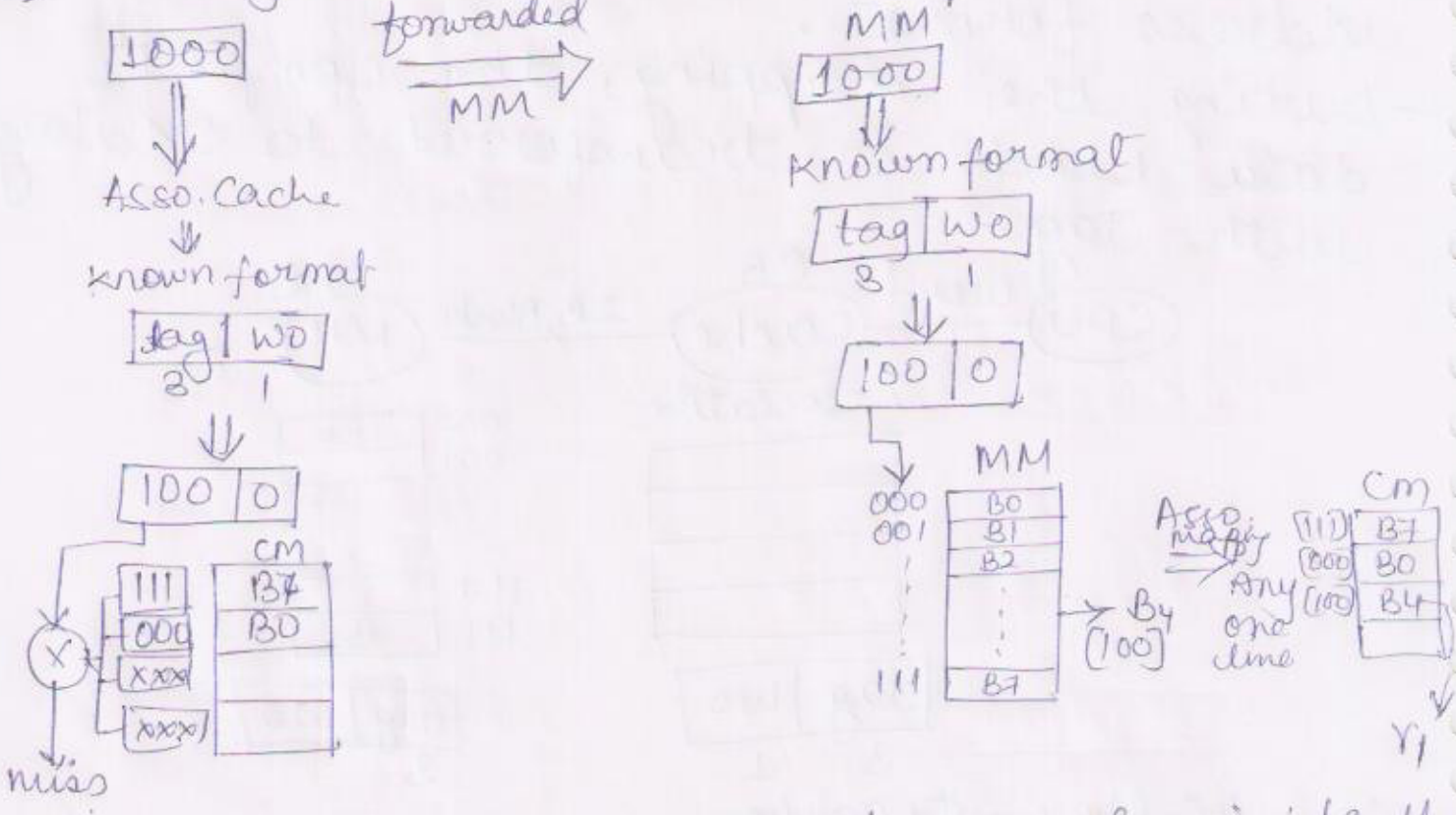
Consider the program segment :-

(i) MOV r0, [0000]
(ii) MOV r4, [1000]
(iii) Add r0, r4

I_1: CPU generates mem req.



I_2 CPU generates the mem req.



The CPU generated mem request is initially referred to associative cache. The assoc. cache controller interprets the request into its known format i.e. tag | WO

- The CPU generated tag is compared with the existing tags in the cache controller.
- If any one is matching, operation is hit. So based on the word offset, the data is transferred into CPU.
- If none of them is matching op^n is miss. So the reference is forwarded to main memory.
- The main memory controller interprets the request into its known format i.e.

tag | WO

- The tag field is directly connected with the address logic of main memory. So corresponding MM block is enabled & transferred to cache mem. by using associative mapping

technique. Later data is transferred to CPU. The tag memory size is equal to # of cache lines * # tag bits in each line.

Tag mem size = 4 * 3 = 12 bits

DIRECT Mapping:-

In this technique memory funct^n is used to Transfer the data. The mapping function is

K mod N = i

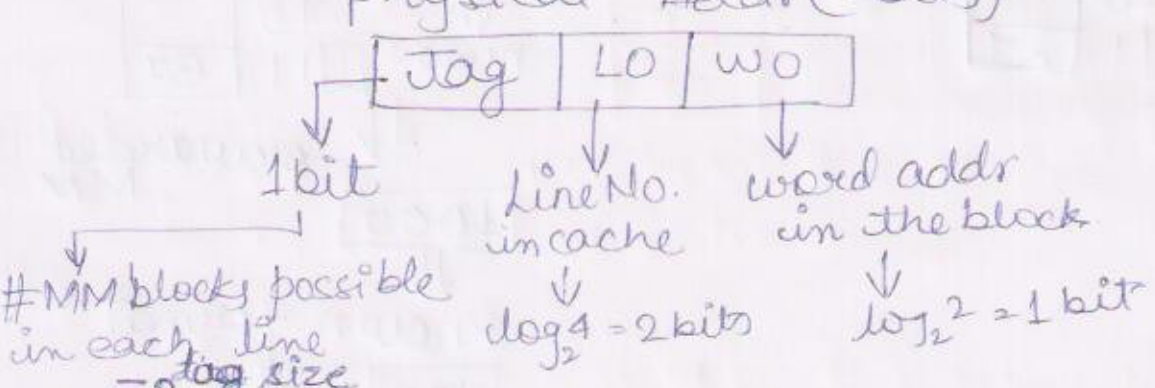
K → MM Block #
N → # cache lines
i → CM line number

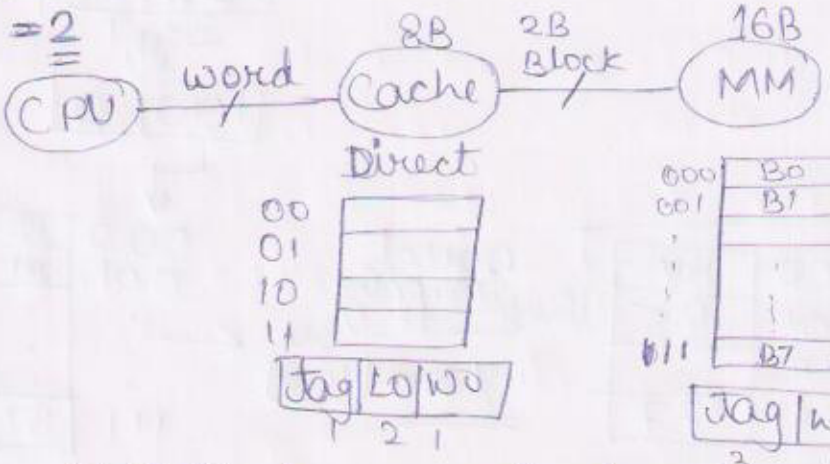
(0-7) mod 4 = (0-3 2 0-3)

The above mapping fun^n shows the relationship b/w MM block & cache mem line.

∴ cache mem address is in use.

The direct cache controller interprets the CPU generated requests as





MM Block

B0 [000]

B7 [111]

Direct Mapping

K mod 4 = i

0 mod 4 = 0 → line 0

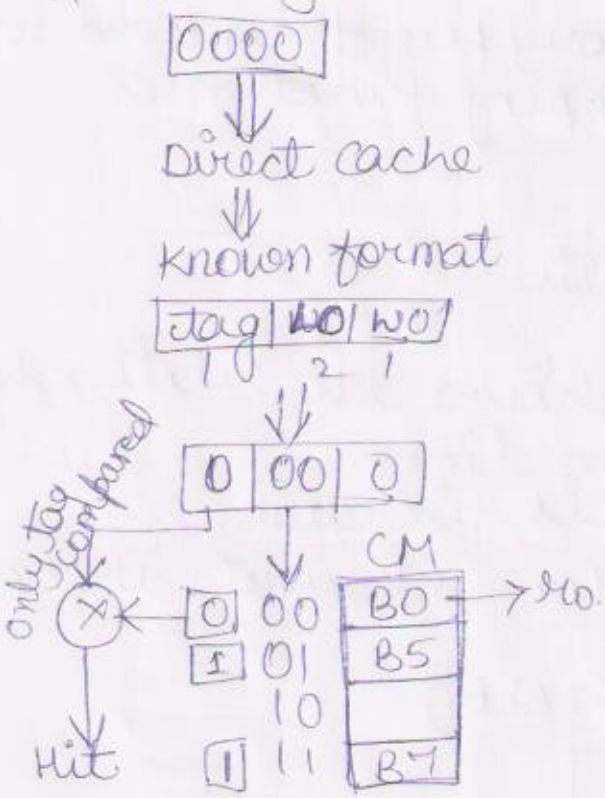
7 mod 4 = 3 → line 3

CM Line

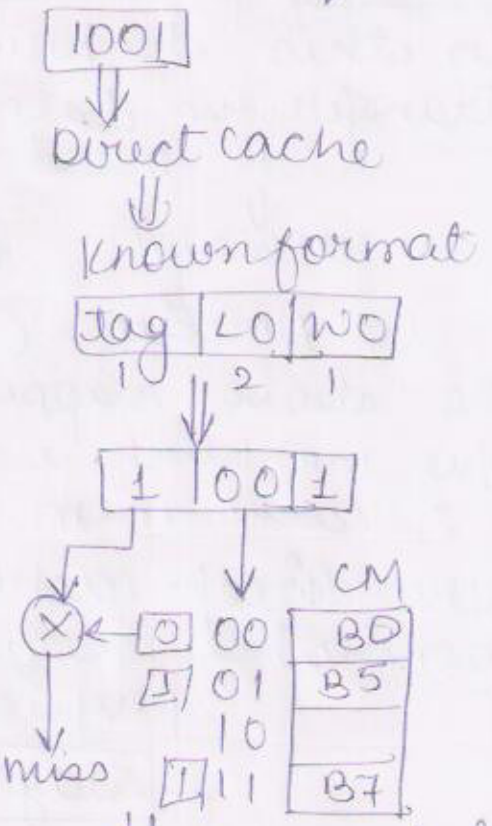
Consider the program segment :-

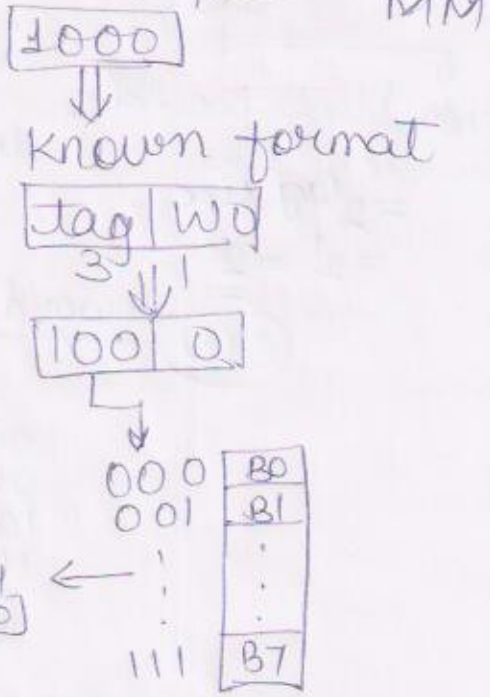
I_1: MOV r_0, [0000]
I_2: MOV r_4, [1001]
I_3: Add r_0, r_4

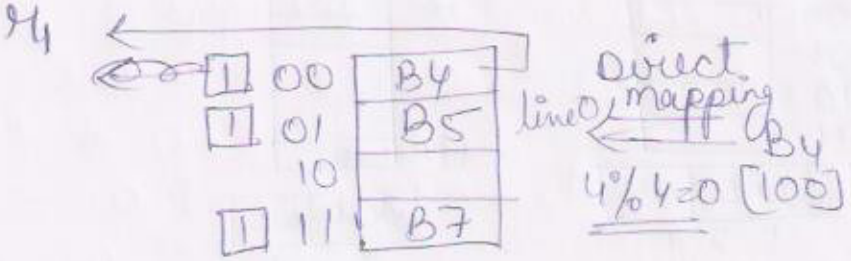
I_1: CPU generates mem req



I_2: CPU generates mem req.







Tag memory size = 4 lines × 1 bit = 4 bits.

- Direct cache controller interprets the req. into its known format i.e. [tag | LO | WO]
- The line offset is directly mapping with

the address logic of cache memory... Correspondingly line is enabled.
- The existing tag in enabled line is compared with CPU generated tag.
- when both matches, the opn is hit. So based on the word offset data is transferred to CPU. otherwise it is a miss. Reference is forwarded to MM
- The required block is transfered from MM to cache memory based on the direct mapping function.

(7D)

Disadvantage in direct mapping is that each cache line is able to hold one block at a time. ∴ Conflict misses will be increased.

Direct

Tag		
0	00	B0
1	01	B5
	10	
	11	

B_4 B_0 B_4

I_1: MOV r_0, [0000] ⇒ B_0, 0th Byte, Hit
I_2: MOV r_0, [1000] ⇒ B_4, 0" , Miss
I_3: MOV r_0, [0001] ⇒ B_0, 1st Byte, Miss
I_4: MOV r_0, [1001] ⇒ B_4, " , Miss

Alternative cache organization

0	B_0	B_4
1	B_5	

I_1: Hit
I_2: Miss
I_3: H
I_4: H

[SET-ASSOCIATIVE CACHE ORGANIZATION]

To overcome the disadvantage of direct mapping, alternative cache organization is used. In this, each cache line is able to hold more than 1 block at a time. The alternative cache organization is called as SET ASSOCIATIVE CACHE.

SET ASSOCIATIVE CACHE ORGANIZATION

In this organization, cache memory lines are again divided into equal parts based on no of main memory blocks can be placed at a time at each cache line.

Each part is called set NO. of sets, S =

$$\frac{\text{no. of cache lines}}{\#\text{blocks placed at a time in each line.}} = \frac{N}{p\text{-ways}} = S$$

Consider 2-way set associative cache of 8 bytes, with block size 2 bytes.

Before org (BB)

000	B
001	B
010	
011	
100	
101	
110	
111	B

After org into the lines

00	2B
01	2B
10	2B
11	2B

$\#\text{lines} = \frac{8}{2} = 4$ direct cache

After org into the sets

$\#\text{sets} = \frac{4}{2} = 2$

0	2B	2B
1	2B	2B

Set associative cache.

Set Associative Mapping: In this technique, mapping function is used to transfer the data from main memory to cache. The mapping function is $k \bmod s = i$ where

k = main memory block number.

s = # cache sets

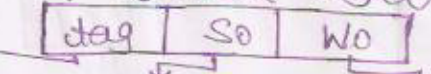
i = cache memory set no.

The above function shows the relationship b/w MM block & Cache memory set. i.e. Cache memory address is in the use.

The set associative cache controller interprets the CPU generated ~~requests~~ address request

into the following format :-

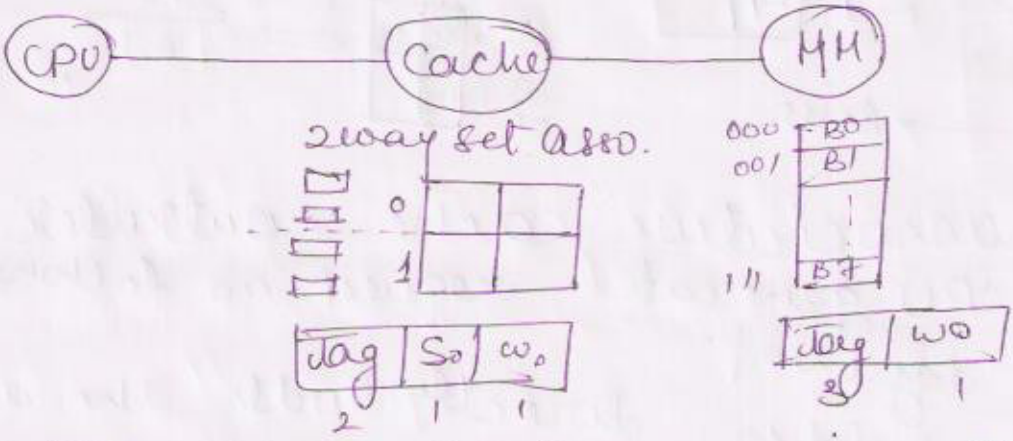
physical address (4 bit)



2-bit → # MM Blocks possible in each set is $= 2^{Tag} = 4$.

set no. in the cache → $\log_2 \#sets$ → $\log_2 2$ → 1 bit

word addr in the block → $\log_2$ blocksize → $\log_2 2 = 1$



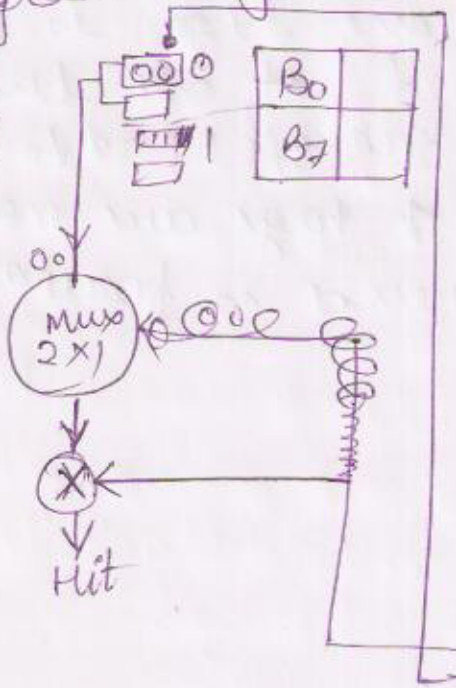
Main memory Block	Set Assoc. mapping	CM set
B0 [000]	K mod 2 = i 0 mod 2 = 0	set 0
B7 [111]	K mod 2 = i 7 mod 2 = 1	set 1.

Consider the program segment :

I_1 : MOV r_0, [0000]

I_2 : MOV r_1, [1001]

I_3 : Add r_0, r_1



I/P CPU generate the mem req.

0000

↓

2-way set Assoc.

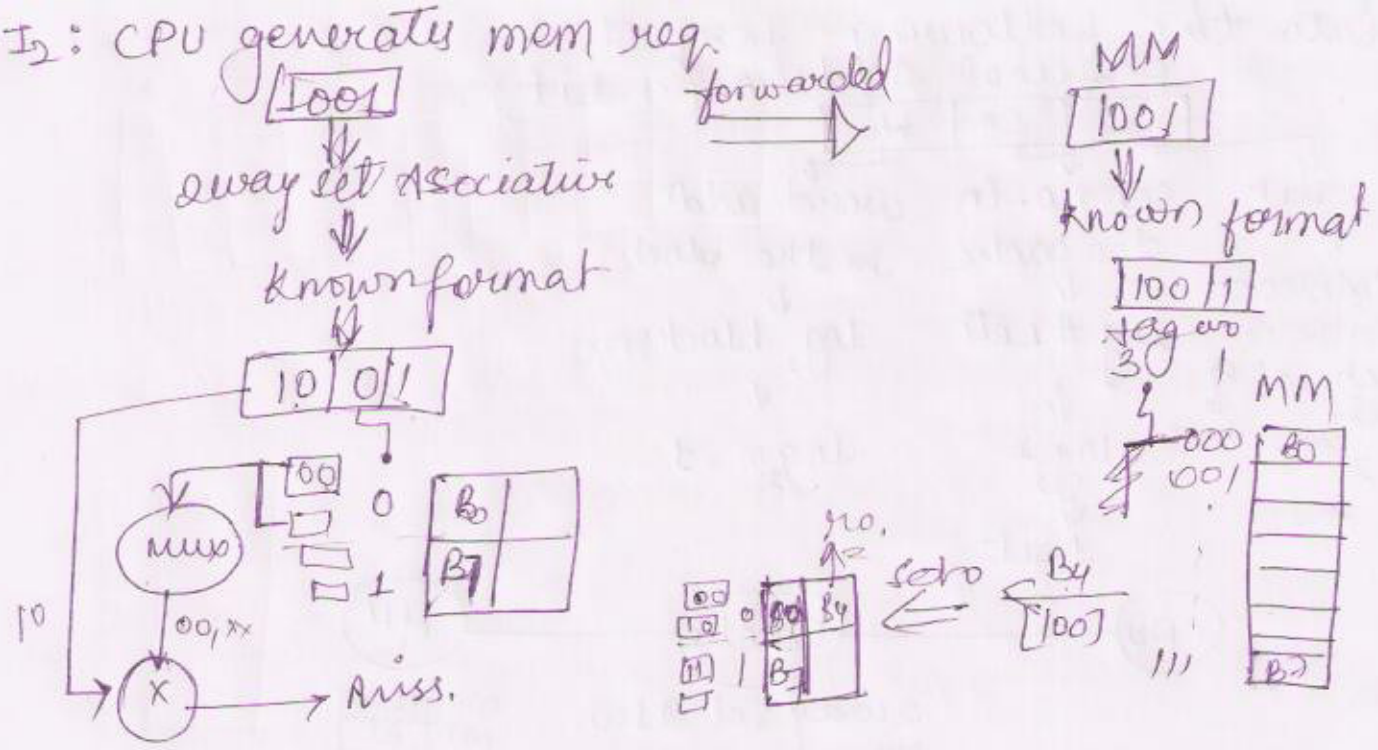
↓

Known format

tag	S0	w0
2	1	1

↓

00	0	0



Set associative cache controller interprets the CPU generated request as follows:-

tag | word |

Set field is directly with the address logic of cache memory. The respective cache set will be enabled.

Each set contains multiple tags. So to compare the existing tags in the enabled set with the CPU generated tag there is a need of multiplexer.

- If any one tags is matching, opr^n is hit So based on the word offset, the data is transferred to the CPU.

If none of tags are matching, opr^n is miss so the reference is forwarded to main memory

Based on the tag value, the respective block is transferred to the cache memory by using the set associative mapping technique

tag memory size = # sets * # Blocks in each set * # tag bits.

= S * P * # tag bits

= 2 * 2 * 2 = 8 bits

REPLACEMENT ALGO

When the cache is full, there is a need of replacement algorithms to replace the existing cache blocks with new blocks.

There are 2 replacement algos used.

(i) FIFO (ii) LRU.

In First In First out, replace the cache block with new block which is having the longest time stamp.

In least recently used algo, replace the cache block with new block which is having the less no. of references with longest time stamp.

Q. Consider 4 block cache memory (initially empty) with the following MM block references.

4, 5, 7, 12, 4, 5, 13, 4, 5, 7

Identify the hit ratio with:-

(i) FIFO (ii) LRU (iii) Direct mapped cache

(iv) 2 way set assoc. with LRU.

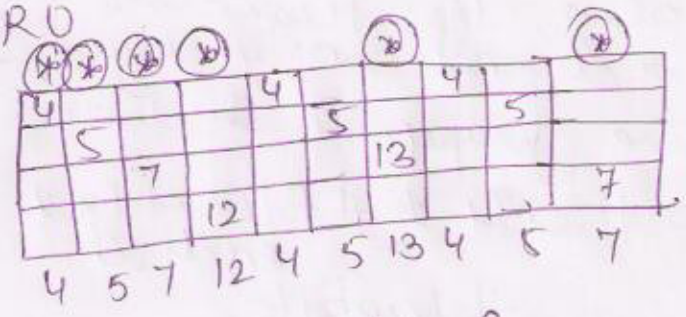
(i)

M	M	M	M	H	H	M	M	M	M
4	4	4	4	4	4	13	13	13	13
	5	5	5	5	5	5	4	4	4
		7	7	7	7	7	7	5	5
			12	12	12	12	12	12	7

4 5 7 12 4 5 13 4 5 7

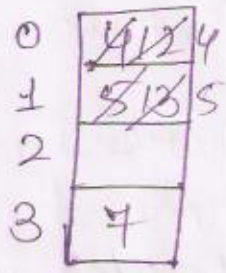
Hit ratio = $\frac{2}{10} = 0.2$

(ii) LRU



hit ratio = $\frac{4}{10}$ = 0.4

(iii) Direct mapped

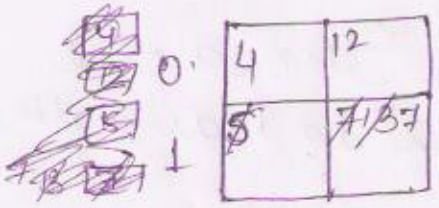


m, m, m, m, m, h, m, h, m, h

hit ratio = $\frac{3}{10}$ = 0.3.

(iv) 2-way set asso with LRU :-

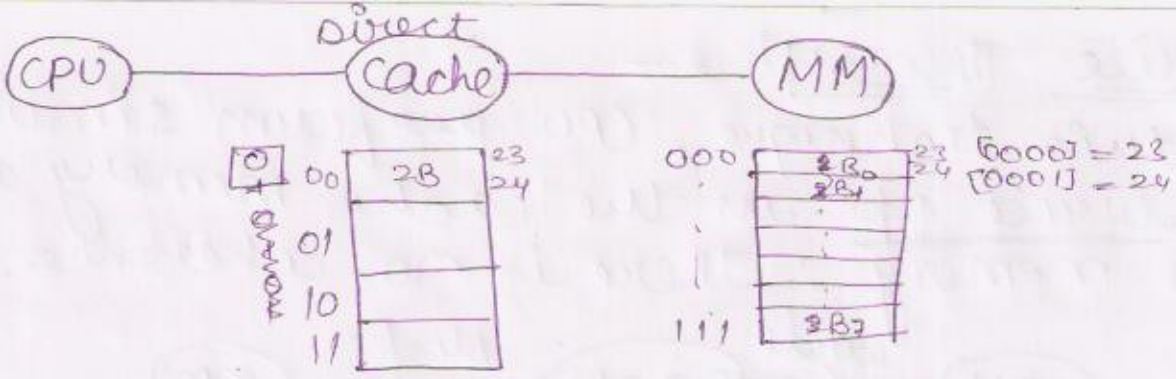
4 5 7 12 4 5 13 4 5 7
m, m, m, m, h, h, m, h, h, m



hit ratio = $\frac{4}{10}$ = 0.4.

UPDATING TECHNIQUES :-

CPU perforums read & write operation only on the cache ~~mm~~memory.
Before operation, it checks for availability of block in cache. If the block is availble, operation is called as read hit or write hit.
If the block is not available, operation is called as read miss/write miss.
When there is a miss operation, the required block is transferred from MM to cache memory. This process is called read allocate / write allocate.
After the allocate, CPU performs read/write operation.



I_0: MOV r_0, [0000] ; B_0, 0th byte → r_0, H $r_0 \leftarrow 23$

I_1: Add r_0, #24 ; $r_0 \leftarrow r_0 + 23$; $r_0 = 47$

I_2: MOV [0000], r_0 ; B_0, 0th Byte → r_0 ⇒ CM, [0000] ← 47

23 47

MM [0000] 23 ⟷ CM [0000] 47

Coherence.

During the execution of above program, inst^{n} 3 indicates mem wr opn. so, CPU perform write opn only on the cache mem. Therefore, different values are maintained in cache memory & main memory with the same address.

- The same address contains different values at different places is called as Coherence

→ Coherence creates the loss of data problems, i.e.

I_3: Mov r_0, [1000] ; B_4 - 0th byte ⇒ M: 4 mod 4 = 0, CM = ~~B0~~ B4 (updated data lost)

I_4 : —

I_5 : —

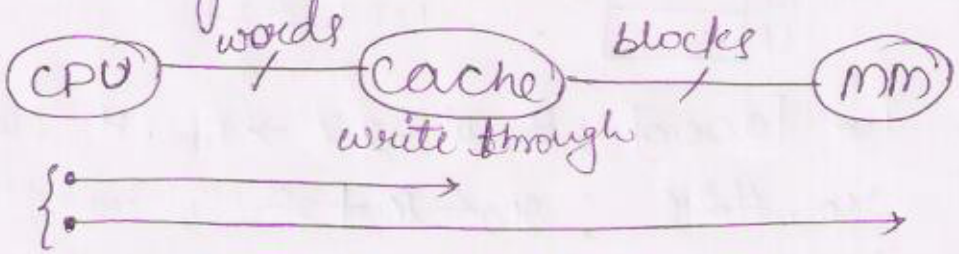
I_6 : mov r_4, [0000] ; B_0 - 0th Byte ⇒ M: CM = 0 mod 4 = 0 ⇒ ~~B4~~ B0 23 (old) 24

During the execution of I_4, due to the miss operation, the updated cache block is replaced with new block. So the updated information is lost. To avoid this problem, updating techniques are used:-

(i) write through (ii) write back.

(i) Write Through :-

In this technique, CPU perform simultaneous write op^n in the cache memory and main memory. ∴ There is no coherence.



$$\text{word updation time } (T_w) = \max\left(\text{word updation time in cache}, \text{word updation time in MM}\right)$$

The average access time of read cycle is

$$T_{avg\,read} = H_R T_c + (1-H_R)(T_m + T_c)$$

H_R → read hit; T_c → read; $(1-H_R)$ → read miss; T_m → read allocated; T_c → read.

The average access time of write cycle is

$$T_{avg\,write} = H_w * T_w + (1-H_w)(T_m + T_w)$$

H_w → write hit; T_w → word updation time; $(1-H_w)$ → write miss; T_m → write allocate; T_w → word updation time.

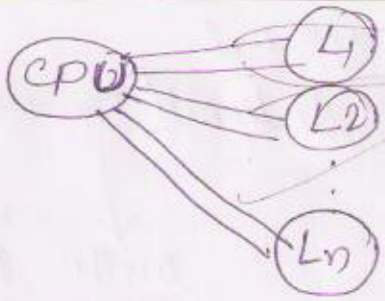
The total average accesstime when considering both read & write is

$$T_{avg_{WT}} = f_r * T_{avg\,read} + f_w * T_{avg\,write}$$

f_r → frequency of read op^n; f_w → frequency of write op^n.

$$\eta_{WT} = \frac{1}{T_{avg_{WT}}} \text{ words/sec}$$

NOTE:- Write through is also ~~avail~~ applicable to simultaneous access memory organizat^n



→ Simultaneous signals generated.
∴ hit ratio = 1.

$T_{avg} = H_{cc} T_{mm} \Rightarrow \boxed{T_{avg} = T_n}$

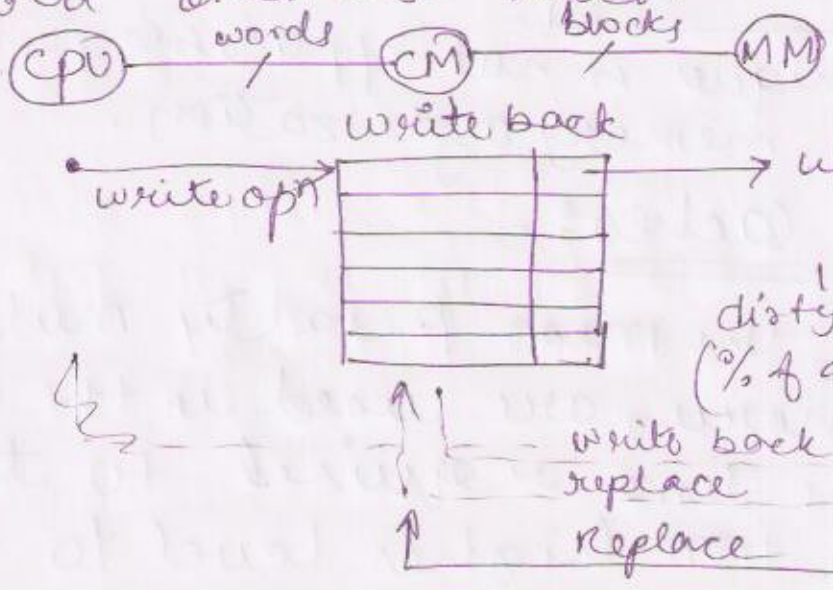
Note:- If $H_w = 1 \Rightarrow$ (simultaneous mem. org.)
$H_w \neq 1 \Rightarrow$ (Hierarchical " . ").

Write back:-

In this technique, the CPU performs write operations only in the cache memory so still coherence is present but this coherence does not create ~~the~~ data loss problem because ~~the~~ CPU reads states of cache block before replace.

If the cache block is updated block then, it transfers the information back to main memory. Later replaces the cache block. Otherwise the new block will be placed into the cache.

In write back cache, each line maintains one extra bit called as dupdate bit. This bit will be set when the block is updated otherwise reset.



The avg access time of read cycle is

$$T_{avgread} = H_r \cdot T_c + (1-H_r)[\%\text{ dirty bit}(T_m + T_m + T_c)$$

H_r: read hit; T_c: read; $(1-H_r)$: read miss; T_m: write back; T_m: Read allocate; T_c: read

$$+ \%\text{ clean bits}(T_m + T_c)$$

T_m: Read allocate; T_c: read

The avg access time of write cycle is

$$T_{avgwrite} = H_w T_c + (1-H_w)[\%\text{ dirty bits}(T_m + T_m + T_c)$$

H_w: write hit; T_c: write; $(1-H_w)$: write miss; T_m: read allocate; T_m: write allocate; T_c: write

$$+ \%\text{ clean bits}(T_m + T_c)]$$

T_m: write allocate; T_c: write

The total average access time when considering both read & write opr^n is

$$t_{avgWB} = f_r * t_{avgread} + f_w * t_{avgwrite}$$

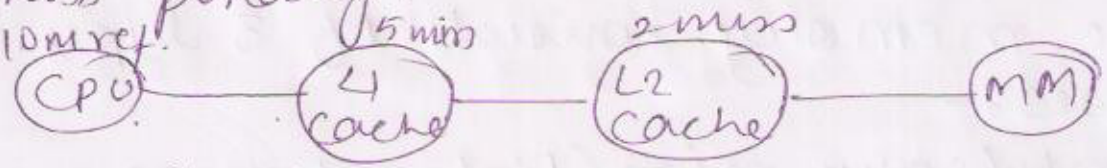
$$\eta_{WB} = \frac{1}{T_{avgWB}} \text{ words/sec}$$

This technique is not applicable for simultaneous access memory organization.

Multilevel Caches:-

To reduce the miss penalty multilevel cache memories are used in the system design. The time required to transfer the data from higher level to level 1

memory due to the miss operation is called as miss penalty.



In multilevel cache org^n we can calculate two types of miss-rate

(i) local miss rate

(ii) Global miss rate

$$\text{Local miss rate} = \frac{\#\text{ misses in cache}}{\text{Total no. of accesses to that cache.}}$$

$$\text{Global miss rate} = \frac{\#\text{ misses in cache}}{\text{Total \# CPU generated references.}}$$

$LMR_{L_1} = 5/10 = 0.5$

$LMR_{L_2} = 2/5 = 0.4$

$GMR_{L_1} = 5/10 = 0.5$

$GMR_{L_2} = 2/10 = 0.2$

} always same.

In multilevel cache org. the avg. access time calculated as.

$$T_{avg} = \text{Hit time}_{L_1} + \text{miss rate}_{L_1} * \text{miss penalty}_{L_1}$$

$$\text{Miss penalty}_{L_1} = \text{Hit time}_{L_2} + \text{miss rate}_{L_2} * \text{miss penalty}_{L_2}$$

$$\text{Miss penalty}_{L_2} = \text{main memory access time.}$$

In the multilevel cache org^n, avg stall cycles per inst^n is calculated as

$$\text{Avg stalls/inst}^n = \#\text{ misses in } L_1/\text{Ins}^n * \text{Hit time}_{L_2} + \#\text{misses in } L_2/\text{Ins}^n * \text{miss penalty}_{L_2}$$

Types of MISSES:-

Cache memory consists of 3 types of misses:-

① Compulsary miss (First-reference or cold start miss)

This miss will occur when the very first reference to cache is miss (cache is initially empty).

(ii) Capacity miss :- Due to the small size of cache, it cannot hold all memory blocks required for program. So some blocks are discarded & later retrieved.

(iii) Conflict miss (collision miss/Interference miss)

This miss will occur when too many blocks are mapped into the same cache line or cache set.

Types of locality of reference :-

(1) Temporal locality :-

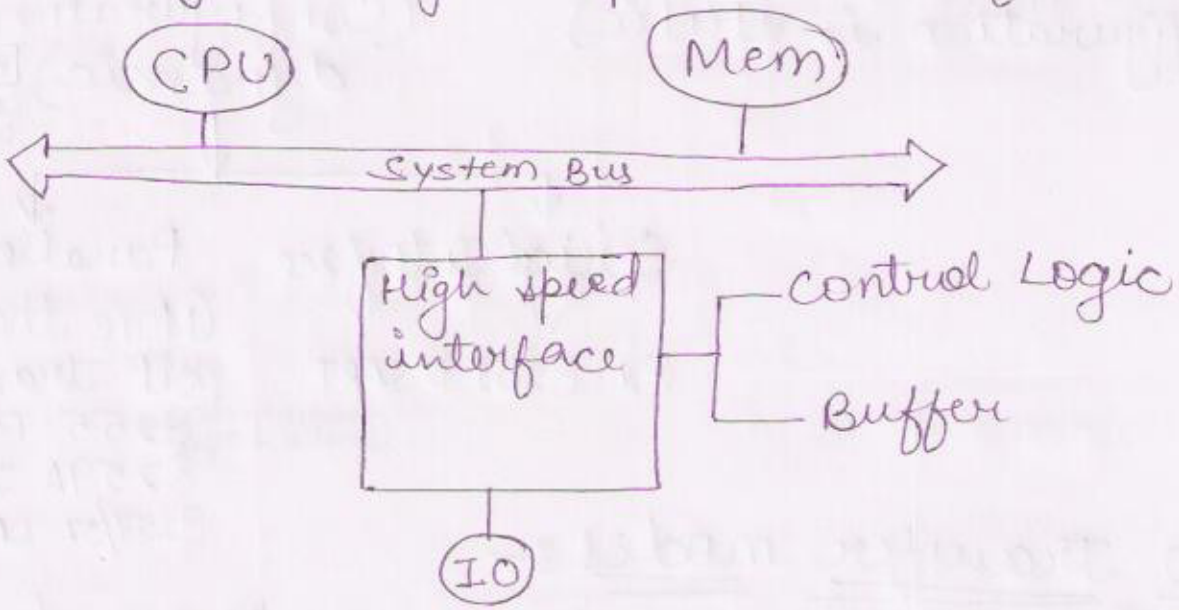
When the CPU is referring same word in same block in near future is called as temporal locality

(2) Spatial locality :-

Whe CPU refers the adjacent words in same block in near future is called as spatial locality.

IO ORGANIZATION :-

- IO devices are very slow devices. ∴ IO devices are never directly connected with system buses.
- IO devices are always communicating with CPU through high speed interface (IO module)
- High speed interface is required because IO devices are electromagnetic devices and CPU is an electronic component. ∴ there is a difference in operating modes, difference in word format, data transfer rate.
- ∴ To avoid the above problems, IO devices are connected with CPU through high speed interface.

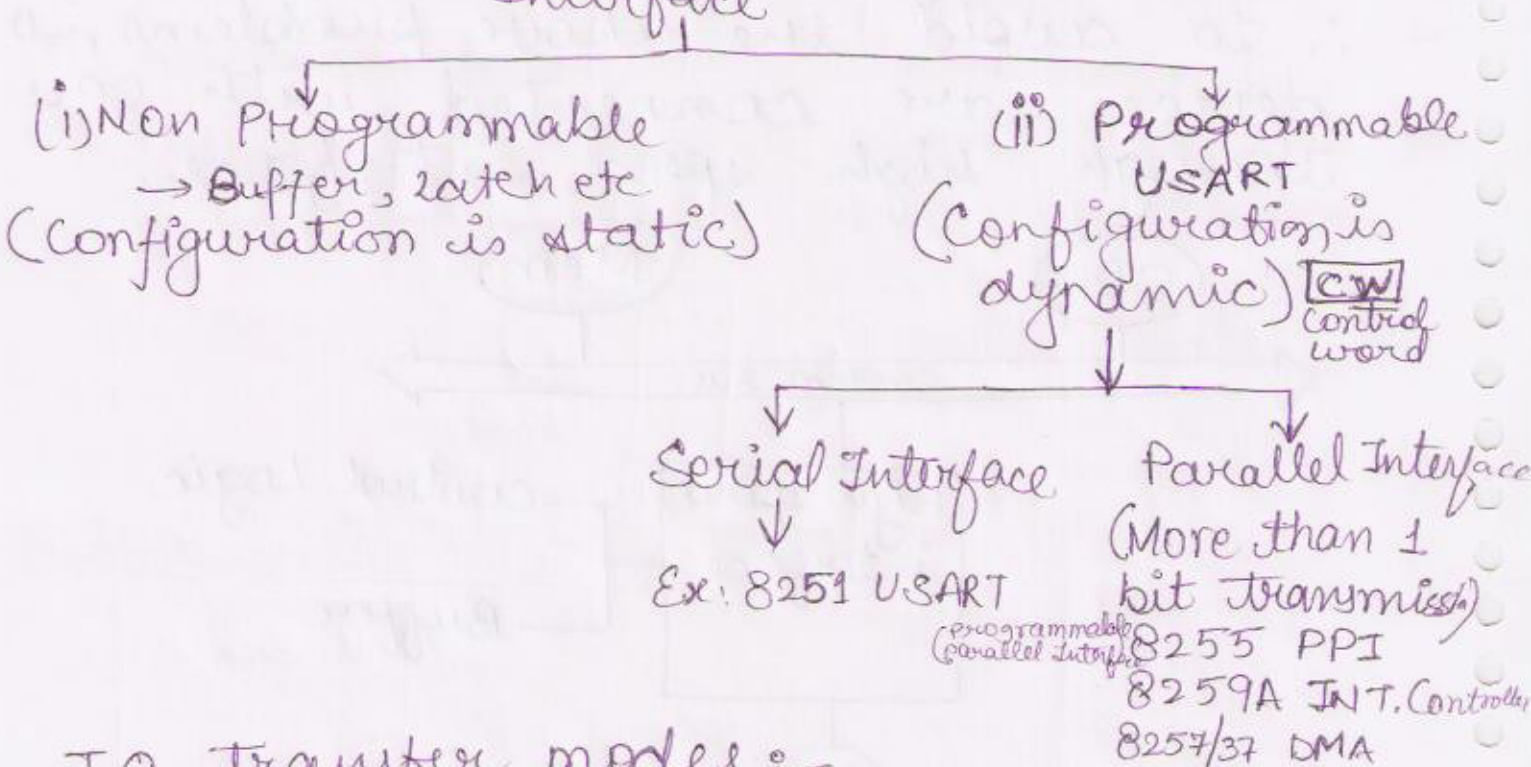


When CPU executes IO operations, it generates IO requests and ~~ee~~ transfers to high speed interface. Later CPU continues its own some other task. During that process, CPU periodically reads the status of interface.

- High speed ~~ee~~ interface control logic interprets the request & enables the

corresponding port for respective operation. Based on the speed of IO device, it prepares data & later transfers to buffer.
- When buffer has the data, high speed Interface generates the signals to CPU & waits for ack.
- After receiving ack, the buffer contents are placed on data lines ∴ speed gap will be minimised.

Types of interfaces :-

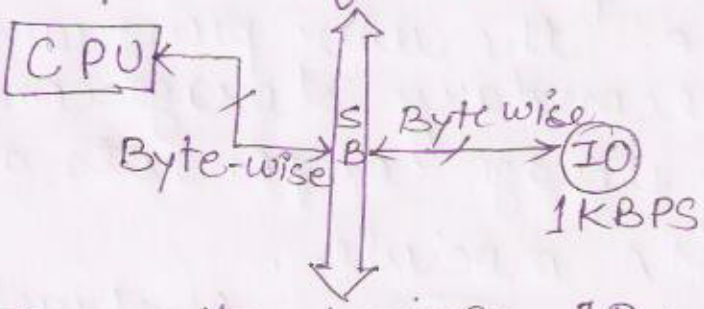


IO Transfer modes :-

There are 3 different ways present to transfer data from IO to CPU/memory.

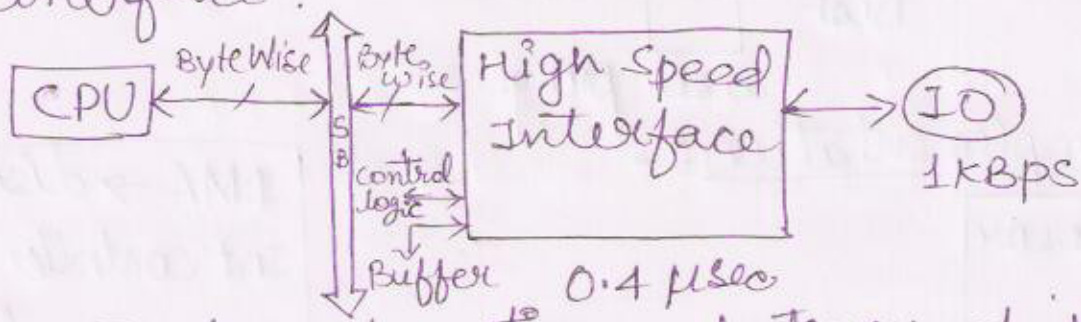
(i) Programmed IO
(ii) Interrupt driven IO
(iii) DMA.

Programmed I/O:- In this mode high speed interface is not present. The CPU is directly connected with IO devices.
∴ Processor utilization is inefficient.
This means that processor waits until completion of IO operation. This wait depends on speed of IO device.



1KB → 1 sec
1B → 1 millisec
Byte transfer time ≥ 1 msec.

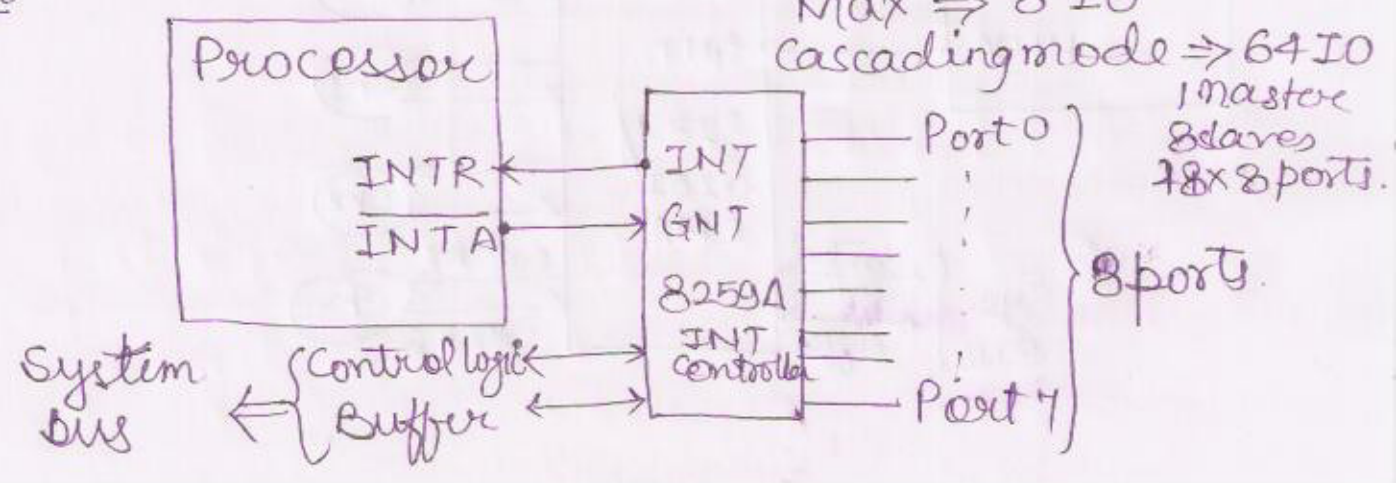
Interrupt driven IO:- In this mode, high speed interface is used to connect the various basic IO devices.
∴ Processor utilization is efficient because processor only communicates with high speed interface.
- So CPU time depends on latency of interface.



Byte transfer time = latency of interface
= 0.4 μsec.

Eg:-

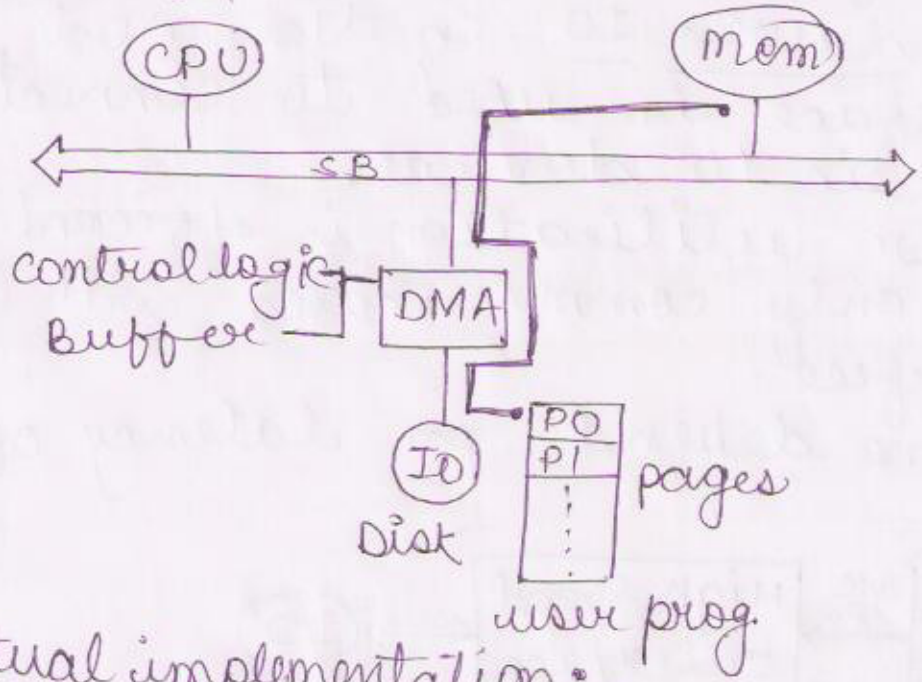
Max ⇒ 8 IO
Cascading mode ⇒ 64 IO
1 master
8 slaves
1+8×8 ports.



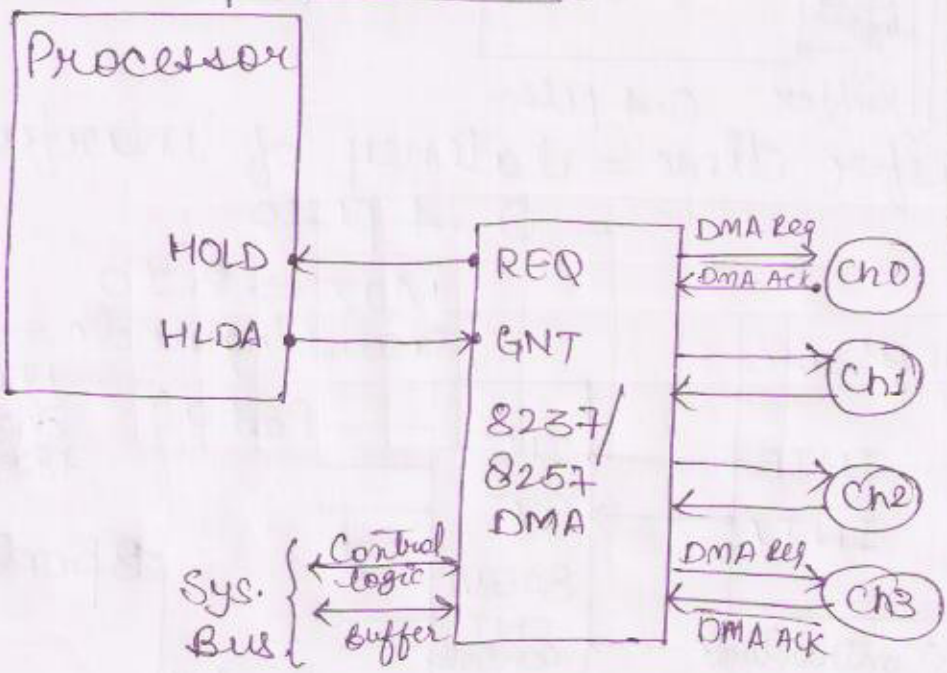
Direct Memory Access: In this mode, bulk amount of data is transferred from IO device to main memory without involvement of CPU.

When user program capacity is greater than main memory capacity, CPU uses virtual memory concept to increase the address space, i.e. the user program is stored into the secondary storage component. All the secondary storage components are connected to DMA module.

Abstract view:-



Actual implementation:-



DMA → disk
Int controller → KB, Mouse, Display etc

The CPU initialises DMA along with IO address, memory address, control signal & count value. Later CPU is busy with other task. DMA control logic interprets the request & enables the respective port for resp. oper.
- Based on the speed of disk, it prepares the data. Later it transfers the data to buffer.
- When buffer contains data, DMA enables the hold signal to gain the control of the bus and wait for HLDA signal.
- After receiving the HLDA signal, DMA transfers data from IO to main memory until count becomes zero
- After the operation, DMA reestablishes bus connections to CPU.
- In DMA oprn, CPU is present in 2 states (i) Busy state (ii) Blocked (HOLD) state. Until preparing the data, the CPU is busy and until transfering the data the CPU is blocked.
- Let x be the preparation time and y be the transfer time. Then % of time CPU is busy = $\frac{x}{x+y} * 100\%$

% of CPU blocked = $\left(\frac{y}{x+y}\right) * 100\%$

Preparation time depends on disk speed & transfer time depends on MM latency.
DMA is operating in the 3 different modes.
(i) Burst mode
(ii) cycle stealing mode
(iii) block mode

In burst mode of DMA, after receiving HLDA signal, bulk amount of data is transferred from I/O device to main memory.

In cycle stealing mode of DMA, before receiving HLDA signal, it forcibly suspend the CPU operation and gain the control of bus & transfers very important data from I/O to MM.

In block mode of DMA, after receiving HLDA signal, the data is transferred from I/O to main memory in blocks.

Pg. 18. Q. 8. Branch Penalty = 3-1 = 2 cycles

Branch frequency = 20%

Average Ino Exe. Time = $(1 + \#\text{stall/ino}) \times \text{cycle time}$

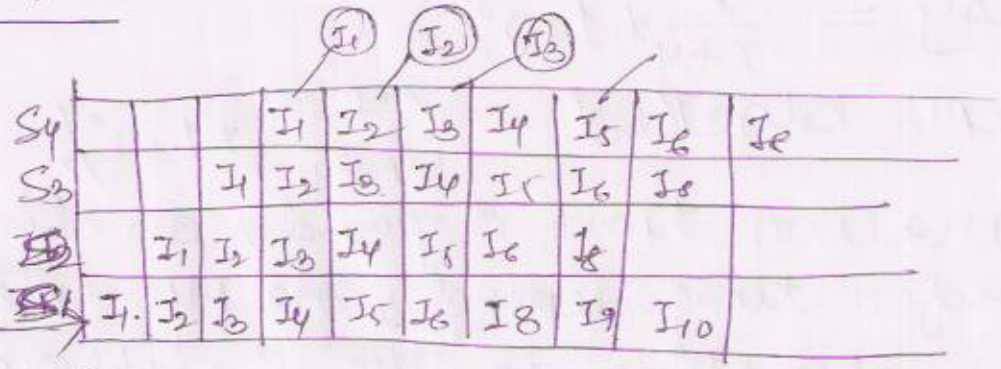
$= (1 + (0.2 \times 2)) \frac{1}{1\text{GHz}}$

$= 1.4$ ns.

Prog. Exe. Time $= 10^9 \times 1.4$ ns

$= 1.4$ sec.

Pg 19, Q 13.



Q 15. $S = \frac{\text{Avg ET}_{np}}{\text{Avg ET}_{p}} = \frac{\Sigma(IC_i * CPI_i) \text{ cycle time}}{CPI * \text{cycle time}}$

$= \frac{((0.4*6) + (0.2*4) + (0.4*5)) * 1\text{ns}}{1 * (\text{Skew time overhead})}$

$$= \frac{4.4\,ns}{1*(1+0.2)\,ns} = 3.7$$

Q16. $\frac{\text{needed}}{\text{given}} = \frac{2 \times 2^{20} \times 2^5}{2^9 \times 2^{10} \times 2^3} = 2^4 = 16.$

Q19. $\frac{250 \times 10^9 \text{ Flops}}{100 \times 10^6 \text{ Flops}} = 2500 \text{ sec.}$

Q24. Avg. Ins^n ET$_{pipe}$ = (1 + #stalls/Instn) cycletime

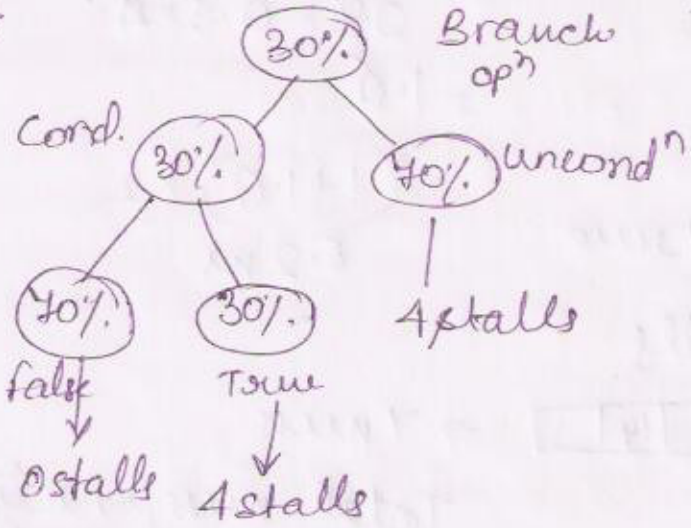
$$= \left[1 + \left(\frac{\%\text{ of freq}}{100} * (\text{penalty})\right)\right] 20\,ns.$$

$$= \left[1 + \left(\frac{30}{100} * (5-1)\right)\right] 20\,ns.$$

$$= 24\,ns.$$

25. $S = \frac{\text{pipeline depth}}{1 + \#\text{ stalls/instn}} = \frac{5}{2.2} = 2.27.$

26.



stalls = (0.3 * 0.3 * 0.7 * 0) + (0.3 * 0.3 * 0.3 * 4) + (0.3 * 0.7 * 4)

$$= 0 + 0.027 * 4 + 0.21 * 4$$

$$= 0.108 + 0.84$$

$$= 0.948$$

Avg Ins^n ET = (1 + #stalls/Ins^n) cycle time

$$= (1 + 0.948) \times 20$$

$$= 38.96\,ns.$$

30.

True		Anti		Output
In	out	out	in	out-out
$I_2 - I_1$		$I_3 - I_2$		3-1
$I_3 - I_2$		4-3		4-2
$I_4 - I_3$		2		2
$I_5 - I_4$				
4				

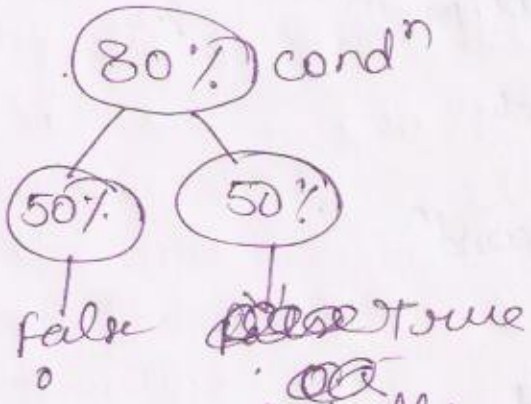
32. $t_n = 12$
$t_p = 6$
$S = \frac{t_n}{t_p} = \frac{2}{=}$

36.

	IF	ID	OF	PO	WB
I_0	1	1	1	3	1
I_1	1	1	1	6	1
I_2	1	1	1	1	1
I_3	1	1	1	1	1

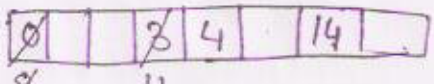
once the stage is over, op is available.
Do not wait for completion of execution till WB.

38.



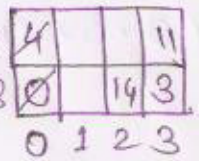
$0.8 \times 0.5 \times 4$
$= 1.6$
$(1+1.6) \times 2$
5.2 ns.

Ag 23. Q2.



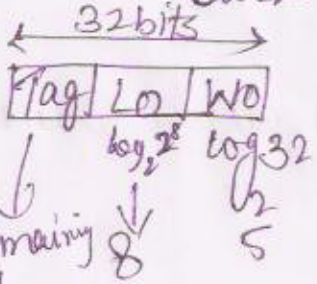
⇒ 7 miss

Total miss penalty
$= 7 * 8 = 56$.



⇒ 1 miss = 10 miss penalty

Q3. #lines = $\frac{\text{CM size}}{\text{Block size}} = \frac{8K}{32} = 2^8$



tag mem size = $(19+1+1) * 2^8$
$= 2^8 \times 21$ bits
$= 5376$ bits

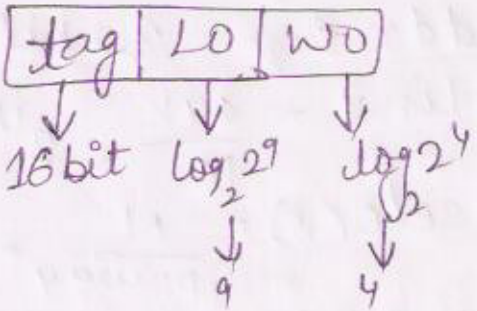
chapter-4

24. $S_A = K/1.2$, $S_B = K/1$

$S_A/S_B = \frac{1}{1.2} = 0.833$

chapter 5

Q. 6.



tagmemsize = #lines × #tag bits
= $2^9 \times 16$ bits
= 2^{13} bits

(7) ⇒ H = 0.7

$S = \frac{Tm}{Tc}$ ∴ $\boxed{Tm = 9Tc}$ Tavg = 80 ns

$Tavg = H \cdot Tc + (1-H)(Tm + Tc)$

$\Rightarrow Tavg_{new} = H \cdot Tc + (1-H)(Tm+Tc)$

80 ns = $0.7Tc + (1-0.7)(9Tc + Tc)$

$= 0.7Tc + 3Tc$

$112 = H(21.6) + (1-H)(21.6 + 194.4)$ 80 ns $3.7Tc$

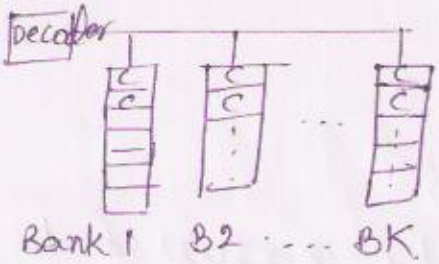
$Tc = \frac{80}{3.7} = 21.6$, $Tm = 194.4$

$112 = H(21.6) + (2160) - 216H$

$-104 = (194.4)H$

$H = 0.53$

11. decoder



One ittration generates = KC Bytes

⇓

Decoder latency * Bank latency = KC Bytes

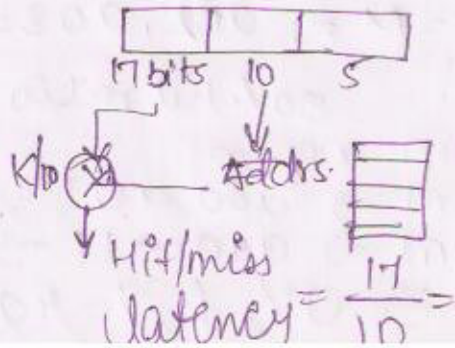
$\frac{K}{2} + 80\text{ ns} = KC$ Bytes

$K = 24$ $\quad = 48$

∴ Time = 92 ns.

To access cache block, 2 iterat's are required ⇒ 2×92 = 184 ns.

12.



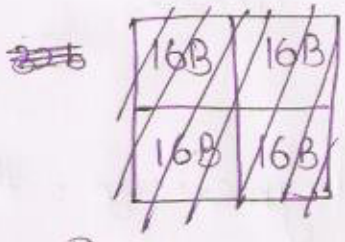
latency = $\frac{17}{10} = 1.7$ ns.

13. Comp + Mux latency latency

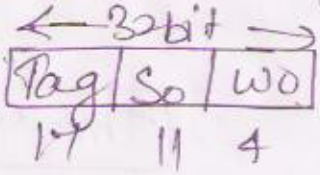
$= 0.6 + \frac{18}{10}$

$= 2.4$ ns.

Q20.



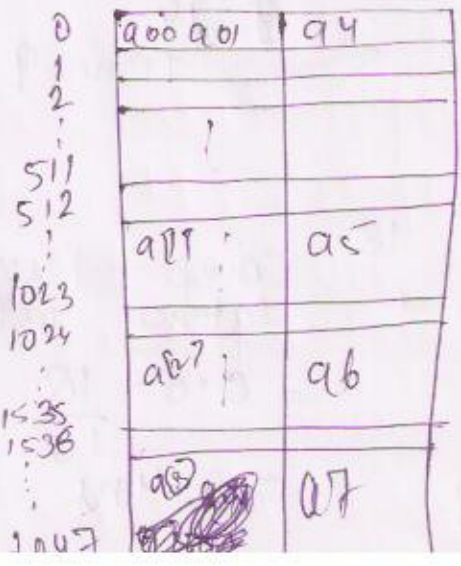
2 way set associative.
CM size = 64 KB
Block size = 16B
Addr Size = 32 bit
lines = $\frac{64K}{16} = 2^{12}$
sets (S) = $\frac{N}{p\text{-way}} = \frac{2^{12}}{2} = 2^{11}$
= 2048.



Tag Mem size
= S × # tag bits
= $2^{11} \times 2 \times 17$
= 68K bits

NOTE 6- Array is an ordered set of homogeneous data elements. Array is always stored in row major order in sequential memory locations. Array elements can be accessed in 2 ways:-
① row major ② Column major.
When the main memory block references are sequential, then arrangement in cache memory is also sequential.

→ One element size = 8B
One Block size = 16B
∴ one block holds 2 elements
one row ⇒ 1024 ⇒ 512 Blocks required to carry 1 row.



Row major:-
a(06) = M a00, a01 ⇒ CM
a01 = H
a02 = M ⇒ a02, a03 ⇒ CM
⋮
50% hit ratio
Column major
a00 ⇒ m ⇒ a00, a01 ⇒ CM
a10 ⇒ m ⇒ a10, a11 ⇒ CM
⇒ 0% hit ratio.

14, 15, 16 :-

Cache ⇒ 32KB

Block size = 128B.

$\#lines = \frac{32K}{128} = 2^8 = 256.$

Element size = 8B.

Each block has $\frac{128}{8} = 16$ elements

Blocks req/row = $\frac{512}{16} = 32$.

Row major ⇒

a00 = M ⇒ a00 → a015 ⇒ CM

a01 → a015 ⇒ H.

∴ No. of Miss per row ⇒ 32

To access complete array = 512 × 32 miss

$= 2^9 \times 2^5 = 2^{14}$

$= 16K$

Column major ⇒

a00 → M ⇒ a00 → a015 ⇒ CM } 512 Miss for 1 column

a10 M ⇒ a10 → a1,15 ⇒ CM }

∴ TMiss = 512 × 512 = 2^{18}

∴ $M_1/M_2 = \frac{1}{16}$

17 #lines = 32, Block size = 64B

Array size = 50 × 50 bytes = 2500 Bytes

Blocks req to store array = $\frac{2500}{64} = 39.04$

= 40 (B_0 - B_{39})

Starting addr: 1100H

Direct map:

tag	LO	WO
5	5	6
00010	00100	000000

Line no ⇒ 4

40 miss + 16 miss = 56 miss.

25 ⇒ $1 + \frac{40}{1000} \times \left(10 + \frac{20}{40} * 100\right)$

$= 3.4$ cycles.

26 ⇒ 1.5M.ref ⟶ 1 Ins^n

100 dom ref ⇒ ? #ins.

$= 667$.

Avg mem stall cycles/ins^n = #miss in 4/Ins^n $* H_{L_2}$

+ #misses in L_2/ins^n $* M_{L_2}$

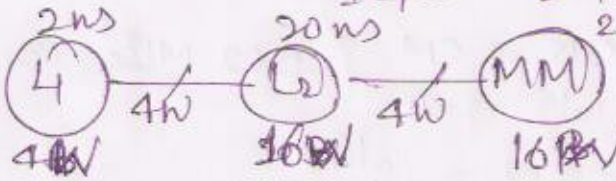
$= \frac{40}{667} \times 10 + \frac{20}{667} * 100$

$= 3.6$

27:- 1GB — 1sec

64B ⇒ ? $\frac{64B}{1GB} = \frac{2^6}{2^{30}}$ 1GB $= 64ns + 32 = 96ns$.

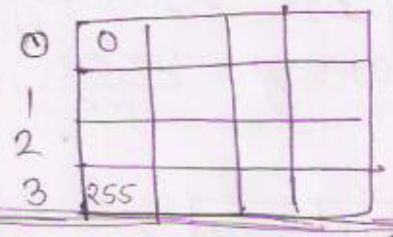
28.



Ans = $2 + \frac{16}{4}(20+2) = 88ns$.

29 ⇒ 9

30.



34. #lines = 512

Blocksize = 32B

direct map.

Addr. FBFC (16bit)

2	9	5
1111	1011 111	11100

1BF

35 ⇒ $S = \frac{512}{8} = 26$

5	6	5
1111	1011 111	11100

1F

Lockup free Cache is also called as miss-information Status handling register. It is used to store outstanding miss-information

40 ⇒ Simultaneous.

Hit ratio, write = 1

$T_m = 100$ ns

$S = \frac{T_m}{T_c}$

$T_c = 10$ ns.

write through.

$H_r = 0.92$

$H_w = 1$ (simultaneous memory org)

$f_r = 85\%$.

$f_w = 15\%$.

$T_{avg_{read}} = H_r T_c + (1 - H_r) T_m$

$= 0.92 * 10 + (1 - 0.92) 100$

$= 17.2$ ns

$T_{avg_{write}} = 100$ ns.

$T_{avg} = f_r * T_{avg_r} + f_w * T_{avg_w}$

$= 29.62$ ns.

42. - $H_r = 0.8$

$H_w = 0.9$ (Hierarchical)

Block size = 2w

$f_w = 30\%$. (dirty bit)

$f_r = 70\%$ (read bit)

$T_c = 20$ ns.

$T_m = 100$ ns/w.

$= 200$ ns/Block

$T_w = \max$ (word updation in cache, word updation in MM)

$= \max(20, 100)$

$= 100$ ns.

$T_{avg_r} = H_r T_c + (1 - H_r)(T_m + T_c)$

$= 0.8 * 20 + (1 - 0.8)(200 + 20)$

$= 16 + 44.0$

$= 60$ ns

$T_{avg_w} = H_w T_w + (1 - H_w)(T_m + T_w)$

$= 0.9 * 100 + (1 - 0.9)(200 + 100)$

$= 120$ ns

$T_{avg} = f_r * T_{avg_r} + f_w T_{avg_w}$

$= 78$ ns

$\text{Bw}_T = \frac{1}{T_{avg}} = 12.8$ million words/sec.

44. $T_{avg_r} = H_r \cdot T_c + (1 - H_r)$ [% dirty bit $(T_m + T_m + T_c)$ + % clean bit $(T_m + T_c)$]

$= 72$ ns.

$Targ_w = Hw \cdot T_c + (1 - Hw)[\%\ dirty\ bit\ (T_m + T_m + T_c) + \%\ clean\ (T_m + T_c)]$

$= 46 ns.$

$Targ = f_r \times Targ_r + f_w * Targ_w$

$= 49.4 + 13.8$

$= 63.2$

$\eta_{WB} = \frac{1}{Targ_\infty} = 15.5$ million words