Date

Textbook :- Morris Mano, William Stallings
Microprocessor :- A.K.Ray
Advanced CA :- Hennessy

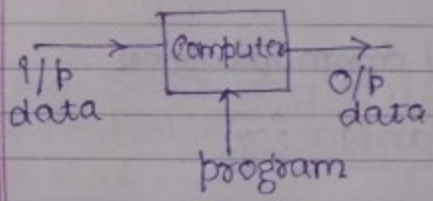
Date
Page No.

4.04.12

COMPUTER ORGANISATION

Syllabus:-
1. Functionality of Computer
2. Component of Computer

Computer :- It converts one form of data into another form under the control of program.
(Execution of program)

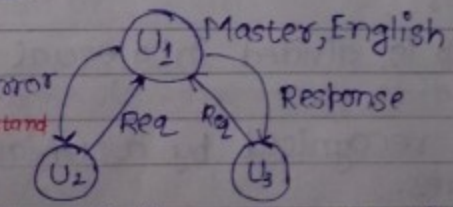


While execution of program, one form of data is converted into another form. System functionality is execution of program.

Program :- Sequence of instructions along with data.
Program → [instruction, data]

Instruction :-

There is an error because master doesn't understand hindi



To get a response from the master, we need to give instructions & data in Master understable form.

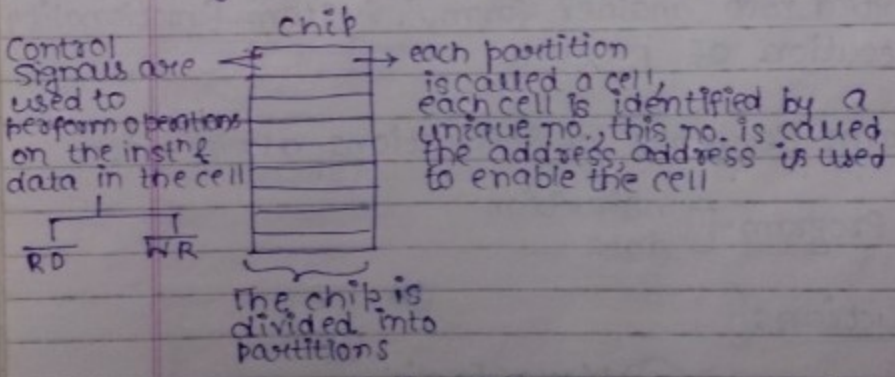
Date
Page No.

Instruction:- Binary sequence which is designed inside the processor to perform some task.
Data:- Binary sequence which is associated with a value based on a weighing factor.

To execute a program we need 3 components:-
1. CPU :- processing
2. Memory :- storage
3. I/O Devices :- external communication

Memory Organisation

Program is stored in the memory.



- Memory chip is divided into equal parts called as cell.
- Each cell is recognised by a unique no. called address.
- Address enable the memory cell.
- Each cell recognises two control signals, i.e. $\overline{RD}$ (read) & $\overline{WR}$ (write).

- When CPU wants to process the instructions, it generates a memory request to red 'read the instructions from Memory.

Mem. Request :-

[Address][CS] → Control Signal (Read or Write)

Address: where the instruction is present in memory

CS: what type of operation we need to perform with the instruction

* For performing an operation, we need address & control signal both.

- Memory Address depends on the memory chip configuration.

Memory Configuration

Memory chip configuration is represented using following format :-

64k X 8

64k X 16

64K X 32, & so on.

Consider 64k X 8 memory chip

(64k) X 8

64k → capacity of the memory

based on capacity, we can define no. of mem. cells.

64 $2^6 \times 2^{10} = 2^{16}$ cells

Mem. space :- 0000H to FFFFH (Hexadecimal representation)

$(0000)_{16}$ to $(FFFF)_{16}$

based on no. of bits req., we determine Mem. map/Address space / Address range.

∴ Mem. map: 0^{16} (16 0's) to 1^{16} (16 1's).

based on no. of mem. cells, we can calculate no. of address bits req. of enable a cell :- 16 bits (which is the power), no. of address lines req. = 16 bits

64 K x (8)

it represents cell size

- Byte Addressable Mem.
- Word Addressable Mem.

BYTE ADDRESS V/S WORD ADDRESS

- Byte Addressable Memory :-
  Each memory cell points to an 8-bit info.
- Word Addressable Memory :-
  Each memory cell points one word info.

Processor	Byte (B)	Word (W)
8-bit microprocessor	8 bits	8 bits
16-bit "	8 bits	16 bits
32-bit "	8 bits	32 bits
n-bit "	8 bits	n bits

Imp → Word size depends on the word length of the processor, i.e. no. of bits processed by the processor at a time.

- Existing Mem. system requires Byte Addressable Memory because it is unambigous.

NOTE:- The mem. system's default address space is byte addressable space.

64 KB → 64 kilo byte

Memory system is represented by bytes, & not by words.

★ Word is ambigous, because it is diff. for diff processors.

Date: Kitty
Page No.

NOTE:- When processor word size is greater than the memory cell size, then there is need for accessing multiple memory cells.

e.g.
16 bit processor :-
MOV AX, [2000] → Memory Address
↓ mnemonic (user understable format) → destination. → AX represents register
converted into machine understable format, (understandable)
i.e. opcode
A opcode indicates the type of operation, th↓ MOV is data transfer operation.

size of register = 16 bits
[Register size is the value of n in n-bit processor]

[2000] → source
↓
Memory Address
↓
8 bits
[Byte Addressable Memory]

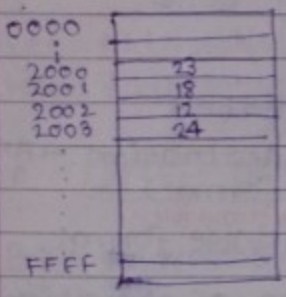
* To transfer the mem. content, we need to transfer 16 bit data, so we need to access 2 memory cells.
M[2000H]
& M[2001H]

MOV R0, [2000] [32 bit microprocessor]
↓
32 bit register

[2000H] represent only 8 bit data, so we need to access 4 cells for the transfer.

R0 ← M[2000H]
M[2001H]
M[2002H]
M[2003H]

Memory:-



★ MOV AX, [2000H]

AX = 18 23
2001 contains higher byte
2000 contains lower byte

or

AX = 23 18
2001 contains lower byte.
2000 contains higher byte

★ R0 = 24 12 18 23 or 23 18 12 24

We dont know whether 2000H contains lower byte or higher byte.
As there is no order of data storage in the memory, the above instructions generates multiple outputs(ambiguity). To avoid ambiguity there is a need of memory address interpretation mechanism (Endianess Mechanism)
Endian-ness shows the order of data storage.

Little Endian Big Endian

→ In Little Endian, the order of data storage is that lower address contains lower byte & higher address contains higher byte, i.e.

4	3	2	1	0	Address
4	3	2	1	0	Byte

∴ acc. to little endian
AX = 18 23

→ In big endian, lower address contains higher byte & higher address contains lower byte, i.e.

4	3	2	1	0	Add.
0	1	2	3	4	Byte

∴ acc to big endian
AX = 23 18

Note:- Little endian mechanism is a default interpretation Mechanism.

RO = 24 12 18 23 (little endian)

AX: | 18 | 23 | (2001 2000) (little endian)

RO: | 24 | 12 | 18 | 23 | (2003 2002 2001 2000)

RO = 23 18 12 24 (big endian)

AX: | 23 | 18 | (2001 2000) (big endian)

RO: | 23 | 18 | 12 | 24 | (2003 2002 2001 2000)

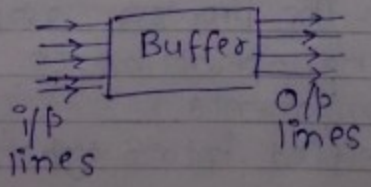
• Control Signals :- The processor supports h/w pins to perform the operations to the externally connected components.
The pins are classified into 3 types :-

① Active low pin
These pins are enabled when input is zero ('0').
They are denoted with "Pinname", e.g.
$\overline{RD}$, $\overline{WR}$, $\overline{INTA}$, $\overline{CS}$

② Active High Pin :-
These pins are enabled when their input is 1, they are denotes as "Pinname", e.g.
ALE, INTR, HOLD, HLDA.

③ Time Multiplexed Pins :-
These pins carry two meanings, whereas the meaning is defined by another pin The advantage is that it reduces the no. of pins.
e.g. AD_0-AD_7 (8085)
Address/Data
The meaning of this pin is defined by ALE. When ALE is '1', AD_0-AD_7 carries the address & when ALE is '0', AD_0-AD_7 carries the data, so the selection i/p's of AD_0-AD_7 is ALE.

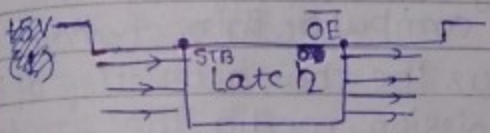
Buffer :- It is the temporary storage where the i/p & o/p lines are always in the enable state, there is no locking.
ie. when there is data in i/p lines, then immediately the data is transffered to the buffer, & when there is data in the buffer, the data is transferred immediately to the o/p lines, we can't lock data in buffer.



Latch :- Latch is also a temporary storage device, where the i/p & o/p lines are under the control of strobe & output-enable pins respec. (STB) ($\overline{OE}$)

STB
- 0 → i/p lines are disabled & data can't be transferred from i/p lines to latch.
- 1 → i/p lines are enabled & data can be transferred from the i/p lines to the latch

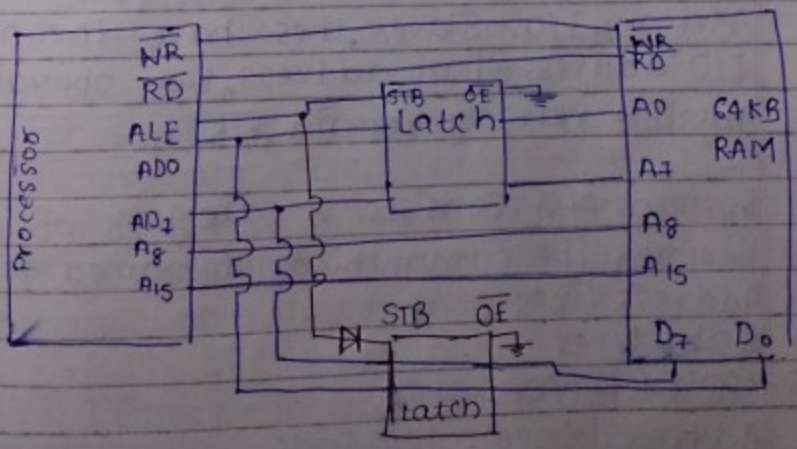
$\overline{OE}$
- 0 → o/p lines are enabled, & data can be transferred from latch to the output lines
- 1 → o/p lines are disabled, & data can't be transferred from latch to the o/p lines.



* Latch may act as a buffer, but buffer can't act as a latch.

Latch will act as a buffer when STB is high & $\overline{OE}$ is low.

Memory Interfacing :-



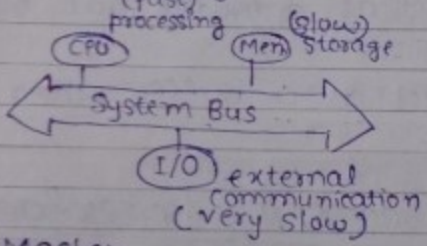
CPU : 2000H & $\overline{RD}$

A_{15}-A_8 A_7-A_0

	T_1	T_2	T_3	T_4
ALE	1	0	0	0
$\overline{RD}$	1	0		

Communication Channel i.e. Bus:-

There is a need of communication channel b/w the major components of the system, i.e. CPU, memory & I/O to satisfy the objective of the computer. This channel is called System Bus. The characteristic of Bus is shared transmission media. The limitation of the bus is only one transmission at a time



CPU:- Master

Memory, I/O:- Slaves, these have the addresses I/O address = port address, two operations are possible namely $\overline{RD}$, $\overline{WR}$.

★ System bus consists 3 category of lines. to perform the communication named as:-

1. Address Lines
2. Data Lines
3. Control Lines.

(unidirectional) Address Lines:- towards mem. & I/O, they are unidirectional, based on the no. of address lines we can determine the capacity of the system.

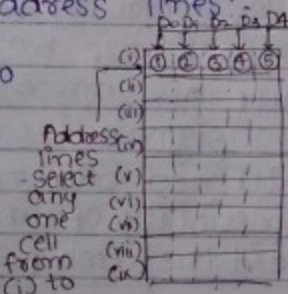
Date: Kitty
Page No.

e.g. in 8085
$AD_0 - AD_7$ $A_8 - A_{15}$ = 16 address lines
$\Rightarrow 2^{16}$ cells
$\Rightarrow$ 64K cells
$\Rightarrow$ 64KB

in 8086
$AD_0 - AD_{15}$ $A_{16} - A_{19}$ = 20 address lines
$\Rightarrow 2^{20}$ cells
$\Rightarrow 2^{10} * 2^{10}$ cells
$\Rightarrow$ 1K * 1K cells
$\Rightarrow$ 1M cells
$\Rightarrow$ 1MB

e.g. D_0 carries ①, D_1 carries ②, D_2 ,, ③, D_3 ,, ④, D_4 ,, ⑤.

a system with 32 address lines:
$\Rightarrow 2^{32}$ cells
$\Rightarrow 2^2 * 2^{10} * 2^{10} * 2^{10}$
$\Rightarrow 4 * 1K * 1K * 1K$
$\Rightarrow 4 * 1M * 1K$
$\Rightarrow 4 * 1G$
$\Rightarrow$ 4G cells
$\Rightarrow$ 4GB



Data lines carry the data stored in the selected cell (from (i) to (ix)), each data line carry one bit from the selected cell. So, no. of data lines specified the "WORD" length & not "BYTE" length)

(Bidirectional)
★ Data Lines :- These lines carry the binary sequence b/w CPU, memory & I/O, hence they are bidirectional. Based on the no. of data lines, the processor's word length depends. Based on word length, performance will be measured.

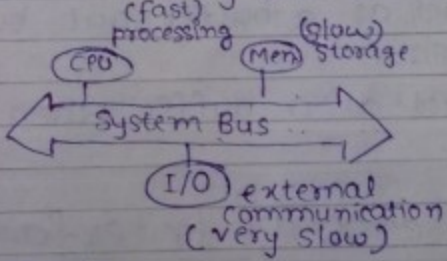
(Unidirectional)
★ Control Lines :- These lines carries the control & timing signals. Control signals are used to indicate the type of operation & timing signals are used to synchronise the mem. & I/O operations with processor clock.

Imp

	T_1	T_2	T_3	T_4
E	1	0	0	0
D	0	1	0	

Communication Channel i.e. Bus:-

There is a need of communication channel b/w the major components of the system, i.e. CPU, memory & I/O to satisfy the objective of the computer. This channel is called System Bus. The characteristic of Bus is shared transmission media. The limitation of the bus is only one transmission at a time.



CPU:- Master

Memory, I/O:- Slaves, these have the addresses I/O address = port address, two operations are possible namely RD, WR.

★ System bus consists 3 category of lines. to perform the communication named as -

1. Address Lines
2. Data Lines
3. Control Lines.

Address Lines:- towards mem. & I/O, they are unidirectional, based on the no. of address lines we can determine the capacity of the system.

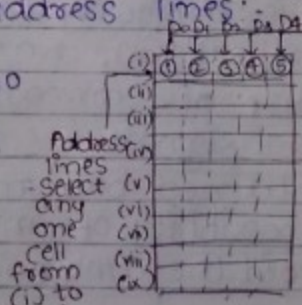
Date: Kitty
Page No.

e.g. in 8085
$AD_0 - AD_7$ $A_8 - A_{15}$ = 16 address lines
$\Rightarrow 2^{16}$ cells
$\Rightarrow$ 64K cells
$\Rightarrow$ 64KB

in 8086
$AD_0 - AD_{15}$ $A_{16} - A_{19}$ = 20 address lines
$\Rightarrow 2^{20}$ cells
$\Rightarrow 2^{10} * 2^{10}$ cells
$\Rightarrow$ 1K * 1K cells
$\Rightarrow$ 1M cells
$\Rightarrow$ 1MB

e.g. D_0 carries ①, D_1 carries ③, D_2 " ④, D_3 " ⑤, D_4

a system with 32 address lines:
$\Rightarrow 2^{32}$ cells
$\Rightarrow 2^2 * 2^{10} * 2^{10} * 2^{10}$
$\Rightarrow 4 * 1K * 1K * 1K$
$\Rightarrow 4 * 1M * 1K$
$\Rightarrow 4 * 1G$
$\Rightarrow 4G$ cells
$\Rightarrow 4GB$

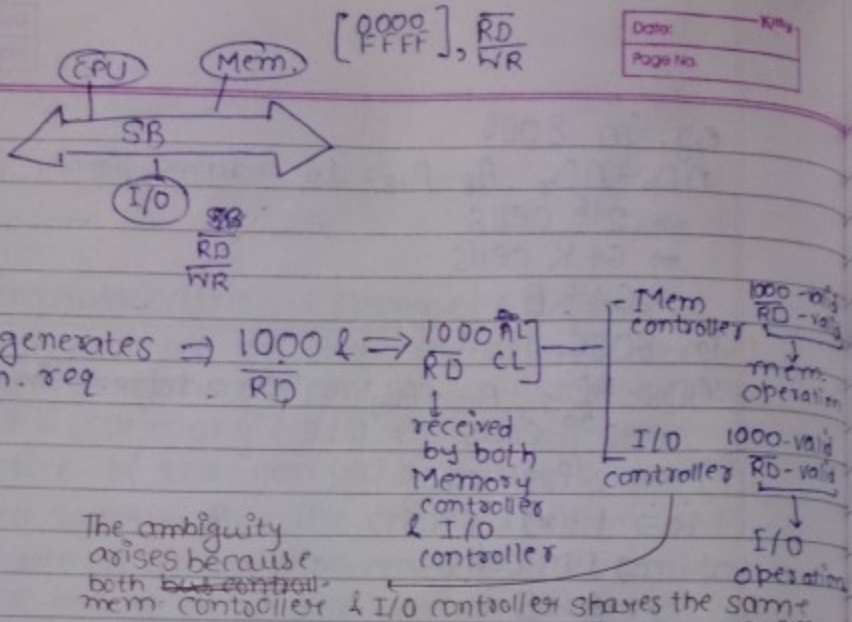


Data lines carry the data stored in the selected cell (from (i) to (ix)), each data line carry one bit from the selected cell. So, no. of data lines specified the "WORD" length (not "BYTE" length)

(Bidirectional)
★ Data Lines :- These lines carry the binary sequence b/w CPU, memory & I/O, hence they are bidirectional. Based on the no. of data lines, the processor's word length depends. Based on word length, performance will be measured.

(Unidirectional)
★ Control Lines :- These lines carries the control & timing signals. Control signals are used to indicate the type of operation & timing signals are used to synchronise the mem. & I/O operation with processor clock.

Imp



- There is ambiguity when both IO & Memory controllers both want to access the bus
- Because of common channel & common control signals & common address space maintained b/w memory & I/O, then there is a possibility of ambiguity.

To avoid the ambiguity bus configurations are used.

BUS CONFIGURATIONS

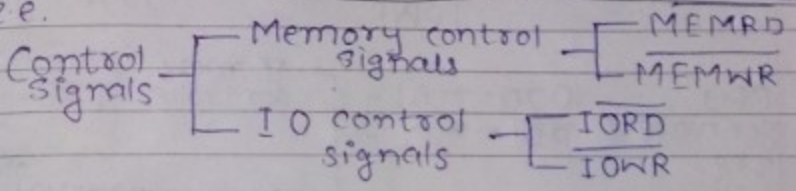
- There are 3 bus configurations used in the processor design, named as:-

→ IOP (Input-Output processor)
→ Isolated IO (IO mapped IO)
→ Memory-Mapped-IO

IOP:- In this configuration, seperate buses are used for memory & I/O, but introducing an additional bus is expensive.
Only high-performance architecture are using this configuration.

- Isolated IO :-

This configuration uses common bus & common address space but different control signals, i.e.



$\overline{MEMRD}$ is initiated by LOAD control signal.
$\overline{MEMWR}$ is initiated by Store control signal.
$\overline{IORD}$ is initiated by IN control signal.
$\overline{IOWR}$ is initiated by OUT control signal.

★ both 8085 & 8086 processors use the Isolated IO configuration that can be implemented as follows.

8086:

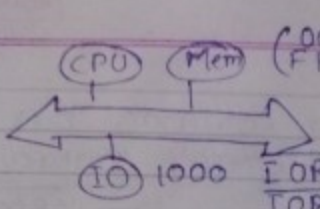
M/$\overline{IO}$ $\overline{RD}$ $\overline{WR}$

this pin selects whether to access memory or Input-Output

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	
1	0	1	→ Memory Read
1	1	0	→ Memory Write
0	0	1	→ IO read
0	1	0	→ IO write

- In 8085, we have $\overline{M}$/IO as the only difference from 8086 where it is M/$\overline{IO}$.

Imp.



where 1000 is some I/O device with I/O port address as 1000.

CPU generate req.: 1000 → AL, RD → CL

Memory controller: 1000 → valid, IORD → invalid → no operation

IO controller: 1000 → valid, IORD → valid → I/O operation.

★ So there is no ambiguity.

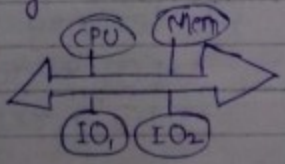
Memory-Mapped I/O :-

In this configuration, common channel & common control signals are used but unique address space is maintained b/w IO & memory.

NOTE:- The unique address is maintained by taking some of the addresses from the pool of memory address space & assigning them to the IO ports & disable the same in the memory.

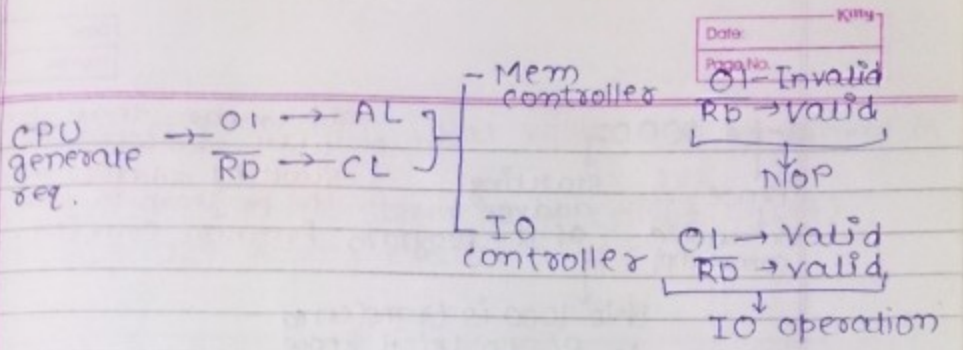
★ At the time of design, we checks how many IO devices are there, & we will take that no. of addresses from the memory space of memory.

e.g.



Imp

we know that there are 2 IO devices, so we take 2 addresses (assume address 0000 & 0001) from the pool of addresses of memory & assign these addresses to IO devices. now these 2 addresses are not assigned for memory usage.

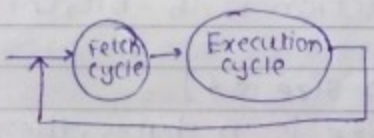


01 address is taken from the pool of addresses of memory to I/O device.

Instruction Cycles

① This concept describes the execution sequence of an instruction.

② Instruction cycle consists 2 subcycles, i.e.
(i) Fetch cycle
(ii) Execution cycle.



Fetch Cycle :-

In this cycle, the instruction is transferred from memory to CPU based on the program counter register. This process is called instruction fetch. At the end of instruction fetch, the program counter is incremented by step-size to point to next instruction address. ∴ the functionality of program counter is that it holds the starting instruc^{n} address & immediately points to the next instruction address.

e.g. if our program is stored starting from mem. location 1000, then we pass this command to let PC knows about the starting address of program.

-t = 1000
- t → trace executable command
- 1000 → starting address of the program

This means that the user transfers the starting address of the program to the Program Counter.

this 1000 is transferred to PC to let it know about the start of the program

PC = 1000 → Mem Req → Memory → this binary seq. is transferred to the CPU → instruction fetch.

Step-Size
- Constant (when CPU supports fixed length instn.)
- Variable (when CPU supports variable length instn.)

then PC ← PC + stepsize

→ step-size is the size of the instruction,
if instruction is 8-bit, then
PC ← PC + 1
& if instruction is 16-bit, then
PC ← PC + 2
[because the step-size is 4].
if instruction size is n bytes, then step-size is n [step-size is in bytes]

Q1. Consider the following program segment

Instⁿ	size (words)	PC	valid PC
I_1	2	PC = 1008	1000 → 1007
I_2	1		1008 → 1011
I_3	1		1012 → 1015
I_4	2		1015 → 1023
I_5	1		1024 - 1027

(a) assume that word size is 32 bit & the program has been loaded in the memory with starting address is 1000 (decimal) onwards

32 × 7 = 224

Date: Page No. Kitty

During execution of I_4, what value will be present in PC

Ans 32 bit will be stored in 4 cells(4 bytes).
∴ PC = 1024 during execution of I_4.

Imp.

★ During Execution of I_4, the program counter is incremented.

(b) Assume that memory is word addressable with word size 32 bit. The program has been loaded in the memory with st. address of 1000 (decimal onwards). During execution of I_3, what is PC

Ins^n	size (words)	valid PC
I_1	2	1000-1001
I_2	1	1002
I_3	1	1003
I_4	2	1004-1005
I_5	1	1006

∴ PC = 1004.

[In both (a) & (b) step-size is variable because the processor is supporting variable-length instructions.]

★ We can see that step-size is variable, because when the processor supports fixed-length instruction, then step-size is constant,
if processor supports variable-length instruction, then step-size is variable.

Q2. A computer has 24-bit instructions. The program has been loaded in the memory with st. address of 3000 (decimal onwards). Which is valid value for PC.

(i) 400 (ii) 500 (iii) 600 (iv) 700

[this is byte addressable memory & each memory cell is of 1 byte, but each instruction is of 24 bit, so to store 1 instruction we need 3 cells.]

[as step-size is 3 byte.]

The step-size is constant because the processor is supporting fixed length instruction of 24 bits.

Date: Kitty
Page No.

→ valid PC.

300	302
303	305
306	308
309	311

Execution Cycle :-

In this cycle, the fetched instruction undergoes processing.
To process the instruction, we need opcode info.
Instruction format defines the opcode.

Instruction Format :-

It gives the layout of an Instruction.

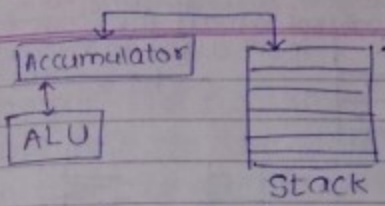
- Encoding :- N combinations requires $\log_2^N$ bits
  e.g. we want to do four operations, like add, sub, mul, div., then there are 4 operations, so we require $\log_2 4 = 2$ bits in opcode.
- Decoding :- N bit decoder generates 2^N o/p combinations
  e.g. 2 bits in opcode, then we can perform 4 operations

Instruction Format is classified into 5 types based on CPU organisation. CPU organisation is classified into 3 types based on internal storage.

① Stack Machine
② Accumulator Machine
③ General-register org^n

} → CPU organisation

Stack Machine :- In this organisation, the ALU operations are performed only on the top of the stack.
[0-Address Instruction]



In this organisation, Source & destination addresses are default addresses, i.e. instruction contain only one field, i.e. opcode.

e.g. opcode → indicates type of operation
- data transfer opn.
- ALU operations.

as data can only be transfered from CPU to TOS & vice versa, so source & destinations are default, we dont need to give source or destination address

Destination	Source	
TOS	CPU(Accumulator)	Push → Data transfer operations
CPU(Acc.)	TOS	Pop → Data transfer operations

→ ALU operations.

→ 2nd type of operation :- ALU operations [0-address instruction]

In this case we need two operands, we will get this from TOS, perform the operation & stores the result back in stack.

e.g.

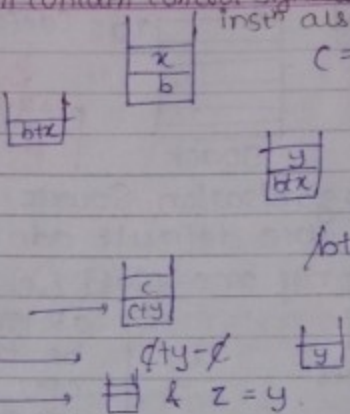
Push	Pop	Add
TOS ← Acc	Acc ← TOS	TOS ← TOS + TOS
Pop		POP, POP, then perform addition & then push the result back into the stack.

Q5. Consider following program & execute onto the stack machine. Each arithmetic operation pops the 2nd operand, pops the 1st operand, operates on them & then store the result back onto the stack.

ADD inst^n intrinsicly contain two pop & 1 push inst^n, so I thought it should be obviously larger than a push or pop inst^n, but I was wrong. & push & add inst^n do not have any dependency to each other, the most we can say that is the microprogram corresponding to add inst^n will contain control signals for push & pop inst^n also, so microprogram for add > microprogram of push/pop instn. (definitely)

Page No.

Push b
Push x
ADD
POP C
Push C
Push y
ADD
Push C
Sub
popz



Q. Which of the following statement(s) is/are true?

(i) If each push & pop instruction req 5 bytes of storage, each arithmetic inst^n req. 1 byte of storage, then the whole program req. 40 bytes of storage. (38 Bytes)

(ii) At the end of execution z contain the same value as y.

(iii) At the end of the execution, stack is empty.

(a) 1 only
(b) 2 only
(c) 2,3
(d) 1,2,3

Accumulator Machine (1-address instruction format)

In this organisation, one of the ALU operand is by default present in the accumulator, other ALU operand may be present in memory or processor register. The Accumulator acts as the source as well as the destination. The instruction ~~con~~ ~~regi~~ contains:-

opcode & address of 1 register

Date
Page No.

The operand is transferred from memory or register to the temporary register (TR).

* In this organisation, the Instⁿ format consists 2 fields, i.e. opcode & address field.

| opcode | Address |

→ of the second source.

Destn	Source
Mem	CPU(Acc)
CPU(Acc)	Mem.

} for Data transfer op. [we need to transfer the address of mem. location or register address]

The destination for ALU operations is always the accumulator.

Destn	Source1	Source2
ACC	Acc	Mem/Reg.

→ ALU operation

e.g.

→ Load [3000]
Acc ← M[3000]

→ ADD [5000]
Acc ← Acc + M[5000]

→ Store [5000]
M[5000] ← Acc.

→ ADD R0
Acc ← Acc + R0

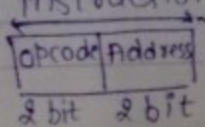
Note:- One Address Machine also supports Zero Address Machine if there is a free space after allocating One Address instruction.

this means that if some free space will be left, we can assign it to stack & hence can work it as a stack machine

Q4. Consider a hypothetical processor which has 4-bit instruction & 2 bit addresses. It supports both 1-address & 0-address instruction. If there exist 2 One address instructions, how many Zero instructions can be formulated

Ans:- Identify the highest instruction format

take 1-address format



In the question it is given that we have 2 1-address instruction, this means that we only have 2 opcodes, these 2 opcodes requires storage of 2 bits (as per the question), rest 2 bits are for the address. Now take the opcode:-

Date: Kirty
Page No.

these 2 are free combinations [ ] → these 2 opcode combinations are used by 1-address format.

Step 2:- Identify the no. of opcode combinations.
no. of opcode combinations = 4. [$2^{\text{opcode size}}$]

Step 3:- Calculate the no. of free combinations after allocating the highest instructions.
→ no. of free combination = 4 − 2 = 2

Step 4:- Calcute the no. of low order instructions by taking the no. of free combination & decode the value of address field.

Step 2:- no. of opcode combinations = 4

00	address, → assume it is
01	address, taken by 1-address
10	address format
11	address

Now, out of these 4, 2 is taken by 1-address instruction.

Step 3:- so, no. of free opcode comb^n = 4 − 2 = 2
where free combn. = 10 Addr.
11 addr.

Now, for 0-address format.
address is 2 bits.
∴ we can have a total of 4 combinations from 2 bit address

∴	10	00		11	00
←	10	01		11	01
	10	10		11	10
	10	11		11	11

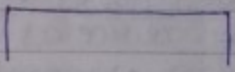
opcodes of 4-bit for 0-address instructions starting with free combination '10'.

↳ opcodes of 4 bits for 0-address instruction starting with '11'.

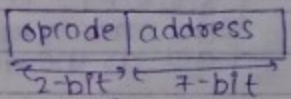
Now for 0-address instruction we have need to supply the opcode only [as we don't need to give the source & destination addresses.], so 10 & 11 are the free combinations we can use the address bits also

★ no. of zero address instn = no. of free combinations $2^{\text{no. of address bits}}$ ★

Q5. Consider a hypothetical processor, a nine-bit instruction is placed in the 128-~~bit~~ word memory, the processor supports both 1-address & 0-address instruction, if there exists ~~3 ta~~ 3 one-address instruction, how many 0-address instructions can be formulated.



128 word $\longrightarrow 2^7$ cells $\rightarrow$ 7 bit address is req.



# of combinations of opcode = 4

# of free combination of opcode = 1

∴ We have

# of 0-address instructions = $1 \times 2^7 = 128$

Q6. Consider a computer which has 32 bit instructions & 24 bit address. It supports both 0-address & 1-address inst^n.

(a) What is the range of 1-address instructions.

(b) What is the range of 0-address instructions

Ans. (a) 0^{32} to $11111110\ 1^{24}$ $0^8 0^{24}$ to $0^8 1^{24}$ $1^7 0^{24}$ to $1^8 1^{24}$

(b) $11111110\ 0^{24}$ to $1111\ 1110\ 1^{24}$ $0^8 0^{24}$ to $1^7 0^{24}$ $1^8 0^{24}$ to $1^8 1^{24}$

no. of opcode combination = $256 = 2^8$

(a) 1 address - instruction: 1^{min} to 255

(b) 0 address - instruction: $255 * 2^{24}$ to $1 * 2^{24}$.

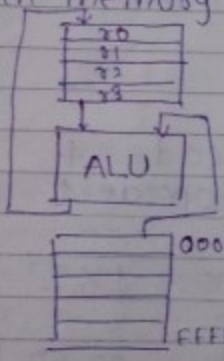
Date:
Page No.

General Register Organisation :- Based on no. of Registers supported by the processor, this architecture is divided into 2-types :-

1. Reg-Memory ref. Arch.
2. Reg-Reg ref. Arch.

Reg-Memory ref. Arch. :- [Source1 & Destination - reg
Source2 :- reg. or mem.]

This arch. supports less no. of registers. The register acts as the source as well as the destination. The 2nd operand is present in memory or register.



Source 1 :- Register the same register will act as the destination.

Source 2 :- Register or memory.

★ We are having 4 ~~addresse~~ registers here.

In this architecture the instruction format consists 2 address fields along with the opcode

| Opcode | Addr 1 | Addr 2 |

Data transfer opⁿ doesn't takes place b/w registers, i.e. R0 ← R1 (not possible) & also not b/w two mem. e.g. M[1000] ← M[2000] (not possible)

Type of opⁿ	Destⁿ	Source
1. Data transfer opⁿ.	CPU (reg)	Memory
	Memory	CPU (reg)
2. ALU opⁿ	Destn. / Source1 reg.	Source1 reg.
		Source2 reg./Mem.

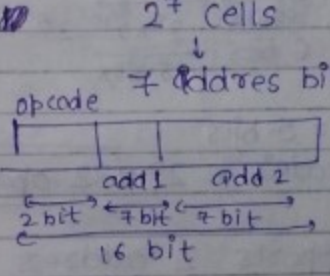
These two are same.

Date: Kitty
Page No.

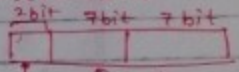
- Mov r0 [4000]
  r0 ← M[4000]
- Add r0, r1
  r0 ← r0+r1
- Mov [5000] r0
  M[5000] ← r0
- Add r0, [4000]
  r0 ← r0+M[4000]

Q. Consider a hypothetical processor which has 128 word memory. A 16-bit instruction is placed in 1 memory cell. It supports 2-address, 1-address & 0-address instructions. It uses expanding opcode technique. If there exist two 2-address inst^n & 250 1-address inst^n, how many 0-address inst^n can be formulated?

Ans.



For 2-address instructions:-



opcode address bits

as only 2 opcode combinations are used, ∴ # of free combinations for 1 address instructions are 2, now, out of rest 14 address bits, we can only use 7 bits for opcode as the other 7 bits will be used for addresses

No. of opcodes = $2^2 = 4$

No. of free combinations = $2(4-2)$

no. of 1 address instn. = $2 \times 2^7 = 256$

Now,

$N = 256$ [in 2nd stage.]

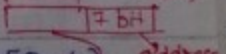
∴ ~~we have~~

now we have 250 1-address inst^n.

∴ no. of free combinations = $6\ (256-250=6)$

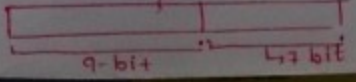
∴ no. of 0-address inst.^{n} = 6×2^7

∴ no. of 1-address inst^n that can be formed are :- $2 \times 2^7 = 2^8 = 256$,



now out of 256 opcode combinations, only 250 1-address opcode combinations are used, so no. of free combinations for 0-address instructions is 6.

so, for 0-address 6×2^7



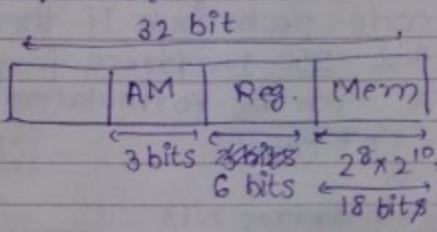
we can use only 6 combinations out of 2^9 combinations.

$2^8 \times 2^{10} = 2^{18}$

$2^8 \times 2^{10} = 2^{18}$

Q. Consider a computer which has 256K word memory. It supports an instruction format with 4-fields i.e. opcode, register addressing field (used to represent ~~one of~~ 1 of 60 processor register), addressing mode field (to represent 1 of the 7 addressing mode), Memory address field. How many no. of operations are supported when a 32 bit instruction is placed in one memory cell.

Ans.



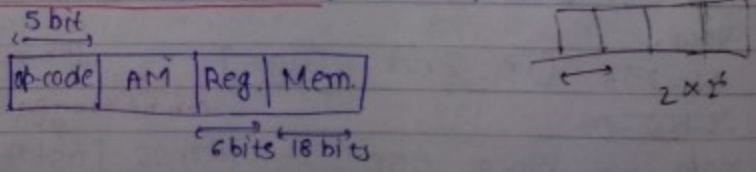
18
+3
+7
28

$32 - (18+6+3) = 5$ bits

∴ no. of opcodes $= 2^5 =$ 32 opcodes

(b) A system supports 2-address & 1-address instructions. If there exist 30 2-address instructions, how many 1-address mem. reference instructions are possible?

Ans.



# of free opcode $= 32 - 30 = 2$

2×2^6 (as it is mem. reference instructions, ∴ mem ref bits will not be taken into consideration.)

∴ 2×2^6.

Date: Kitty
Page No.

① The above instruction format contain register reference & mem. reference.

② As a part of 2-address inst^n, we can read the source1 from reg. & source2 from the memory.

③ After allocating 30 2-address inst^n, 2 combinations are free.

④ Those combinations are used to define either 1-address mem. ref. inst^n or 1-address reg. ref. inst^n

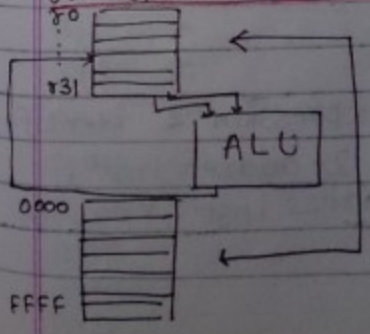
⑤ For the mem. ref. inst^n, the free opcode combinations are combined with the register address, i.e. 2×2^5

⑥ For register-ref. inst^n, the free opcodes are combined with memory address, i.e. 2×2^{18}

Register to Register Ref. Architecture

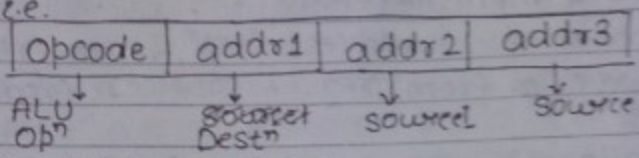
[Destination-reg. Source1-reg. Source2-reg.]

This architecture supports more no. of registers, ∴ there is no possibility of overwriting. The ALU operations are performed only on the registers. Load & Store inst^n are used to transfer the data b/w memory & registers.



★ In reg-mem ref. arch., the ALU opn are taking place b/w reg. (source 1) & reg./mem. (source 2) but in reg-reg ref. arch, the ALU opn. are taking place b/w reg. (source 1) & reg. (source 2).

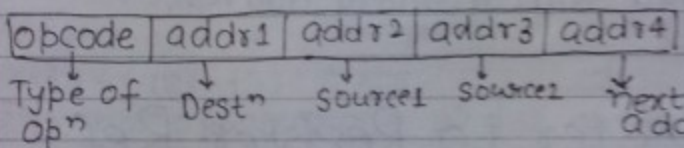
ALU supp opⁿ supports 3-address instⁿ format, i.e.



e.g. Add r0, r1, r2
r0 ← r1 + r2
Add r0, [4000], r2 → not possible
[as all must be registers.]

4-Address Instⁿ format

There is a special instⁿ format that contains 4-address field along with the opcode



★ Not implemented in program design.
Only Concept, no reality.

Note: ☆
- 8085 is an example of Accumulator Machine.
- 8086 is an example of Register to Memory ref. architecture.
- RISC is an example of Register to Register ref. architecture.

Q. Consider the following expression & identify how many 3-address instⁿ, 2-address instⁿ, 1-address instⁿ & 0-address instⁿ req. to execute the expression:

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

Where X, A, B, C, D are the mem. addresses.

Ans. Register-to-Register :-

[First load the data into registers from memory

```
Load r0, A
Load r1, B
Load r2, C
Load r3, D

Add r4, r0, r1
Add r5, r2, r3
Mul r6, r4, r5
Store X, r6
```

2-Reg.-Mem. ref. :-

```
MOV r0, A
MOV r1, B
ADD r0, B
MOV r1, C
ADD r1, D
MUL r0, r1
Store r0
MOV X, r0
```

Acc. Machine:-

```
Load A    ACC ← M[A]
Add B     ACC ← ACC+M[B]
Store T ; M[T] ← ACC
Load C  ; ACC ← M[C]
Add D   ; ACC ← ACC+M[D]
MUL T   ; ACC ← ACC*M[T]
Store X ; M[X] ← ACC
```

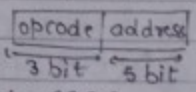
0-addr. instⁿ :-

```
Push A
Push B
ADD
Push C
Push D
ADD
Mul
Pop X
```

Date 21.04.12

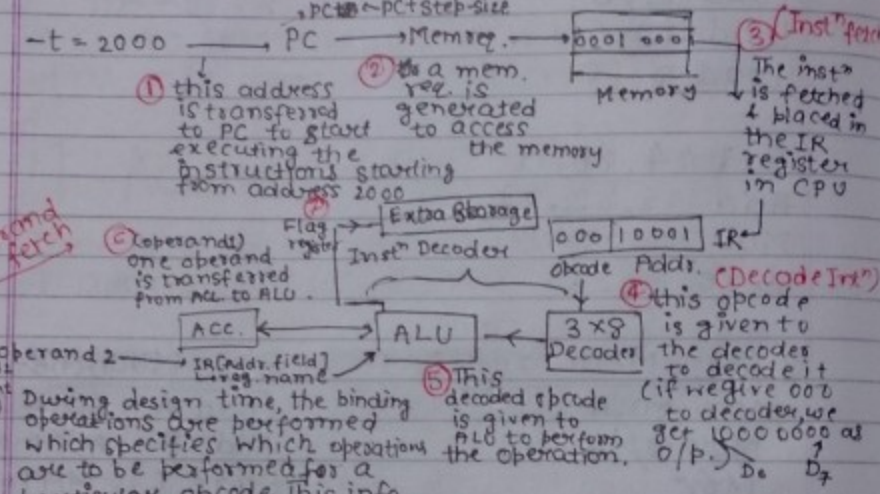
Execution Cycle = Fetch cycle :-
1. Inst^n Fetch
2. Decode Inst^n
3. Operand Fetch
4. Process data & Store result

Q1 Consider Accumulator Machine. It supports 8-bit inst^n & 3 bit opcode. ∴ inst^n format is:-



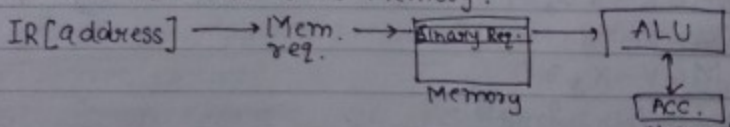
-t = 2000 → PC → Mem. Req. → Memory

↳ executable command to CPU (trace the command at address 2000).

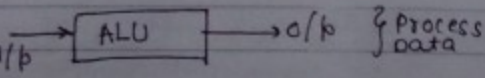


During design time, the binding operations are performed which specifies which operations are to be performed for a particular opcode. This info. is present in ROM. (that contains info. that 000 opcode when decoded to 1000 0000 should perform addition, etc.)

★★ But when operand 2 is in Memory.



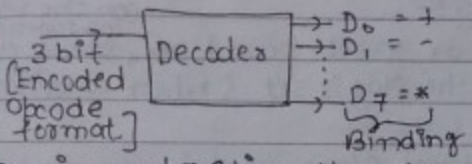
→ Now operand is fetched & inst^n was decoded earlier.



★ Execution Cycle consists of following parts:-
1. Inst^n Decode
2. Operand Fetch
3. Process Data (converting i/p to o/p.)

Execution Cycle

- Instruction Register:- It holds the currently fetched inst^n to decode because the inst^n format is predefined in this register.
The encoded opcode format is connected to the decoding circuit to identify the type of operation.
Each decoded signal is bind with a unique meaning & the info. about the decoded signal is stored in ROM during design time.



- During operation, the corresponding operation related control signals are generated based on the opcode.
- The process of enabling the ALU gates based on the opcode present in the inst^n is called as inst^n decoding.
- Acc. to accumulator machine, the 1st operand is transfered from accumulator & 2nd operand may be present in the register/memory.
- The 2nd operand address is present in the address field of inst^n register.
- The process of transferring the binary sequence, either from register or from mem. to ALU is based on IR register address field, this is called operand fetch.
- The process of converting one form of data into another form based on the enable ALU gates is called as process data.

☆ PSW (Program Storage Word) / FLAG REGISTER :-

To store the exceptional conditions of ALU, there is a need of extra storage. The extra storage is called as

flag. Flag is a flip-flop. Flip-flop is a bi-state device used to store 1-bit information

Flag → Flip-flop → [set (1), reset (0)]

Flags are classified into 2 types:-

1. Conditional Flags
2. Control Flags.

Conditional Flags:-

These flags are set or reset based on (on the result nature of ALU) some conditions. If condition is true, the flag is set & when condition is false, flag is reset.

There are 6 conditional flags used in the computer:-

1. Carry
2. Parity
3. Auxillary Carry
4. Zero Flag
5. Sign
6. Overflow

- **Carry Flag:-** If there is an extra bit after MSB

Carry flag is also set/reset during subtraction operation. In such cases it acts as a borrow flag, when there is borrow from MSB, then carry flag is set, otherwise it is reset.

True → set = 1 = C (C:- carry)

False → reset = 0 = NC (NC:- no carry)

This flag is used in unsigned arithmetic operations to represent the range exceeding conditions.

"n" bit unsigned range = 0 to $2^n - 1$

4 bit " " = 0 to 15

```
1001
0010
1011   → in range

1001
1000
10001  → out of range.
```

eg:
```
1010
-0111
0011
```
as there is no borrow for MSB, ∴ CY = 0

```
1010
-1011
1111
```
as there is borrow for MSB, ∴ CY = 1

Date:
Page No.

Parity Flag → i.e. if the result of an ALU op^n contains even no. of 1's

If the ALU o/p contains even no. of 1's

- T → set=1=PE (even parity)
- F → reset=0=PO (odd parity)

This flag is used in serial data communication.

Auxiliary Carry :- (in Packet BCD format)

"If there is an extra bit from lower nibble to higher nibble". or "3rd position to 4th position".

- T → set=1=AC (Auxiliary Carry)
- F → reset=0=NA (No Auxiliary carry).

This flag is used in the BCD arithmetic operations.

Packed BCD :- each decimal is represented by 4 bits.

Unpacked BCD :-

This format is used to represent decimal numbers. There are 2 BCD formats :-

- Unpacked BCD :- each decimal digit is represented by using 8 bits,

  e.g. 23 → 0000 0010 (2) 0000 0011 (3)

- To minimize this storage, there is packed BCD format.

  In this format, each decimal digit is represented by using 4 bits.

  e.g. 23 → 0010 (2) 0011 (3)

Case :- when no is outside the range of BCD.

In this case we add 6 because in lower nibble addition, the 13 is outside the range of BCD.

e.g.

23	0010	0011	→ 0100 1011
28	+ 0010	1000	+ 0110 (add 6)
51	0100	1011	0101 0001
	4	B	5 1

Even though auxiliary carry is 0.

Here Auxiliary carry is not set.

Case 2:- When nos are in the range of BCD, but there is a carry from lower nibble to higher nibble (1)

29	0010	1001	→	0101	0001	nibble & Auxillary flag is set.
28	+0010	1000		+	0110	
57	0101	0001		0101	0111	
	5	1		5	7	

→ Here auxillary carry is 1.

In this case, both 5 & 1 are in the range of BCD, but there is an auxillary carry from lower nibble to higher nibble, ∴ even though both 5 & 1 are valid, but Auxillary Carry is set. ∴ we add 6.

- **Zero Flag :-**

If the ALU result is zero

T → Set=1=Z

F → reset=0=NZ

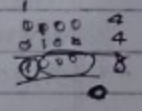
This flag is used to implement the control structures.

- **Sign Flag :-**

If in ALU result, MSB is 1

T = Set=1 = NL (Negative logic)

F = reset = 0 = PL (Positive logic)



- **Overflow Flag :-**

★ "If there is a carry into MSB & no carry out of MSB"

★ "If there is no carry into MSB & carry out of MSB"

T → Set = 1 = OV (Overflow)

F → reset = 0 = NV (No overflow)

Overflow is used in signed arithmetic computations.

Date: Page No.

- Signed nos. are represented by using 2's complement format.
- When MSB bit is '0', then the no. is +ve, no need to take the complement to conclude the value.
  If MSB bit is '1', then the no. is -ve, so there is a need of taking the 2's compliment to conclude the value.
- The range of n-bit 2's compliment no. is $-(2^{n-1})$ to $+(2^{n-1}-1)$

4 bit 2's complement range is :-
$-(2^{4-1})$ to $+(2^{4-1}-1)$ = -8 to +7 [total 16 combinations.]

Binary Sequence

Binary (2's complement)		Value :-
0 0 0 0	=	+0(0)
0 0 0 1	=	+1
0 0 1 0	=	+2
0 0 1 1	=	+3
0 1 0 0	=	+4
0 1 0 1	=	+5
0 1 1 0	=	+6
0 1 1 1	=	+7
1 0 0 0	=	-8
1 0 0 1	=	-7
1 0 1 0	=	-6
1 0 1 1	=	-5
1 1 0 0	=	-4
1 1 0 1	=	-3
1 1 1 0	=	-2
1 1 1 1	=	-1

0111
+ 1
1000

0110

0101
+ 1

Example:-

(a) $+7 \quad 0111$
$+3 \quad +0011$
$+10 \quad 1010 \rightarrow -6$
↳ "can"

Justify:- 0 1010
↳ acc.
↳ when the sign of result +10 is +ve, 5th bit is 0.

range is exceeding, & we can see that there is carry into MSB & no carry out of MSB.
∴ OV = 1.

(b) $+7 \quad 0111$
$-3 \quad 1101$
$+4 \quad 10100$

as there is carry into MSB & carry out of MSB, ∴ OV = 0.

(c) $-7 \quad 1001$
$+3 \quad 0011$
$-4 \quad 1100$

-4 is in the range of -8 to +7.
∴ overflow flag is '0'.

OV = 0.
Justify:- 0 1100 → accumulator
↳ overflow flag → (0 means the range is not exceeded, so no need of 5th bit.)

(d) $-7 \quad 1001$
$-3 \quad 1101$
$-10 \quad 10110$

no carry into MSB, but carry out of MSB,
∴ OV = 1.
we need 5th bit. ① 0110 → in accumulator
↳ in overflow flag.

Justify:- OV = 1, acc. = 0110
5th bit depends upon the sign of the result, which is -ve. (∵ 5th bit is 1).
∴ result:- 10110 (2's complement)

01001
$+ \quad 1$
$01010 \rightarrow$ ans.

eqn. of Overflow flag

MSB of operand 2

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

MSB of operand 1 — MSB of result

Q. Consider following two 2's-complement nos. & perform addition operation & evaluate flag conditions.

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

(C) Carry = 1 — Zero = 0 (NZ)
(PE) Parity = 01 — Sign = 0 (PL)
(NA) Auxillary = 0 — Overflow = 1 (OV)

Control Flags :-

Based on the status of these flags, the ALU execution sequence is changed. Three flags are employed under this category:-

1. Trap Flag
2. Interrupt Flag
3. Direction Flag

- Trap Flag (go = G) (trace = t) — used for debugging purpose

"If it is 0" :- all statements are executed at a time.
"If it is 1" :- only one statement is executed at a time.

-t=2000 → this means that trap flag is set & only one statement executes at a time starting from 2000.

- Interrupt flag
  - 0 → Disable the interrupt (DI) — Disable the Maskable interrupt
  - 1 → Enable the interrupt (EI) — Enable the Maskable interrupt
- Direction flag
  - Autoincrementing – 0 (CLD) (Clear Direction Flag)
  - Autodecrementing – 1 (STD) (Set Direction Flag)

Date
Page No.

Addressing Modes :-

It shows the way where the required object is present.

Object may be an instruction or data, ∴ the addressing modes are classified :-

(focussed on data)
Data oriented addressing mode
(1) Sequential control flow addressing mode.
(2) Transfer of control flow " ".

Instruction oriented addressing mode (focussed on instn)

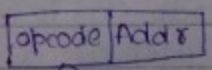
(1) instructions are stored sequentially.
(2) instructions are accessed using incrementation of program counter.
[no if, while, etc statements]

- O/p address of Addressing mode is effective address (EA). EA is the address where the required object is present.
- Object = [EA] ← content of EA

Addressing modes are implemented in 2 ways :-

(1) Implicit
(2) Explicit

- **Implicit** :- In this implementation, the type of addressing mode info. is present in opcode, ∴ there is no need of extra field in the instn format.

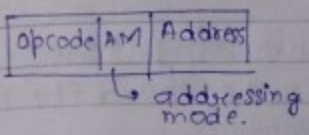


eg MVI B,23
MVI → move immediate
immediate addressing mode
this states that addressing mode is immediate which is specified in the opcode itself.

Mov r0, #25 → immediate mode
reg name → register addressing mode
[ ] → direct addressing mode.

Mov re,
@ or () → Indirect addressing mode

Explicit :-
In this implementation, addressing mode field is maintained in inst^n format to identify the type of addressing mode.



Decoders are required to decode which addressing mode is encoded.

Sequential Control Flow Addressing Mode :-
- This mode is focussed on the data.
- Data is present at two places :-
  - → Registers
  - → Memory

∴ This addressing mode is further divided into 2 types :-
1. Register Based AM :- when data is present in register
2. Memory Based AM :- when data is present in memory

Register Based AM :- This addressing mode is used to read & write the data when it is present in processor registers. Under this category only 1 addressing mode is used i.e. register addressing mode. This mode is used to access local variables. Registers are referred by the names.
EA = Register name
data = [EA] = [register name]

* Register Based AM:- When the data is present in the register itself this mode is used to access local variables because registers reside in the processor architecture whereas memory is seperate from processor architecture & it takes less time to access data from register.

e.g. MOV R0,R1 [R0 ← R1]
EA=R0 EA=R1

Opcode | Addr
↓
EA
↓
data | reg. name
↓
read/write

Memory Based Addressing Mode:-

It is used to read/write the data when it is present in the memory.
There are diff. addressing modes employed in this category:-

(i) **Implied**:- In this mode, the data is itself available in the opcode, ∴ no effective address.
e.g. STC:- set carry [there is no address field, it is implied in the opcode itself]
in this case we have to set the carry flag, the data is present in the carry flag, so it is implied that operand is in carry flag, other examples are CLD CLI (Clear Direction Flag), etc.

* all the zero address instructions employ the implied addressing mode, e.g. stack machine.

(ii) **Immediate Addressing mode**:- This mode is used to access the constant. In this mode, the data is present in the address field of instruction, i.e.

opcode | data → Immediate mode is used to initialize the registers with data.

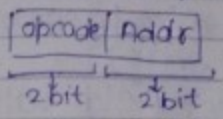
e.g. MOV R0,#25
source AM = immediate, then data = 25.
Destn AM = register, then EA = R0
R0 ← 25

Note:- The limitation of immediate is that the range

** Add A, B where A & B are the memory addresses.
M[A] ← M[A] + M[B]
3 memory references.

Date: Kitty
Page No.

of possible constants are delimited by the size of address field.



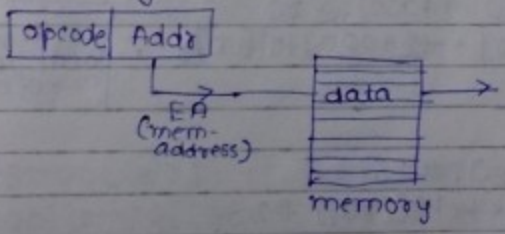
∴ Possible constants = 0 to $2^2 - 1 = 0 - 3$

(iii) **Direct Addressing Mode :-**
This mode is used to access the static variables.
In this mode the data is present in the memory, and that memory address is present in the address field of instruction.

EA = address field value.
↓
Memory addr.

Data = [EA]
= [Mem. Addr.]

1 memory reference is required to read/write the data using this mode.



ADD r0, [4000]
r0 ← r0 + M[4000]

(iv) **Indirect Addressing Mode :-**
This mode is used to implement the pointers.
Based on the availability of EA, this mode is divided into 2 types, i.e. register indirect AM, memory indirect AM.

EA = content of address field value

Date: Page No.

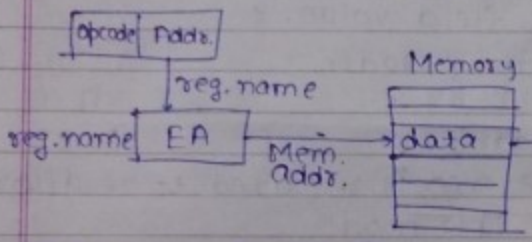
- **Register Indirect AM:-**
  In this mode, the EA is present in the register, that register address name is present in the address field of inst^n.
  ∴ EA = content of address field value.
  or EA = content of register name in address field.

  Data = [EA]
  = [[reg. name]]

  1 reg. ref = EA
  1 mem. ref = data

  To access the content of register which is given in the address field this content gives the EA in the memory to get the data.

  for 2 bit 1 mem ref



  e.g. ADD [4000], @R0
  M[4000] ← M[4000] + M[[R0]]
  ∴ Total references = 4
  Total memory reference = 3

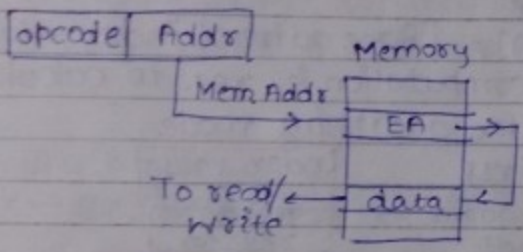
- Add @R0, #23
  M[[R0]] ← M[[R0]] + 23
  1 reg. ref = EA, 1 mem. ref = To write data
  1 reg. ref = EA, 1 mem. ref = read data
  23 → constant

  MOV R1, R2
  R1 ← R2
  Mov [R1], [R2]

- Addr **Memory Indirect AM:-**
  In this mode the EA is present in the memory, that memory address is present in the address field of instruction.

Date: Kitty
Page No.

EA = [Addr. field value]
= [Mem. Address]
data = [EA]
= [[Mem. Address]]
1 mem. ref. = EA
1 mem. ref. = to read/write data
2 memory reference are req.



→ e.g. ADD @4000, (5000) → 6 memory reference.

M[M[4000]] ← M[M[4000]] + M[M[5000]]

or

M[[4000]] ← M[[4000]] + M[[5000]]

1 mem. ref. = EA
1 mem. ref. = write data

→ ADD @5000, @r1 → 5 memory reference

M[[5000]] ← M[[5000]] + M[[r1]]

1 mem. ref. | 2 mem. ref. | 1 mem. ref. + 1 reg. ref.

→ MOV @5000, @r1

M[[5000]] ← M[[r1]]

2 mem. ref | 1 mem. ref. + 1 reg. ref.

→ Load @r1

ACC ← M[[r1]]

1 mem. ref. + 1 reg. ref.

→ Load @5000

ACC ← M[[5000]]

2 mem. ref.

(v) **Indexed AM:-**

Case I:- (When base address resides in the address field of the instⁿ & the index value resides in the index register)

This addressing mode is used to implement the arrays. To access the array element, there is a need of base address & the index value. Base address points the starting address of an array & the index value gives the offset address of an array.. ∴ EA = base address + index value

data = [EA] = [Base address + index value]

Arithmetic computation is req. to calculate the EA in the index addressing mode.

Base Address → present in address field of instⁿ

Index value → Index register.

Data = [[Base address + [Ri]]]

1 reg. ref. to get index value
1 arithmetic operⁿ to calculate EA.
1 mem. ref. is req. to read/write data.

e.g. ADD r0, 2(r1) (2 → base address) Where r1 is an Index reg.
↳ Index addressing mode

- r0 ← r0 + M[2 + [r1]] ← 1 mem. ref.
  1 reg. ref. 1 reg. ref. 1 mem. ref. 1 reg. ref. 1 arithmetic opⁿ.
- ADD @4000, 200(r2) → 5 mem. ref.
  M[M[4000]] ← M[M[4000]] + M[200 + [r2]]

★ Index AM is also known as Base-Indexed AM.

(Base Index AM):- ↳ address field = ?

Case II:- (When both base address & index value resides in register)

Some of the processors supports base & index registers to hold the base address & the displacement respec.

∴ EA = [Base register] + [Index reg.]

Data = [[Base reg.] + [Index reg.]]

1 reg. ref = to read base addr.
1 reg. ref. = to read index value
1 arithmetic opⁿ = to calculate EA
1 mem. ref. = to read/write data.

e.g.
MOV AX, [BX][SI]
AX ← M[[BX]+[SI]]

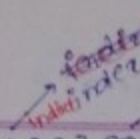
Case III:- Hypothetical Case:-
(When the address field contains a value of memory where EA of base address value is stored & index value is stored in index register)

Indirect Index AM:-
In this mode the base address is present in the memory, that memory address is present in the address field of the inst".

EA = [mem. Addr.] + [Index Register]
= [Addr. Field value] + [Index reg.]

Data = [EA]
= [[Mem. addr.] + [Index reg.]]
1 M. ref. = Base addr.
1 reg. ref. = Index value
1 Arithmetic = EA
1 M. ref = read/write.

e.g. MOV r0, [2000][r1] r1 ← index reg.
r0 ← M[2000] + [r1]
1 mem. ref. 1 reg. ref.
1 Arithmetic opⁿ = EA
1 mem. ref. = data



ADD A[R1] , B where A & B are mem. addr.
R1 is index address

M[[A]+[R1]] = M[[A]+[R1]] + M[B]]

⭐⭐ Using index addressing mode, we can access the random array elements. (by changing randomly changing the value of index register)

(vi) Auto-Indexed :- this means we can only access the array elements one by one

This addressing mode is used to access the linear array element.

∴ EA = Base address + step size

data = [EA]
= [Base address + step size]

this implies that we can access element no 5 only when we have accessed array no 1, 2, 3 & 4 (in autoincrement mode)

If array element size = 1 byte, then step size = 1
If size of array element = 2 byte, then step size = 2.

The step size depends on the amount of the data element to be accessed from the array.

Classification of Auto-Indexed AM :-

- Auto-Indexed AM
  - Auto Increment
    - pre autoincrement MOV R0, +(R1)
    - post autoincrement MOV R0, (R1)+
  - Auto Decrement
    - pre autodecrement MOV R0, -(R1)
    - post autodecrement MOV R0, (R1)-

Q. Diff. addressing modes are used to access the operands. The frequency of operand accessing is shown :-

Operand Accessing	freq.(%)
Register	30
Immediate	20
Direct	22
Mem. Indirect	17
Indexed	11

If 2 cycles are consumed for mem. operations, 1 cycle consumed for arithmetic computation & 0 cycles consumed if operand is available in the register/inst^n itself, what is the avg. operand fetch rate of the machine.

Ans. $0.22 \times 2 + 4 \times 0.17 + 3 \times 0.11 = 0.44 + 0.68 + 0.33$

$= 1.45$

Avg. operand fetch $= (0.3*0) + (0.2*0) + (0.22*2) + (4*0.17) + (3*0.11) + 0.11 = 1.45$ cycles. to fetch the one operand.

Transfer Of Control Flow AM

To implement the control structures, i.e. selection statements (If-then-else, switch-case & goto), iterative statements (while, do-while & for) & subprogram concept, there is a need of transfer of control operations.

During the execution of the above statements, the program control is transferred from current location to target location.

The transfer of control operations are associated with the target address. There are 3 possible mnemonics available to implement the transfer of control operations:- ① Branch ② Jump ③ Skip.

Date
Page No.

Transfer of control operations are basically divided into 2 types:-

1. Unconditional Transfer Of Control
2. Conditional Transfer of Control

- **Unconditional Transfer Of Control:-**
While execution of these inst^n, the control is transferred to the target address w/o ~~them~~ checking any condition.

e.g.

1000: JMP 2000 → Target Address
1001:
↳ Unconditional TOC

At the end of this inst^n
PC ← 2000

If JMP 2000 is at 1000 location & next inst^n is at 1001 location, then prior to the execution of JMP 2000, PC contains 1001, but after execution of JMP 2000, 1001 is overwritten by 2000.

Consider the program segment

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

The execution sequence of the program is:-

- t = 1000
PC = 1000
1001
1002 JMP 2000

PC points to 1003 even before JMP 2000 is executed, ∴ PC becomes 1003, after execution of JMP 2000, the PC is replaced by 2000.

PC = 1003 → before execution of JMP 2000
PC = 2000 → after execution of JMP 2000

~~dep~~ content 1003 is

Spaghetti ca

Date
Page No.

PC = 2000 : BI_1
PC = 2001 : JMP 1003
PC = 2002 → before execution of JMP 1003.
PC = 1003 : I_4 → after execution of JMP 1003 & before execution of I_4.

★ To implement the goto, halt statements, machine control statements unconditional transfer of control statements are used.

→ Halt is implemented using unconditional Transfer of control when the target address is same as current inst^n.

2000 : Halt;
or 2000 : JMP 2000
l_1 : JMP l_1

this will halt the processor by executing the same statements again & again by loading the PC with same inst^n address.
To come out of halt inst^n, we need to restart the system.

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

Imp.

PC = 1000
= 1001
= 1002 : halt
= 1003 → before execution of halt.
Halt 1002 :- after execution of this halt.
1003 :- before execution of this halt.
Halt 1002 :- after execution of this halt.
1003 :- before execution of this halt.

Date / Page No.

Machine Control Meaning:-

Conditional Transfer Of Control:-

These inst^n are associated with the conditions, during execution of conditional TOC execution, the associated cond^n undergoes evaluation. If cond^n is true, then PC is loaded with target address otherwise no change in PC. (i.e. PC will hold the next sequential address.)

e.g. JNZ 2000

↳ Jump when not zero

If inst^n before JNZ 2000 resets the zero flag, then only the PC is loaded with 2000, if inst^n before JNZ 2000 sets the ZF, then the PC is only incremented to the next instruction after JNZ 2000.

1000: MOV CL,03	PC=1000	CL=03
1001: ADD AL,BL	1001	+
1002: DEC CL	1002	CL=02
1003: JNZ 1001	1003	(when PC=1003 then CL=02.)
1004: MOV [4000],AL	1004	NZ ↳

PC=1001 +
PC=1002 CL=01
PC=1003 NZ
PC=1004
PC=1001 +
PC=1002 CL=00
PC=1003 NZ(F)
PC=1004 → at this time, PC will remain loaded with 1004, no overwritting of PC takes place.

non-zero condition is checked, but PC=1004, but Non-zero condition is satisfied, so 1004 is overwritten by 1001.

*NOTE:- To implement the selection & iterative statements conditional transfer of control are used.

Date 22.04.12

V: Imp.

Date: Page No:

Subprogram

Subprogram means reusable program. Seperately reappeared inst^n in the main application & store them in the seperate memory space called subprogram.
The advantage with the subprogram is that it reduces the memory space, development time & development cost.

Characteristics of subprogram:-

1. Single entry, single exit.
2. Main program is suspended during execution of subprogram.
3. Control is transferred back to the main program after completion of the subprogram.

Implementation of subprogram:-

Subprogram concept is implemented by using pair inst^n, i.e. call-ret

↓ entry pt. (i.e. entry pt. is present in the main program.) → main program

↓ exit pt. → present in the subprogram.

call invokes the push operation of the stack to save the return address.

return invokes the pop op^n of the stack to retrieve the return address of main program.

As we haven't saved the return address anywhere, ∴ the subprogram waits for the return address.

Consider the following program segment & show execution sequence.

1000: I_1	2000: BI_1
1001: I_2	2001: BI_2
1002: I_3 (call 2000)	2002: BI_3
1003: I_4	2003: BI_4 (Ret)
1004	2004

(wait)

t = 1000
PC = 1000
PC = 1001 I_1 is executed
PC = 1002 I_2 " "
PC = 1003 I_3 " "

control is transferred to the subprogram.

PC = 2000
PC = 2001 BI_1 is executed
PC = 2002 BI_2 " "
PC = 2003 BI_3 " "
PC = 2004 BI_4 " "

because return address is not saved anywhere. [We need runtime stack to save the return address]

control is transferred to the main program, so PC value 2004 is replaced by

Now, the subprogram waits at PC = 2004 because the return address is not available.

Due to absence of return address, the control is not transferred back to the main program. To make the implementation successful, there is a need of runtime stack. Runtime stack is used to store the return address.

Execution seq. with stack

PC = 1000

1001 I_1 is executed

call invokes the push opn & passes the current value of PC into the stack

1002 I_2 is "

1003 I_3 " " [I_3 :- call 2000]

PC = 2000

2001 BI_1 is executed

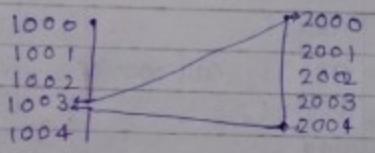
2002 BI_2 is "

2003 BI_3 " "

2004 BI_4 " " [BI_4 is a return statement]

at this time, return pops the TOS & retrieves the return address of the main program. This return address is used to overwrite PC value of 2004

PC = 1003



★ Only PC is changed during transfer of control operations.

JMP 200 — Transfer Of Control

Direct TOC
- The target address is present in the instn itself e.g. JMP 2000, JNZ 2000, CALL 2000 (target address in the instn.)

Indirect TOC
- Target address is not present in the instruction, e.g. RET → no target address in the instn.

TOC	Unconditional	Conditional	(call on not carry)	(call on zero)
Direct	JMP, goto, Hall, call	JNZ, if-then-else, while, do-while, switch, for,	callc (call on carry),	callnc, callz
Indirect	Ret	Retc, Retnc, Retz, Retnz		

Return on carry, Return on no carry, Return on zero, Return on non zero

Under the transfer of control flow, 2 addressing modes are employed to calculate the EA of the next instⁿ, i.e.

① Relative / PC-relative AM

② EA = PC + IR[Address field value]

When the control is to be transferred in the same segment. [Intrasegment]

Displacement / relative value from the current PC value, what displacement must be added to PC to get the EA of next instruction, so the address field of IR contains this displacement.

∴ PC ← PC + displacement.

② Base-register addressing mode :-

In this mode, the base register is used to point the code segment address. When the program is present in the diff. segments, to execute such program base register addressing mode is used, easy relocation of program.

When the control is to be transferred from one segment to other [Intersegment]. In this case, the base register contains the starting address of the other segment & the address field of IR contains the displacement in other segment where next instⁿ is present.

∴ EA = [Base reg.] + IR[Addr field value]

Starting add. of segment

Displacement / relative value

Q1 A PC relative branch instⁿ size is 32 bits. It is placed in the memory with starting address of 342032 (decimal). What is the target address if the signed displacement -12 is placed in the instⁿ.

Ans. 342032, 342033, 342034, 342035 } — Instⁿ — opcode | -12

The instⁿ is fetched from 342032, now as instⁿ is of 4 bytes & it is byte addressable memory, ∴ instⁿ is fetched from 342032 - 342035, but before execution of this instⁿ, PC is incremented to 342036 so after execution of this instⁿ, PC = 342036.

PC = 342036 at present

EA = PC + IR[addr. field value] = 342036 + (-12) = 342024

PC = 342036

Now, PC = PC + IR[Address field] = 342036 - 12 = 342024

Q2. A PC-relative branch instⁿ with 5 bytes long is stored in memory 243028 (decimal) address onwards. If EA = 243009, what's the displacement?

Ans EA = PC + displacement

243009 = 243033 + displacement

∴ displacement = -24

243028
243029
243030
243031
243032
→ Instn.

PC = 243033

Types Of Operations/Types Of Instn./ Instⁿ Set

- Instⁿ set of processor consists 3 category of instⁿ to perform 3 diff operations, named as:-
  1. Data Transfer Opⁿ.
  2. Data Manipulation Opⁿ.
  3. Transfer of Control.

- **Data Transfer Instⁿ**:- While execution of these instⁿ, the data is transferred from source to destination. The constraint is that source may or may not have the storage but the destination always have the storage.

Source	Destⁿ	
reg	reg	→ MOV
reg	Mem	→ Store
Mem	reg	→ load
? → Mem.	Mem	→ not possible
Immediate	Reg.	→ MOV
Immediate	Mem.	→ MOV
TOS	reg.	→ POP
reg	TOS	→ push
Input device	reg.	→ IN
reg.	o/p device	→ OUT

Diff. mnemonics are used to implement various data transfer operations.

MOV:- General purpose data transfer operation.

MOV R0,R1 MOV [5000],R0

MOV R0,[4000] MOV R0,#23 → 8051 instⁿ.

Load:- It performs only memory read operations.

Load R0,[4000] Load R0,@5000

Load R0,@R1 Load R0,R1 → invalid

Load R0,#23 → invalid [no memory to memory operations in Load.]

Store:- It is used to perform memory write operations only.

Store [4000],R0
Store @R0,R1
Store 2(R1),R0
Store @5000,#23
} valid

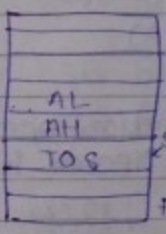
Store @R0,@R1 X → not valid [because it is memory to memory opⁿ.]

Store R0,@R1 → invalid

Push:- ~~Data is~~

SP:- Stack pointer

Push AX



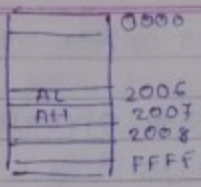
Now, TOS will contain some data, ∴ 2008 will contain some data, SP is decremented by 1, ∴ SP=2007, AH is stored at 2007, then SP is again decremented by 1, ∴ SP=2006 & AL is stored at 2006.

It is used to transfer the data from register to TOS. Based on the displacement defines upon the size of register.

For every successful push instⁿ the stack pointer is decremented by displacement times, in above case stack pointer is decremented by 2.

SP ← SP-1
TOS ← AH
SP ← SP-1
TOS ← AL
} → pre increment.

Pop:



Pop BX

know BX is 16 bit, so we require 2 cell values.

∴ BL = M[PC] → post increment
PC ← PC+1
BH = M[PC]
PC ← PC+1, now PC will be at 2008, & 2008 mem. address will now be the TOS.

For every successful pop opⁿ, stack pointer is incremented by the displacement times.

If pop r
where r is a register of 32 bits, then SP is incremented by 4

- **Data Manipulation Instn.**

While execution of these instⁿ one form of data is converted into another form. The classification of Data Manipulator instⁿ is :-

1. Arithmetic Instⁿ (Add, Sub, Mul, Div, Inc, Dec)
2. Logical Instⁿ (And, OR, NOT, XOR, CMP)
3. Shift & Rotate Instⁿ

- CMP :- This instⁿ invokes the subtraction operation, after the operation the result is reflected in the Zero & Carry Flags. e.g. CMP R0, R1 ; R0 = 23, R1 = 23
in this case Zero flag is set.

If both operands are same, then ZF = 1, CF = X

After the opⁿ read the status of zero flag, if it is 1, then we conclude that both operands are equal.

example 2:- CMP R0, R1 ; R0 = 40, R1 = 20

SUB: 0100 0000
0010 0000
0010 0000 → NZ
→ NC

If Op1 > Op2, then ZF = 0, CF = 0

→ ~~Carry fl~~
If ZF = 0 & CF = 0, then we conclude 1st operand > 2nd operand.

example:- CMP R0,R1 , R0=20 R1=40 → If ZF=0 & CF=1, then we conclude 2nd operand is greater than the 1st operand.

Borrow =1 — 0010 0000 − 0100 0000 = 1110 0000

If Op1<Op2 ZF=0 CF=1

∴ CF=1 ZF=0 [the ~~borrow~~ carry flag is set when we take borrow from left hand side of MSB.]

③ Shift Oper^n — logical shift — left (LSL), right (LSR); arithmetic shift — left (ASL), right (ASR)

While execution of this inst^n, the bits are moving towards either left or right by 1. The classification is shown below:

→ In logical shift op^n, all the bits are moving towards left or right including the sign bit.

LSL: lost ↑ MSB (sign) reg. 0

The MSB bit is lost, & a '0' is inserted at LSB.

LSR: reg. → 0 → MSB ... LSB

The LSB bit is lost, & a '0' is inserted at MSB

★ Shift op^n is used in serial data communication.

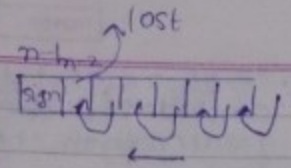
LSL R0 ; R0=23 Multiplication by 2. LSR R0; R0=23 Division by 2.

0 0 1 0 1 1 1 ← 0 0 → 0010 0011

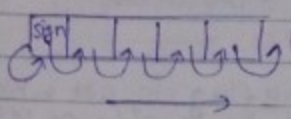
0100, ~~0~~ 0110, 0001, 0001

4, 6, 1, 1

NOTE:- Logical shift op^n is used in the serial data communication. In arithmetic shift op^n all bits are moving towards left or right except the sign bit.

ASL: 

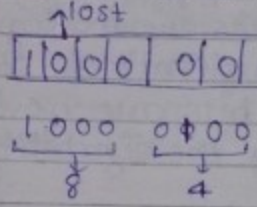
sign bit remains same, (n-2)th bit is lost.

ASR: 

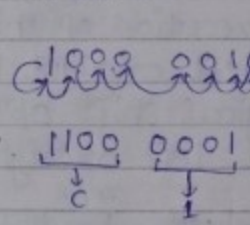
LSB is lost, & at sign bit the same bit is shifted in the sign bit.

This is used in signed arithmetic computations.

ASL R0 : R0=82H

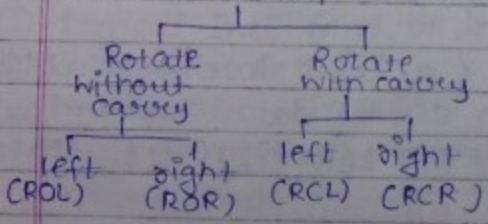


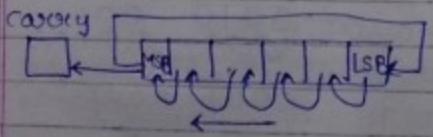
ASR R0

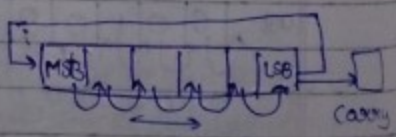


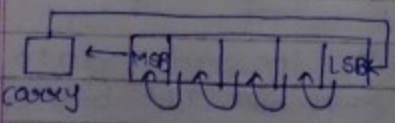
④ Rotate Operation:-

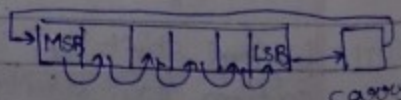
While execution of this inst^n all the bits are moving towards left or right without loss of data. The classification



ROL: 

ROR 

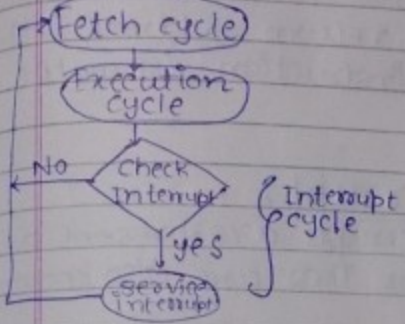
RCL: 

RCR 

Note:- Rotate inst^n is used in optimization in digital motors, to check whether a no. is even or odd (check LSB), to check whether a no. is +ve or -ve. (check MSB).

Interrupts

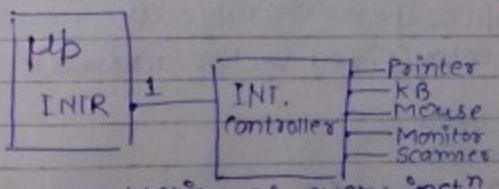
An usual event to disturb the normal flow of execution. Instⁿ cycle with interrupt concept describes the execution sequence of the program along with the interrupts.



CPU responds to interrupt only when current instruction is executed, i.e. if the interrupt occurs during the execution of instⁿ, then the interrupt will only be handled after the current instⁿ is completely executed. If two interrupt occurs during

★ The above flow chart shows that the CPU will respond to the interrupts only after execution/completion of current instruction.

NOTE: The interrupt sources are enabled or disabled based on the status of the interrupt flag. If interrupt flag — 0 → Disable the interrupt sources → No interrupt cycle; 1 - Enable the interrupt sources → Interrupt cycle is required to service the interrupts.



• After completion of every normal instⁿ, CPU checks whether there is an interrupt or not.

After completion of every instⁿ execution, the CPU reads the status of Interrupt sources, e.g. INIR.

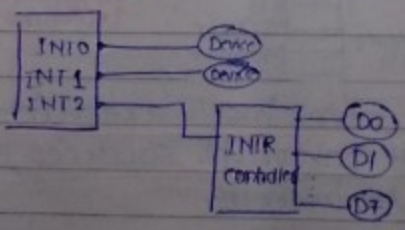
INIR — 0 = No-interrupt; 1 - Interrupt.

The processor supports diff. interrupts, so diff. subprograms are required to handle diff. interrupts. All these interrupt subprograms (ISR) are stored in the long term memory which is called as Interrupt Vector Table (IVT). Each ISR must end with IRET (Interrupt Return) or RETI instruction.

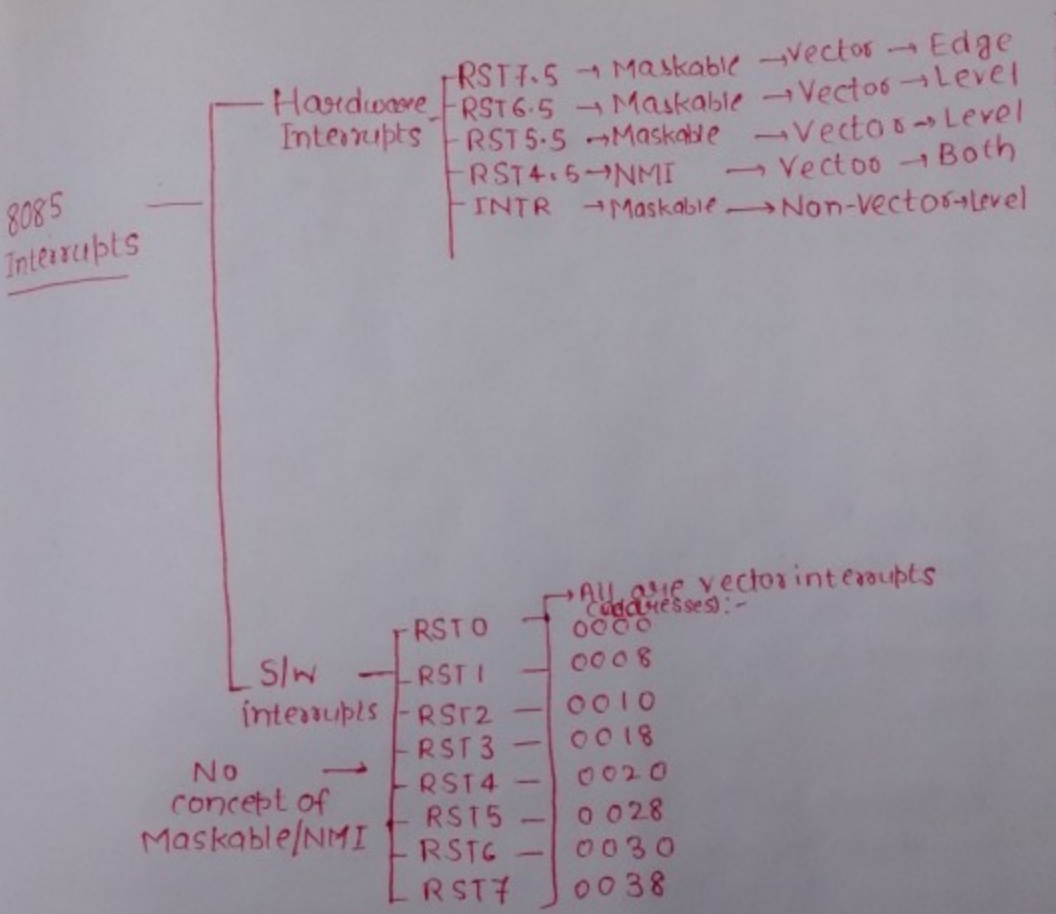
When the CPU identifies the interrupt, it pushes the PC into the stack & load the vector address into the program counter. When the CPU encounters IRET/RETI instructions, it invokes the pop operation to restore the PC with the return address. ∴ the CPU can fetch the next instruction from the main program.

Types Of Interrupt:-

1. H/W interrupt :- These interrupts are present at the h/w pin of the processor. These are also known as external interrupts.
2. S/W interrupt :- These interrupts are present in the program, i.e. s/w interrupts are the instruction.
3. Maskable interrupts :- These interrupts can be enabled or disabled.
4. Non-maskable interrupts (NMI) :- These interrupts are always in the enabled state, so the power failure conditions, temp. sensor conditions & critical condition of the processor are connected to the NMI pin.
5. Vector Interrupts :- These interrupts are associated with static vector pointer (fixed vector address).



INT0, INT1 are vector interrupts as only one device (which obviously have a single address) is attached to the pin,
but INT2 is non-vector interrupt when more than one device is attached to the interrupt pin.

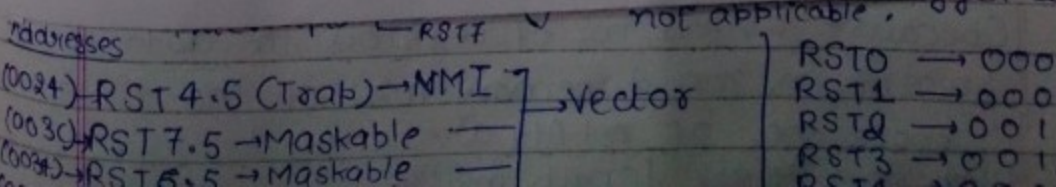


Interrupt Priority:-

RST4.5 > RST7.5 > RST6.5 > RST5.5 > INTR

RST4.5: Highest priority

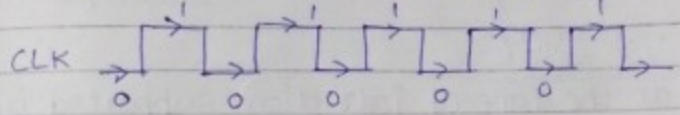
INTR: lowest priority



Date:
Page No.

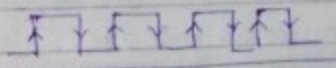
④ Non-vector interrupts: These interrupts are associated with the dynamic vector interrupts. These in the example, INT2 pin is connected with multiple sources, ∴ multiple vector address are possible.

⑤ Level triggered interrupts:- These are enabled or disabled based on the levels, i.e. 1 or 0.

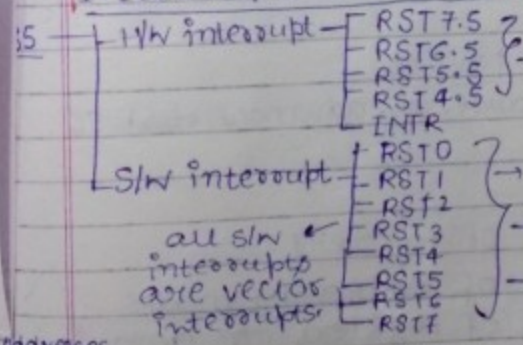


eg. INTR pin
when Level is 1, only then interrupts are enabled.

⑥ Edge triggered interrupts:- These are enabled or disabled based on the rising edge transition or falling edge transition.



Interrupt Structure:-



addresses

(0024) RST 4.5 (Trap) → NMI → Vector
(0030) RST 7.5 → Maskable
(0034) RST 6.5 → Maskable
(003C) RST 5.5 → Maskable → Non-vector
INTR → Maskable

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

Edge or Level:-

RST 7.5 → Edge
RST 6.5 → Level
RST 5.5 → Level
RST 4.5 → Both level & edge
INTR → level

Date:
Page No.

Interrupt Priority

When the processor supports multiple interrupt sources then there is a possibility of simultaneous interrupts so to service 1 interupt at a time, there is a need of priority level.

RST4.5 > RST7.5 > RST6.5 > RST5.5 > INTR

Highest priority — Lowest priority

RISC Vs CISC

Based on the type of instruction supported by the processor, architecture is divided into 2 types, i.e.

→ Reduced Instruction Set Computer
→ Complex " " "

Characteristic of RISC:-

1. It contains more no. of register. (this means that possibility of getting data from register is high.)
2. It supports less no. of Addressing Modes. (As most of the times data is present in the registers, ∴ memory access is low, less of addressing modes are there)
3. It supports fixed length inst^n.
4. One instruction per cycle because of fixed length inst^n.
5. It allows successful pipeline.
6. Motorola processor, power PC, advanced ~~risk~~ RISC machine (ARM).

Characteristic Of CISC:-

1. It contains less no. of registers.
2. It supports more no. of Addressing modes. (because as less no. of registers, the data is present in ~~no~~ memory most of the time, ∴ more no. of add AM are there.)
3. Supports variable length inst^n
4. ~~CPI~~ CPI ≠ 1. ~~1 Instructions~~ cycles per inst^n,
5. Unsuccesful pipelining
6. Pentium processor.

Register Organisation in RISC processor :-

1. In RISC processor, the registers are organised as overlapping register windows.
2. RISC processor supports diff. types of registers, i.e. global registers (G), Local registers (L), In reg. (C), out reg (C).

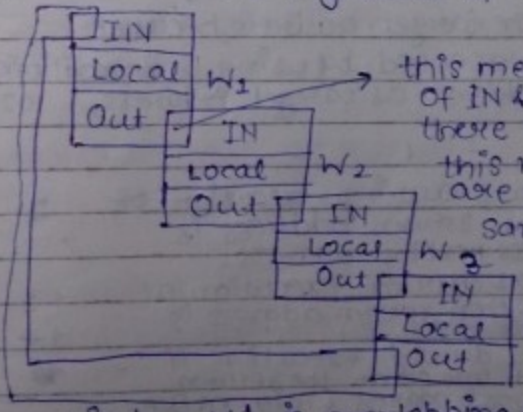
Global reg. are accessed by all the register windows.
Register window consists of Local, In & out reg.

Window
In (C)
Out (C)
Local (L)

Window size = L + 2C + G → as global reg. are accessed by register windows, we have added G also.
↳ no. of registers in one window.

★ The In reg. of one window is accessing the out reg. of another window.

The register file indicates the no. of registers supported by the processor. Overlapping register window organisation means one window out register acts as the In registers of another window.



this means that same no. of IN & OUT registers are there in overlapping region, this means that if there are n_1 out reg. in W1, then same n_1 reg. acts as the IN reg. in window W2.

Now, window W1 is overlapping with W2, W2 with W3, W3 with W4 & W4 with W1.

Register file = W*(L+C) + G → total no. of registers in the processor.

W :- total no. of windows
L :- no. of local reg.
C :- no. of common reg (IN-OUT)
G :- no. of global reg.

Date

28.04.12

Date:
Page No.

Components Of the Computer

The computer consists 3 basic components:-

1. CPU
2. Memory
3. I/O

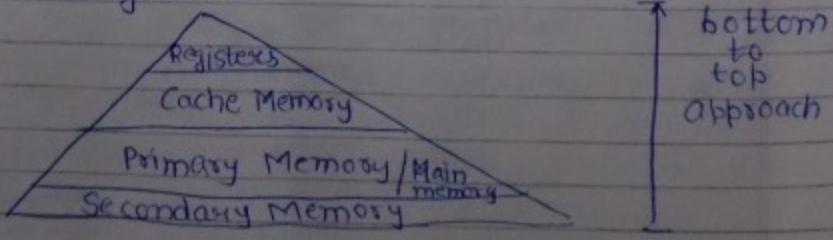
CPU Organisation:-

CPU consists of 3 components, named as:-

1. Registers
2. ALU
3. CU

→ Registers:- Registers are used to store the data temporarily. (Registers are present in the CPU itself, i.e they are not connected to CPU via Bus)
While execution of the program, the CPU is frequently communicating with the memory, synchronisation mechanism is req. Memory hierarchy design is the synchronisation mechanism b/w memory & CPU, the objective of memory hierarchy design is "Balancing the bandwidth b/w CPU & memory or Minimizing the speed gap b/w CPU & memory or synchronizing the data transfer rate b/w CPU & memory."

Memory hierarchy means arranging ~~the~~ all the system supported memory on priority basis & assign the accessing order.



From bottom to top (from Secondary Mem. to Registers)

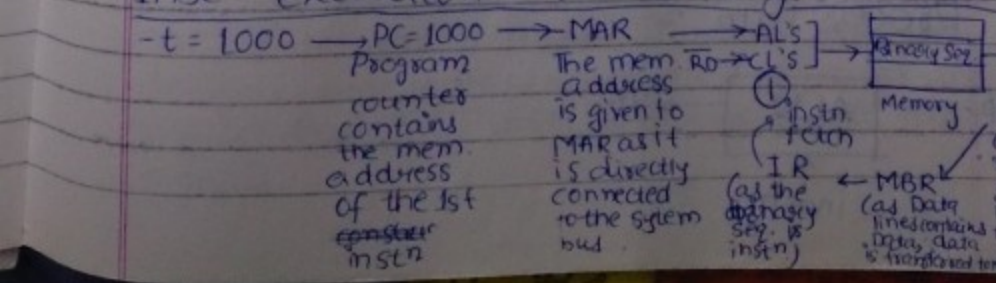
1. Capacity Decreases
2. Access time decreases.
3. Performance $\propto \frac{1}{\text{Access time}}$
4. Performance increases
5. cost per bit increases, expensive

Acc. to the memory hierarchy design, the registers are small sized storage components, so the frequently used data is placed in the registers. Those registers are present in the CPU, they are not connected to the system bus.

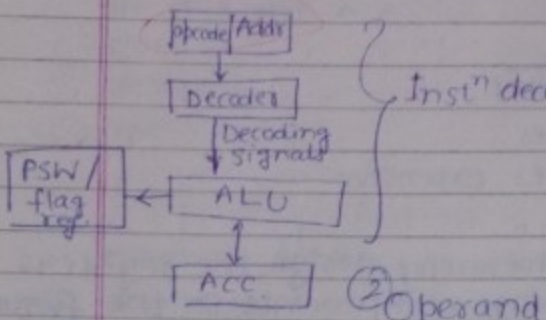
Mandatory Registers :-

1. Program Counter :- holds the starting inst^n address & immediately points to next inst^n address.
2. Inst^n register (IR) :- used to hold the currently fetched inst^n to decode because inst^n format is predefined in this register.
3. Accumulator (Acc) :- It is used to hold the ALU i/p's & o/p's.
4. Memory Address Register (MAR) :- used to carry the address. This reg. is directly connected with the address lines of the system bus.
5. MBR (Mem. Buffer register) / MDR (Mem. data register) :- used to carry the binary sequence. It is directly connected with the data lines of the system bus.

Inst^n execution Sequence using microoperations :-



After this PC ← PC + step size (after the current inst^n is fetched & is given to the CU for its decoding, the PC is incremented acc^g to the step-size of the inst^n, & now next inst^n is in the transit for fetching while the current inst^n is decoded.)



② Operand Fetch

① Reg. Addressing Mode:-
IR[Addr] → regname → ALU

② Immediate AM
IR[Addr] → ALU

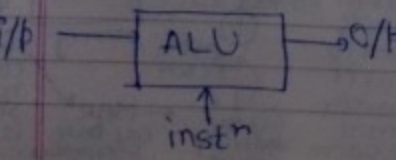
③ Direct AM
IR[Addr] → MAR → AL's / RD → CL's → Memory (Binary Seq.) → DL's → MBR → ALU

④ Reg. Indirect AM
IR[Addr] → Reg. name → EA → MAR → AL's / RD → CL's → Memory (Binary Seq.) → DL's → MBR → ALU

⑤ Mem. Indirect AM
IR[Addr] → MAR → AL's / RD → CL's → Memory (Binary Seq.) → DL's → MBR → MAR / RD → Memory (Binary Seq.) → DL's → MBR → ALU

The binary seq is address, it is transferred to MAR

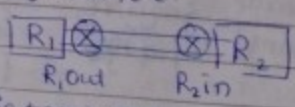
③ Process Data :-



• Instruction :- written by programmer, which is decoded by Control Unit.
• Microinstruction :- Every instⁿ has a sequence of microinstⁿ behind it that executes to get Microoperations :- oberⁿ of that instⁿ

Page No.

• Microoperation is a basic reg to reg. transfer opⁿ It is also known as elementary opⁿ or atomic opⁿ Each microopⁿ can be completed in 1 cycle Some of the microoperations requires more than one cycle e.g. $R_1 \rightarrow R_2$, to execute the μopⁿ, there is a need of control signals, i.e.



all registers are connected to each other, so for transfer of $R_1 \rightarrow R_2$, R_1out should be enabled, but the data is accessible to all other reg., but only those reg. which want to access the data will enable their Rin pin.

Microprogram :- The seq. of μopⁿ is known as the μprogram μopⁿ is also known as μinstⁿ or control word e.g. the fetch cycle μprogram is

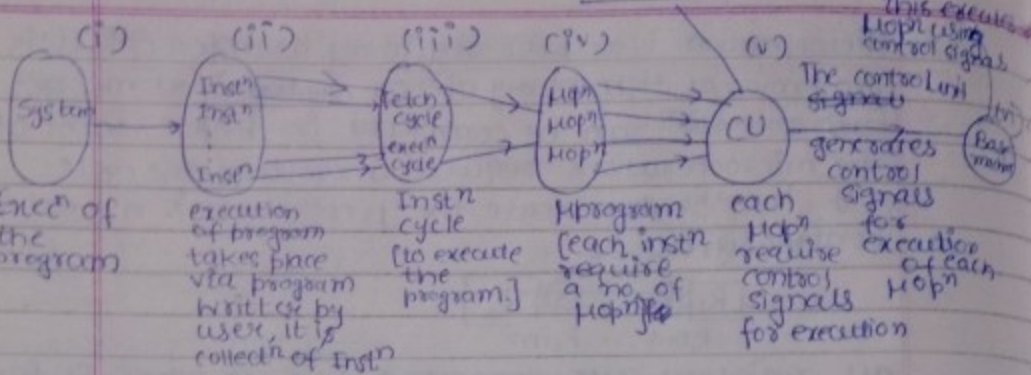
Instⁿ Fetch
T_1: PC → MAR, PCout, MARin
T_2: M[MAR] → MBR; MARout, MBRin :- This μopⁿ can't be completed in 1 cycle as it includes mem. access.
T_3: MBR → IR, MBRout, IRin
PC ← PC + step-size, PCout, PCin

Control Unit :-

System functionality is execution of the program. Program contains 2 objects, i.e. instⁿ & data. Instⁿ cycle is used to execute the instⁿ, Instⁿ cycle contain fetch & execution cycle These subcycles invokes their μprogram. μprogram means sequence of μopⁿ (control signals are req. to execute a μopⁿ. control signals are implemented in the control unit during the design time of processor. CU generates the control signal. Basic b/w unit is able to process the control signals.

Horizontal prog | Vertical prog

Hardwired Approach | μ-programmed Approach



This executes μopⁿ using control signals

To design the control unit, the designer identifies 3 points :-

① no. of instⁿ supported by the processor.
② no. of μopⁿ req. to execute one instⁿ.
③ no. of control signals req. to execute 1 μopⁿ

e.g. 3 Instⁿ : ADD, SUB, MUL
4 μopⁿ : T_1, T_2, T_3, T_4
control signals : S0, S1, S2, S3

μopⁿ	ADD	SUB	MUL
T_1	S0, S3	S0, S2	S3, S0
T_2	S1, S0	S1, S3	S1, S2
T_3	S2, S3	S2, S0	S0, S2
T_4	S1, S3	S1, S0	S0, S1

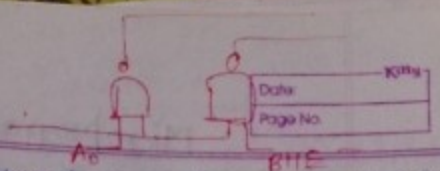
This database is stored in the control unit for decoding the μopⁿ for execution of the program.

In RISC processor, all the microinstⁿ completes in same no. of T-states, while in CISC processor, each microinstⁿ may complete in variable time states.

→ In (ii) we write the program as ADD
ADD

→ In (iii), the program is fetched & executed, & the instⁿ are decoded to find out which opⁿ to perform by accessing the CU, so fetch cycle sees the opcode in instⁿ & checks the CU for that opcode, which opⁿ to perform, as ADD is to be performed.

→ In (iv), as we get to know which opⁿ is to be performed, it checks the database in CU to check which μopⁿ are req. to perform the program execution.



After maintaining the database related to the inst^n, μop^n & control signals, the designer can use any one of the following approaches to design the control unit :- i.e

1. Hardwired control unit
2. Microprogrammed CU

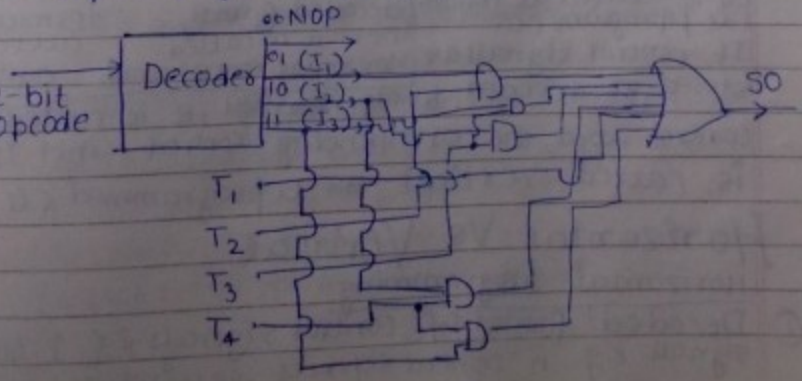
Hardwired CU :-

1. In this design, the control signals are expressed as Sum of Product expression (SOP). & they are directly realized on the independent h/w. (ex: arithmetic, processing)
2. This design is used in real time applications.
3. It is the fastest control unit design.
4. Even a minor modification requires redesign & reconnection of the control signals, it is not flexible.
5. It is not suitable for testing & design places.
6. The RISC control unit is the hardwired CU.

Q1. Consider a hypothetical CU, it supports 3 inst^n (I_1, I_2, I_3) & 4 control signals (S0, S1, S2, S3). Each inst^n requires 4 μop^n (T_1, T_2, T_3, T_4). The following table indicates the no. of control signals used for each μop^n for each inst^n. Get the control exp^n for S0 & S3 signals

Ans. $S0 = \bar{T}_1 + I_1 T_2 + I_2 T_3 + I_3 T_3 + I_2 T_4 + I_3 T_4$

$S3 = \bar{T}_1 I_1 + \bar{T}_1 I_3 + \bar{T}_2 I_2 + \bar{T}_3 I_1 + \bar{T}_4 I_1$

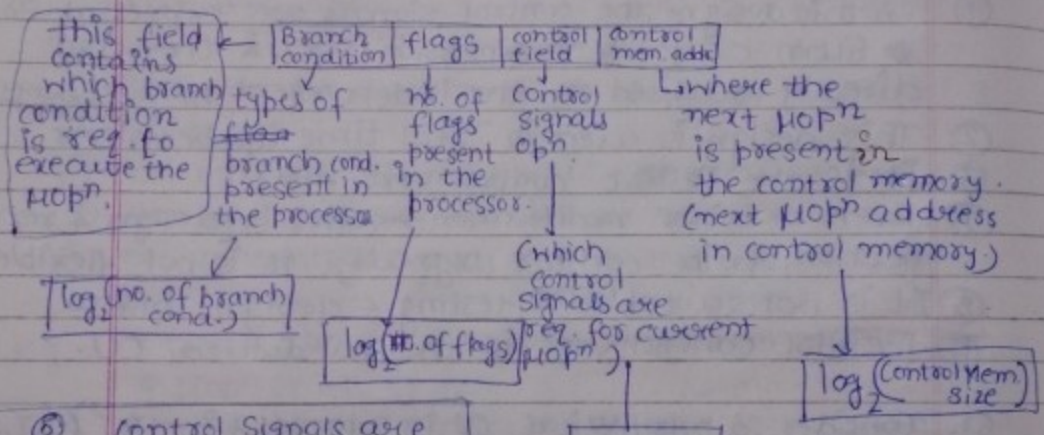


Date
Page No.

Microprogrammed CU

① In this design, the control memory is used to store the μprogram.
② The control memory is a permanent memory, i.e. ROM.
③ The μinstⁿ (control word) are stored in the control memory.
④ The format of μopⁿ is described below:-

Control word/μInstⁿ/μopⁿ



⑤ Control Signals are stored in 2 formats, i.e Decoded or Encoded format.
If control signals are stored in decoded format, it is called as horizontal control word, the corresponding CU is called as horizontal μprogram CU.
If control signals are stored in encoded binary format, it is called as vertical control word, the corresponding control unit implementation is called vertical μprogrammed CU.

Decoded format [1 bit/control signal] → When control word is stored in decoded format, then it is called Horizontal control word.

Encoded format [n control signals requires $\log_2 n$ bits.] → When control word is stored in encoded format, then it is called vertical control word.

Horizontal Vs Vertical -

Horizontal programming:

① Decoded format of control signals i.e 1 bit/control signal. e.g. n control signals requires n bits.

③ It supports ~~#~~ longer control world.
④ No need of external decoders to generate the control signals, so it is faster than vertical.
⑤ It allows high degree of parallelism.
e.g If degree of parallelism is 'N', which means n control signals are enabled at a time, so it is used in parallel processing applications.
⑥ It is bit flexible compared to the hardwired CU.

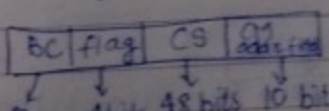
Vertical Programming:-

① It supports encoded format of control signals, i.e. n control signals req. $\log_2^n$ bits.
② Shorter control words.
③ Decoders are req. to generated the control signals, so it is slower than horizontal.
④ Low degree of parallelism, i.e. 1 or none.
⑤ It is very flexible.

- Ascending order in terms of speed:-
  Vertical < Horizontal < Hardwired
- Ascending order in terms of flexibility:-
  Hardwired < Horizontal < Vertical.

8. Consider a CU which supports 48 control signals & 12 flags. It has 1024 control word memory, what is the size of control word (in bits) & control mem. in bytes using horizontal programming.

Ans.



CW size = 62 bits

control word mem = 62×1024 bits

$= \frac{62 \times 1024}{8}$ bytes

Q. Consides a hypothetical control unit, it uses 6 groups of mutually exclusive control signals.

Group	G_1	G_2	G_3	G_4	G_5	G_6
CS	1	5	8	14	3	6

How many no of bits are saved using the vertical over horizontal

→ Horizontal - 1+5+8+14+3+6=37

Vertical - $\lceil \log_2 1 \rceil + \lceil \log_2 5 \rceil + \lceil \log_2 8 \rceil + \lceil \log_2 14 \rceil + \lceil \log_2 3 \rceil + \lceil \log_2 6 \rceil$

$= 1 + 3 + 3 + 4 + 2 + 3 = 16$

No of bits saved = 37 - 16 = 21

High Performance Architecture:-

Performance is an indirect measurement which depend upon the execution time or throughput

- Execution Time:- time required to complete the program execution. It is also called as elapsed time or response time

→ Consider two systems X & Y, used to execute task1, X system requires 5ns & Y requires 10ns to complete the task

$$\text{Performance} \propto \frac{1}{\text{Execution Time}}$$

- Throughput means the no of tasks complete in a unit time. Consider X & Y systems used to execute 10 tasks in 12 hours. X completes 10 tasks & Y completes 5 tasks.

$$\text{Performance} \propto \text{Throughput}$$

→ Performance Gain can be calculated using speed up factor(S). Speed-up factor compares 2 entities

$$S = \frac{\text{Performance of X}}{\text{Performance of Y}} = \frac{1/ET_X}{1/ET_Y} = \frac{ET_Y}{ET_X} = n$$

X System runs "n" times faster than Y system

$S = \frac{10}{5} = 2$; Speed-up factor = 2

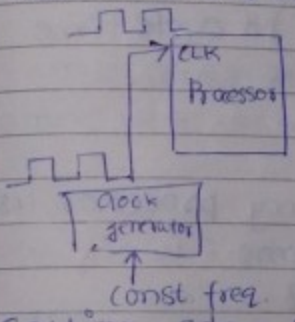
- $S = \frac{\text{Throughput}_X}{\text{Throughput}_Y} = "n"$

X system runs "n" times faster than Y

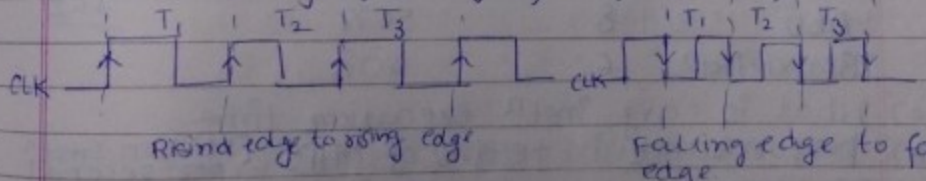
$\therefore S = \frac{10}{5} = 2$

CPU time calculation:-

CPU time - program execution time. It can be calculated based on the processor clock



Processor contains clock pin as an i/p pin, this pin is externally connected with clock generator. Clock generator is operating with constant frequency. It then generates the clock pulses. These clock pulses are carried into the processor through the clock pin. All the processor's activities are controlled by the clock. The time The time state is defined as rising edge to rising edge transition or falling edge to falling edge transition. It is also called as cycle, clock cycle, tick, clock tick, clock period etc



Cycle Time:-

The time required to transfer the pulse either from rising edge to rising edge or from falling edge to falling is called as cycle time. The cycle time depends on clock frequency. If 1GHz clock is used, then cycle time is

$$t = \frac{1}{f} = \frac{1}{1\text{GHz}}\text{ sec} = \frac{1}{10^9}\text{ sec} = 10^{-9}\text{s} = 1\text{ns}$$

CPU time = Program Execution Time =
no. of inst^n per program *
no. of cycles per inst^n *
no. of seconds per cycle

Or | CPU time = no. of cycles req. per program * cycle time |

If we know the no. of cycles req. for the program & the inst^n count, then we can compute CPI (Cycles per inst^n)

i.e | CPI = no. of cycles req. per program / Inst^n count |

| CPU time = CPI * Inst^n count * cycle time |

$$\text{CPU time} = \left(\sum_i CPI_i * IC_i\right) * \text{cycle time}$$

i :- type of Inst^n

Q. Consider 1 GHz clock frequency processor used to execute the following program:-

Inst^n	CPI	freq.
Load & Store	4	30%
ALU	8	50%
Branch Inst^n	6	20%

(a) What is avg. inst^n execution time.
(b) performance in terms of MIPS [Million Inst^n per second].
(c) If program contain 10^9 inst^n, what is the program execution time.

Ans. (a) 6.4 = avg CPI

cycle time $\frac{1}{10^9}$ s = 10^{-9} s

avg. inst^n execution time = 6.4 ns

(b) 1 inst^n → 6.4 ns

1 s → $\frac{1}{6.4} \times 10^9$

$$= 156.25 \text{ MIPS}$$

Date
Page No.

(c) 1 inst^n → 6.4 ns
10^9 inst^n → $6.4 \times 10^9 \times 10^{-9}$ s = 6.4 s

Q consider 1 ns clock cycle processor which consumes 8 cycles for load inst^n, 7 cycles for ALU op^n & 6 cycles for branch op^n The relative freq. of these inst^n are 30%, 40%, 30% respec An ideal processor With CPI = 1 & clock speed is 1.3 ns used to execute the program, what is the speed up using the ideal processor over another processor

ans 7×1 ns = 7 ns. → Processor 1

$1*0.3 + 1*0.4 + 1*0.3 = 1$ ns $= 1.3$ ns → Processor 2

$S = \frac{7 \text{ ns}}{1.3 \text{ ns}} = 5.3$

$3 + 2*6 = 15$

9.2 instn
cycle time $= \frac{1}{2.3} \times 10^{-9}$ s
$9.2 \times \frac{1}{2.3} \times 10^{-9}$ s

1 inst^n → 4 ns
1 s → $\frac{1}{4} \times 10^9$ inst^n
$= 0.25$
250 MIPS

1 inst^n → $\frac{9.2}{2.3} \times 10^{-9}$ s
1 s → $\frac{2.3}{9.2} \times 10^9$

9.2 cycles per inst^n
cycle time $= \frac{1}{2.3} \times 10^{-9}$ s.

avg. inst^n
$4 \times \frac{1}{2.3} \times 10^{-9}$
1 cycle
= 4 ns

High performance architecture exploit the concurrency.

→ Concurrency:- means two or more event execution is called concurrency, this event may be a subprogram, program, inst^n, stages in inst^n. concurrency implies to parallelism, simultaneous, pipelining.

→ Parallelism means two or more events are executed at the same time period (in range of time state i.e. T_1, T_2 state)

→ Simultaneous - two or more events are executed in the same time instance (in same time state, two initiation are req. for simultaneous)

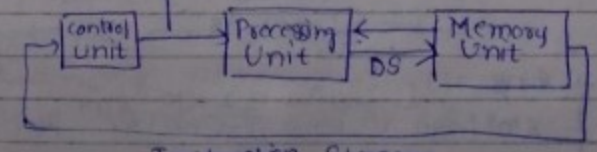
→ Pipeling - two or more events are executed in overlapping time span.

High Performance Architecture is classified in 4 types acc to Flynn's Classification:-

① SISD (Single Inst^n Stream Single Data Stream)
② SIMD (" " " Multiple " ")
③ MISD (Multiple " " Single " ")
④ MIMD (" " " Multiple " ")

(Uniprocessor System) **SISD :-** (ii) control unit generates control signals for processing unit

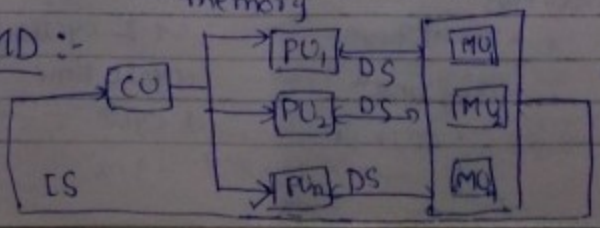
One inst^n executes at a time.



Sequential Execution

Instruction Stream
(i) Reads the inst^n from memory

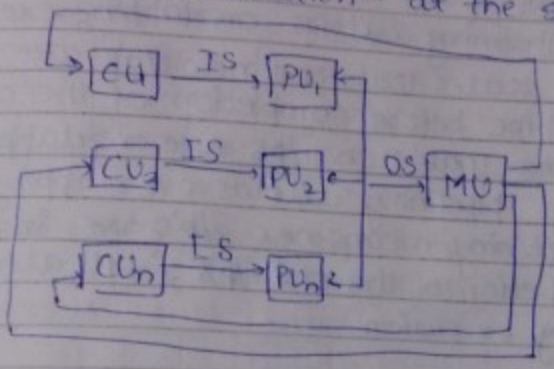
(Array Processor) **SIMD :-**



In this case same program is fetched from memory, but t

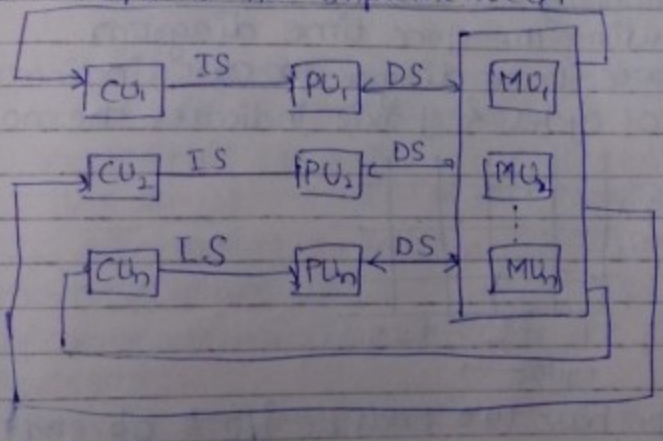
→ Simultaneous execution [same inst^n is performed at the same time]

MISD :-



→ Only one processor is in active state, because there is only one memory unit, so at a time only one processor can access the memory unit.
→ This concept is not implemented.

MIMD :-



→ It is parallelism.
→ Different inst^n are performed at the same time, ∴ parallelism.
→ To improve the SISD, we add pipelining concept

Pipelining

① One pipe output is connected to the input of another pipe.

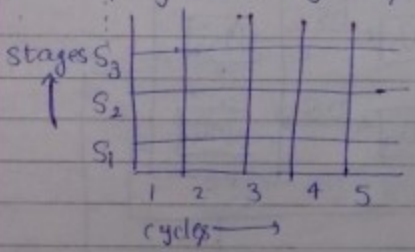
Date: Kitty
Page No.

Definition: Accepting new i/p's at one end before previously accepted input appears as an o/p at the other end. ∴ Pipelining allows overlapping execution.

Note: The def^n states that the new i/p's are inserted into the pipe before completion of the old i/p's, which means that new i/p's are overlapping with old i/p's, ∴ pipelining allows overlapping execution. Non-pipelining means new i/p's are inserted only after completion of the old i/p's, ∴ it allows non-overlapping execution.

⁂ $\boxed{CPI = 1}$

- Succesful characteristic of pipeline is for every new cycle new i/p must be inserted into the pipe, $\boxed{CPI = 1}$.
- Overlapping execution sequence can be represented by using the space time diagram.
- In space-time diagram, x-axis indicates the no of cycles & y-axis indicates the no. of stages.



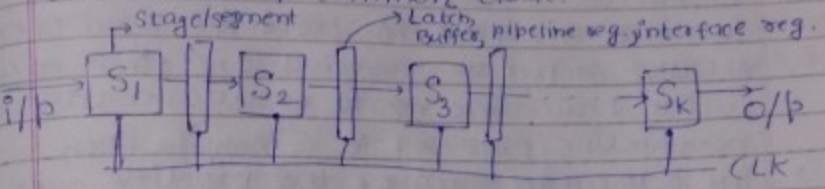
Designing of the pipeline: → Pipe has two ends, i.e i/p & o/p end. Bw i/p end & output end different pipes are interconnected with each other based on the function of the pipeline to satisfy the objective of the pipeline.

- Each pipe is called stage or segment. Bw the stages iterface registers are used to store the intermediate result. It is also called as pipeline register or buffer or latch.

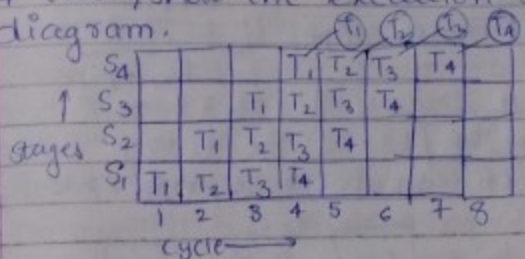
5 | 10 Kirty

Date:
Page No.

- All the stages along with the buffer are under the control of common clock.



→ Consider 4 segment pipeline used to execute the 4 task, show the execution seq. using space-time diagram.



Tasks: T_1, T_2, T_3, T_4

★ The 1st i/p to the pipeline requires k cycles to execute (k:- no. of segment).

→ The rest i/p's require 1 cycle to output

$\therefore$ $k + (n-1)$ [n:- no. of tasks.]

→ In non overlapping execution = $k \times n$

~~Non-overlapping~~

Performance evaluation of the pipeline processor:-

→ consider k segment pipeline with clock cycle time of tp. The very 1st inst^n (task) in the pipeline is executed in the non-overlapping order, so it requires k cycles to complete the op^n. The remaining $(n-1)$ tasks emerges from the pipe @ ~~one cycle~~ one task/cycle, $(n-1)$ cycles are req. to complete the remaining task, $\therefore$ the total time req. to

execute n task using k segment pipeline is
$ET_{pipe} = (k+n-1)t_p$.

- consider a non-pipeline processor used to execute n-tasks, each task requires t_n time to complete ∴ the total time req. to execute n tasks using non-pipeline processor is $ET_{nonpipe} = nt_n$
- The performance gain of the pipeline over non-pipeline is:-

$$S = \frac{\text{performance}_{pipe}}{\text{performance}_{non\text{-}pipe}} = \frac{\frac{1}{ET_{pipe}}}{\frac{1}{ET_{non\text{-}pipe}}}$$

$$\boxed{S = \frac{nt_n}{(k+n-1)t_p}}$$

- When the no of tasks increases, $n \gg\gg k-1$, ∴ $k+n-1 \approx n$

$$\therefore S = \frac{n t_n}{n t_p} = \boxed{\frac{t_n}{t_p}} = \frac{k t_p}{t_p} = \boxed{k}$$

$$\therefore \boxed{S = \frac{t_n}{t_p}}$$

- When all inst^n takes same no. of cycles, then non-pipeline inst^n execution time is equal to the no. of segments in the pipeline

$t_n = kt_p \Rightarrow S = kt_p/t_p = K$

$\therefore \boxed{S = K}$ → Pipeline Depth ☆

- When processor operates at 100% efficiency, then the max. speedup is pipeline depth, i.e.

$\boxed{S_{max} = k}$

- $\boxed{\eta = \frac{S}{S_{max}} = \frac{S}{k}}$ → pipeline efficiency

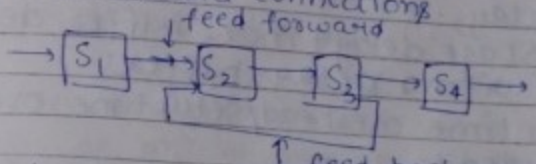
The throughput of pipeline is no of tasks processed per total time req to process the tasks.

i.e. $\boxed{\text{Throughput} = \frac{n}{(k+n-1)t_p}}$

Types of Pipelines

① Linear Pipeline :- This pipeline is used to perform only one function. (only feed forward)

② Non-Linear Pipeline - This can perform multiple functions. It can consist of feed-forward & feed-backward connections.

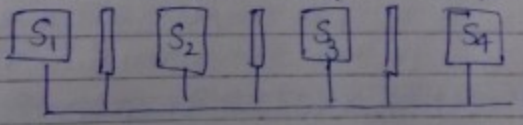


③ Synchronous Pipeline :- In this pipeline, the initiations are controlled by the clock. i.e. ~~the~~ P each stage need to complete its opⁿ with the clock cycle, i.e.

④ Asynchronous Pipeline :- initiations are controlled by handshaking signals, i.e. take the feedback of the stage before assigning the opⁿ to the next stage. [Not in use at the moment].

Hypothetical Pipelines

① Uniform Delay pipeline :- in this pipeline, all the stages completes the opⁿ with the same delay. 2ns 2ns 2ns 2ns



case ① : t_p = stage delay

case ② : If the stage takes 2ns, & the reg. takes 1ns delay, ∴ t_p = 3ns.

case ③ : t_p = stage delay + buffer delay

Case ①:- t_p = stage delay

Case ②:- If buffer delay is included, then
t_p = stage delay + buffer delay

Case ③:- If we have uniform stage delay but non-uniform buffer delay,
t_p = stage delay + max. buffer delay

~~Case ③:- If non uniform delay < buffers~~

Case ④:- If skew time overhead/setuptime overhead is included,
$t_p = t_p$ + skewtime/setup time overhead.

→ Non-uniform Delay pipeline:-
In this pipeline diff. stage delays are maintained blw the stages
① t_p = max. stage delay
② If buffer delay is included,
t_p = max. stage delay + buffer delay
③ If non-uniform buffer delays are used:-
t_p = max. stage delay + max. buffer delay.

*In Non-pipeline processor, buffer is not included.

4.04.12

Q1. An instⁿ pipeline which has the speed up factor 10, while Opⁿ with 80% efficiency what is the no. of stages in the pipeline.

$$0.8 = \frac{S_{10}}{k}$$

$$k = \frac{10}{0.8} = 12.5 \approx 13.$$

Q2. Consider 2 pipelines A & B where A is having 6 stages of uniform delay of 2ns. Pipeline B is having 5 stages w.r.t stage delays of 2ns, 1ns, 3ns, 2ns, 2ns. How much time is saved using pipeline A instead of pipeline B if 100 tasks are pipelined.

Ans. $t_{pA} = 2ns$

$t_{pB} = 3ns$

$ET_A = (6+99) \times 2 = 210\ ns$

$ET_B = (5+99) \times 3 = 312\ ns$

Time save 102 ns

Q3. Consider (4) segment pipeline w.r.t. stage delays 3ns, 5ns, 8ns, 2ns. The interface reg. are having the delay of 1 ns. What is the asymtotic speedup when very large no. of instⁿ are pipelined.

Ans. $t_p = 8+1 = 9ns$

$$S = \frac{t_n}{t_p} = \frac{36}{9} = 4$$

~~$t_n = 3ns+1ns+5ns+1ns+8ns+1ns+2ns+1ns = 24ns$.~~

Q4. Consider a non-pipeline processor. It takes 50 ns to complete the task. The same task can be processed by using 6 segment pipeline w.r.t. stage delays of 5ns, 8ns, 10 ns, 12 ns, 2 ns, 4 ns. What is the speedup factor.

Ans $S = \frac{n t_n}{t_p} = \frac{50}{12} = 4.16$

$S = \frac{n t_n}{(n+k-1) t_p}$ $\quad n >>> k-1$

Q5. Consider 4 stage pipeline where diff. inst^n are taking diff amount of time at diff stages. i.e. shown below:

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

$I_1 \to 3$
$I_2 \to 2$
$I_3 \to 3$
$I_4 \to 2$
→ Asynchronous pipeline

(a) How much time is req. to complete the inst^n?
(b) What is the speedup factor?
(c) What is the efficiency?
(d) What is the throughput?

Ans (a) $I_1 \to t_p = 4 \times 3 = 12$ ns

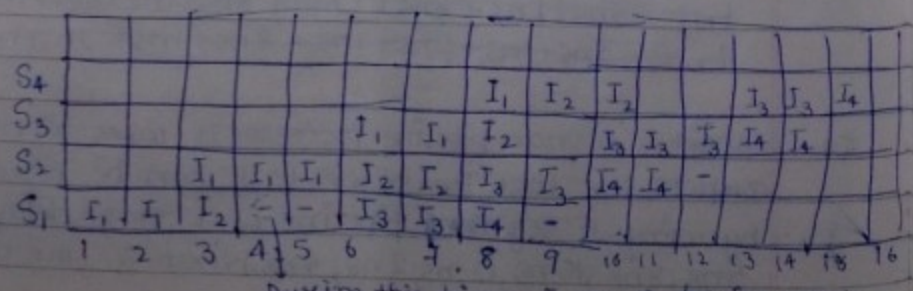
★★ In Asynchronous pipelining, before passing the inst^n to next segment, we checks whether the next segment is busy or not

	S_1	S_2	S_3
I_1	2	3	5
I_2	2	3	5
I_3	2	3	5

→ This is non-uniform delay.

★★ For Asynchronous pipeline, draw the space time diagram. [feedback op^n is performed via handshaking]

"—" indicates waiting on the same stage



During this time, I_2 can't be forwarded to S_2 as it is executing I_1; I_2 will wait in S_1 stage.

Date
Page No.

(a) 15 cycles are req. to complete the opⁿ.

(b) $S = \frac{t^n}{t_p} = \frac{8ns + 6ns + 9ns + 6ns}{15ns} = \frac{29}{15} = 1.9$

(c) $\eta = \frac{S}{k} = \frac{1.9}{4} = 0.47$

(d) Throughput = no. of tasks executed per unit time
$= \boxed{4/15}$

$t_p = 3ns$
$t_n = 7ns$

$t_p = (4 + 999) \times (165)$
$= 1003 \times 165\ ns$

$= 12\ ns$

$S = \frac{t^2}{k}$

$t_p = 12ns$
$S = \frac{36ns}{12ns}$

(15) 4×0.4
$1.6 + 0.8 + 2 = 4.4$
Avg. instn execution time = 4.4ns
ETpipe = CPI * (0.2 + 1) = 1.2 * 1 = 1.2ns
$\therefore S = \frac{4.4}{1.2} = 3.7$

(16) $n \times 2^9 \times 2^{10} \times 2^3 = 2 \times 2^{16} \times 2^{10} \times 2^5$
$n \times 2^{12} = 2^{16}$
$n = 2^4 = 16.$

(17) $S = \frac{318}{110}$ (100 + 10 = 110)
$11 \overline{)31}\ (2.8$
-22
90

(18) $\frac{50 \times 100}{(6+99) \times 10} = \frac{5000}{105} = 47.6$

(19) $\frac{25 \times 10^2}{10 \times 10} = 2500$
4
$6 \overline{)2500}$

(20) $0.8 = \frac{10}{k}$
$k = \frac{10}{0.8} = \frac{100}{8} = 12.$

(21) $(8+99) \times 2 = 107 \times 2 = 214\ ns$
$(8+99) \times 3 = 104 \times 3 = 312\ ns$

(29) $(t_p)(P_1) = (5+a9)4 = 109 \times 4 = 436$ ns

$(t_p)P_2 = (8+99) \times 2 = 107 \times 2 = 214$

time saved = 202 ns

Date: Kitty

Page No.

S_4					I_1	I_1			I_2		I_3	I_3	I_4	I_4
S_3	-			I_1	I_1		I_2	I_2	I_2	I_3	I_3	I_4	-	
S_2			I_1	I_2	I_2	I_2	I_3	I_3	-	I_4	I_4			
S_1	I_1	I_1	I_2	I_3	I_3	-	I_4	-						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(1)

Ans $S = \frac{29}{15} = 1.9$ (2)

$\eta = \frac{1.9}{4} = 0.475$ (3)

(28)

S_4					I_1	I_1	I_2		I_3		I_4	I_1	
S_3				I_1	I_2	I_2	I_3	I_3	I_4	I_4	I_1		
S_2		I_1	I_1	I_2	I_3	-	I_4	-	I_1	-			
S_1	I_1	I_2	I_2	I_3	I_4	I_4	I_1	I_1					
	1	2	3	4	5	6	7	8	9	10	11	12	13

I_1

$k = 1$ $k = 2$

(1)

Overlaping cycles → 7 to 11 (2)

(11)

S_4					I_1			I_2	I_2	I_3	I_3	I_3	I_4	I_4	I_1			I_2
S_3	-			I_1			I_2	I_2	I_3	-	I_4	I_4	-	I_1	-		I_2	I_2	I_3
S_2			I_1	I_2	I_2	I_2	I_3	-	I_4	I_4	I_1	-	-	I_2	I_2	I_2	I_3	-	I_4
S_1	I_1	I_1	I_2	I_3	I_3	-	I_4	-	I_1	I_1	I_2	-	-	I_3	I_3	-	I_4	-	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

(2)

12/6

I_2	I_3	I_3	I_3	I_4	I_4
-	I_4	I_4	-		
I_4					
20	21	22	23	24	25

Advanced Pipeline

RISC processor supports 3 categories of inst^n:-

① Load & Store : Load r0, 2(r1)
r0 ← M[2+[r1]]
Store 3(r1), r0
M[3+[r1]] ← r0

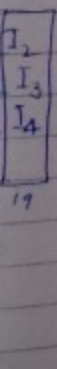
→ RISC processor is register to register architecture.

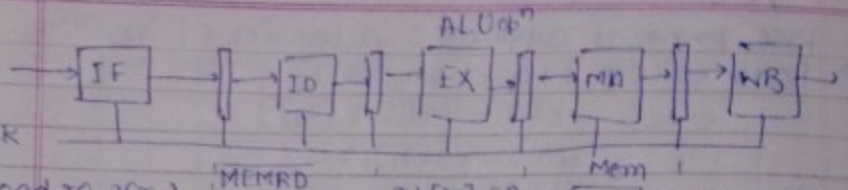
② ALU op^n Add r0, r1, r2
r0 ← r1 + r2

③ Transfer of Control:-
Unconditional TOC:- JMP 2000
PC ← 2000
Conditional TOC:- JNZ r0, 2000

RISC processor uses 5 stage pipeline to execute the inst^n :-

① Inst^n fetch (IF) :- In this stage, based on the PC, the
② inst^n is transferred from memory to CPU, at the end of this process, program counter is incremented to step size to point the next inst^n address.
③ Inst^n Decode (ID) :- 2 op^n are performed in this stage, i.e. decode the inst^n to identify the type of op^n & accessing the register file to read the operands.
This stage also contains comparator logic to evaluate the branch conditions, ∴, TOC inst^n execution is completed at the end of 2nd stage of the pipeline.
④ Execution (EX):- ALU op^n are performed in this stage.
⑤ Memory Access (MA):- Memory read & mem. write op^n are performed in this stage.
⑥ Write Back (WB):- Register file is updated in this stage.
This stage op^n is divided in 2 parts, in 1st part register file is updated & in 2nd part, reg. file is read to access the data.





① 1000: Load R0, 2(R1)
PC ← 1001

MEMRD
Source: Mem 2 R1
Destination register R0
We will get the value of R1

2 + [R1] = EA

the register value was available in 2nd stage.

EA

The value is stored in the latch & not in the reg

R0 ← xfng+t
R0 ← M[2 + [R1]]
↓
the content is transferred from latch to R0.

② 1001: Store 3(R2), R1
PC ← 1002

It is a mem write op^n
MEMWR
Source: register, R1
Destination: 3, R2
We will get the value of R1 & R2 in 2nd stage itself

3 + [R2] = EA
EA is calculated.

EA R1

EA was calculated in previous stage, & R1 is stored at EA

As no reg. write op^n is needed, nothing will happen in this stage.

③ Add R0, R1, R2

+ → add op^n is identified
Source: register R1 R2
Destination register R0
We will get value of R1 & R2 in this stage itself

R1 + R2

R1 & R2 is added & Stored in ALU register

This result is transferred to the mem. latch without accessing memory

R0 ← R1 + R2

④ 1000: JMP 2000
PC ← 1001

Unconditional JOC is identified in this stage,
PC ← 2000

⑤ 1000: JNZ R0, 2000
PC ← 1001

Cond. JOC is identified
Cond. :- JNZ
Source - R0
Target Add = 2000
CMP NZ R0
if R0 = NZ, then PC ← 2000
if R0 = NZ, then no change in PC

This is performed by comparator logic in ID stage

Date
Page No.

Dependencies in pipeline :-

1. Structural Dependency
2. Data »
3. Control »

→ Stalls in the pipeline
→ Extra cycle w/o opⁿ.

- Structural Dependency :- this is present in the pipeline due to resource conflict, the resource may be memory, register or functional unit.

	CC_1	CC_2	CC_3	CC_4	CC_5
I_1	Mem	ID	ALU	(Mem)	
I_2		Mem	ID	ALU	
I_3			Mem	ID	
I_4				(Mem)	

some resource

- In clock cycle 4, I_1 & I_4 are accessing (memory) at the same time, hence there is a resource conflict.
- Conflict is an unsuccessful opⁿ, so keep instⁿ ④ into the waiting state until the resource becomes available, this waiting creates the stalls.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7	CC_8
I_1	Mem	ID	ALU	Mem	WB			
I_2		Mem	ID	ALU	Mem	WB		
I_3			Mem	ID	ALU	Mem	WB	
I_4				/ /	/ /	/ /	Mem	ID

Stall

→ I_4 can't be fetched in CC_4, as I_1 is accessing the memory.

→ In this case CPI ≠ 1 because of presence of extra cycles due to dependencies present in the pipeline.

☆ ~~Due to~~ To minimize the stalls in the pipeline due to structural dependency renaming concept is used. eg in clock cycle 4, instⁿ 1 is accessing the memory to read the data & I_4 is accessing mem. to read the instⁿ, because instⁿ & data both are present in same mem. area

there is a conflict. Renaming implies that dividing the mem. in two modules & stores inst^n & data in diff modes i.e. code memory & data memory.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7	CC_8
I_1	CM	ID	ALU	DM	WB			
I_2		CM	ID	ALU	DM			
I_3			CM	ID	ALU			
I_4				CM	ID			
I_5					CM			

→ no stall cycles, CPI=1

CM - code mem.
DM - Data mem.
↳ when code & data are in diff modules of the mem.

- **Data Dependency:-**

Consider the program segment where inst^n j follows inst^n i in program.

i: Inst^n
j: Inst^n

j is data dependent on i when j tries to read the data before i writes it. This means one of the source in j inst^n is the dest^n in i inst^n.

e.g. I_1 : Add r0, r1, r2 → until completion of
I_2 : MUL r3, r0, r4 →

Data dependency exists b/w adjacent inst^n, i.e. it is known as true data dependency.

If Data Dependency exists b/w inst^n, keep dependent inst^n in waiting state untill the o/p of previous inst^n is available

e.g. Keep inst^n j in waiting state until inst^n 'i' execution is completed. This waiting creates stalls in the pipeline

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1: Add r0, r1, r2	IF	ID (Source r_1, r_2; Dest^n: r_0; r_1 & r_2 value will be accessed)	ALU ($r_1 + r_2$)	MA	WB ($r_0 \leftarrow$ ...)		
I_2: MUL r3, r0, r4		IF	ID (Source r_0, r_4; Dest^n: r_3; r_3 value)	///	///	ALU	

→ we can't get the value of r0, because it is dependent upon the result of I_1

To check whether the resources are dependent or independent resources.

Tomasulo Algo.

S/No	Function	Dest	IOP_1	IOP_2	DOP_1	DOP_2	Status
1	+	r0	r_1	r_2	-	-	1
2	*	r3	-	r4	r0(I_1)	-	

The r0 which acts as operand in I_2 is the destination for I_1.

~~If the Status~~

IOP - Independent Operand
DOP - Dependent Operand

	CC_3	CC_4	CC_5	CC_6
I_3	IF	///	///	ID
I_4				IF

★ Total 2 stalls.

★ I_3 remains in IF, because ID segment is occupied by the inst^n I_2.

In the above execution sequence in clock cycle 3, during the decoding process of I_2, the CPU recognizes that dependency exists b/w I_2 & I_1, so execution state is not allocated for I_2 until completion of I_1, so 2 stall cycles are created.

To minimize the stalls in the pipeline due to data dependency, operand forwarding mechanism is used. Operand forwarding is a h/w technique, it is also known as By-passing/short-circuiting.

→ Bypassing means one stage ALU o/p acts as the another inst^n ALU i/p.

This means before updating the result into the reg file, we can access it from the latch.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1	IF	ID	ALU	MA	WB		
I_2		IF	ID	ALU	MA		
I_3			IF	ID	ALU		
I_4				IF	ID		

I_1 :- Add r_0, r_1, r_2
I_2 :- MUL r_3, r_0, r_4 → only one data dependency because I_1's o/p acts as I_2's i/p.
I_3 :- DIV r_5, r_0, r_6
I_4 :- SUB r_7, r_0, r_8

no data forwarding in this case, because it will read the data from the reg. file

I_2 instn gets r_0 from the I_1 ALU's reg. rather than waiting & taking it after WB from reg. file.

* While execution of transfer of control instn, the program execution sequence is altered. Control dependency exists in the pipeline due to the TOC instn execution.

* **Unconditional TOC :-**

Consider the following program segment :-

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

Execution sequence :-

$I_1 - I_2 - I_3 - BI_1 - BI_2$

fall through
1000
1001
1002 : JMP

taken path 2000 : BI_1
2001

compute in ID stage

PC is incremented after inst^n fetch opⁿ.

The process of removing the unwanted inst^n is known as flush/freeze.

V.V Imp

★ If the target address is available at "kth" inst segment, then no. of ~~flus~~ stalls will be "k-1".

★ In RISC processor, the no. of stalls will be 1, as the target address is availble after the 2nd stage. [when only 1 branch inst^n is present.]

★★

★ Inst^n fetch is overlapping with inst^n decode, ∴ before decoding the previous inst^n, a new inst^n is inserted into the pipeline.

★ If the decoding stage decodes the inst^n as data transfer or data manipulation, the next inst^n is a wanted inst^n.

★ If the decoding stage decodes the inst^n as unconditional JOC, then the next inst^n is unwanted inst^n, the unwanted inst^n are flushed out from the pipeline. The process of removing unwanted inst^n is known as flush/freeze.

Date: Page No.

* The flush/freeze opⁿ creates the stalls in the pipeline. These stalls are known as Branch penalties
* Branch Penalty = At what stage the target addr is available - 1
* for RISC, branch penalty is always 1.
* The no of stall cycles created from the branch opⁿ = Branch freq. * Branch penalty

e.g.
Consider 4 stage pipeline, the target address is not available until completion of instⁿ, if program contains 30% unconditional branch instⁿ, how many no. of stall cycles are created from those branch opⁿ?

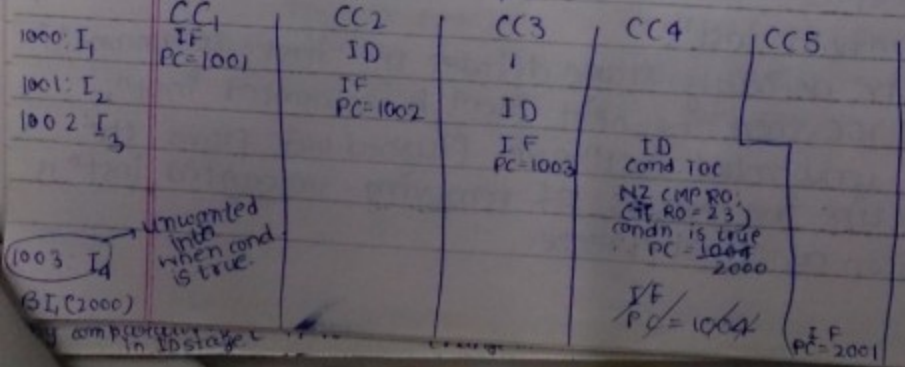
Ans. Branch penalty = 3
Stall cycles = 0.3 * 3 = 0.9

Conditional TOC:-

Consider the program segment:

1000:	I_1
1001:	I_2
1002:	I_3 (JNZ R0, 2000)
1003:	I_4
:	
2000:	BI_1
2001:	BI_2

Execution Sequence (Condition True):-
$I_1 - I_2 - I_3 - BI_1 - BI_2$
(False condition):-
$I_1 - I_2 - I_3 - I_4$



So, I_4 is flushed & a stall is created.

★ When the decoding stage decodes the inst^n as conditional TOC with true condition, then the next inst^n is unwanted inst^n, flushout the unwanted inst^n, otherwise (if it is false cond^n) the next inst^n is a wanted inst^n, so no need to flush.

Note:- If inst^n is conditional TOC, how many times the cond^n evaluates to true, only those many stalls are present.

e.g.

30% cond^n TOC
- 70% (true) → Branch Penalty
- 30% (false) → No branch penalty

$0.3 \times 0.7 = 0.21$

No. of stall cycles from cond. branch = Branch freq * Branch penalty

$= 0.3 * 0.7 * 3$ [previous ques]

$= 0.21 * 3 = 0.63$

★ Due to Unconditional TOC & conditional TOC, the pipeline doesn't operates at 100% efficiency.

① To minimize the stalls in the pipeline due to control dependency, branch prediction buffer (or) Branch target buffer is used.

② Branch target buffer is placed in the inst^n fetch stage

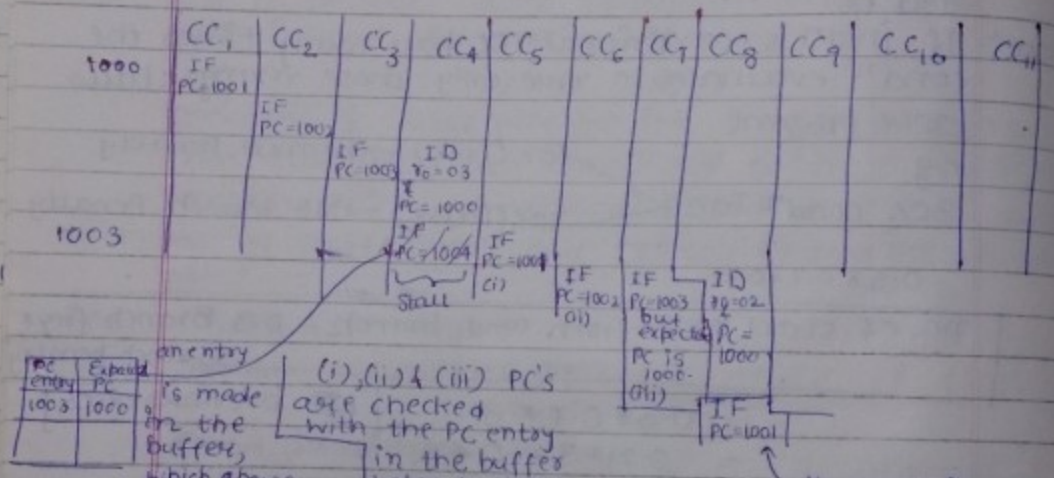
PC entry	Expected PC

★ No exact soln. for stalls in TOC.

★ In the PC entry, the next inst^n address after the TOC

inst^n is maintained. In the expected PC, the target address is maintained.

1000: I_1 : CL 04
1001: I_2 : ADD
1002: I_3 : JNZ r_0, 1000 ; let r_0 = 04
1003: I_4



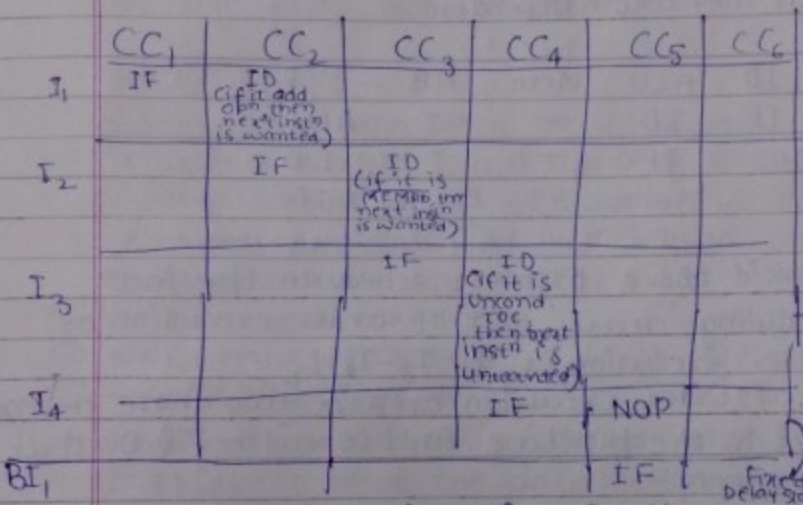
★ This concept is useful to execute the loop, in the loop execution, the stalls are present during 1st & last cycle of the loop. The objective of prediction buffer is to predict the TA before decoding.

Delayed Branches:-

This concept preserves the execution path in the pipeline while execution of transfer of control op^n.

→ Delayed Branch slot is associated with NOP op^n.

→ Unwanted inst^n slots are defined as the delayed slots, ∴ it substitutes the NOP op^n., so functionality is safe & not missing.



★ The delayed slot is defined in the 2nd cycle (ID). If the decoding stage decodes the inst^n as data transfer or data manipulation, then the delayed slot is forwarded to next cycle). If the decoding stage decodes the inst^n as unconditional TOC or conditional TOC with true cond^n, then the delayed slot is fixed, so NOP is substituted during the decoding process.

Scheduling

Consider following program segment:

I_1 : Add r_0, r_1, r_2
I_2 : MUL r_3, r_0, r_4
I_3 : Add r_4, r_5, r_6
I_4 : Add r_3, r_7, r_8

Date: Page No.

The processor executed the inst^n in in-order-execution i.e. $I_1 \to I_2 \to I_3 \to I_4$.

→ I_2 is data dependent on I_1, (I_3 & I_4) are independent inst^n.

☆ Scheduling causes the out-of-order execution.

→ I_2 undergoes waiting till the completion of I_1, ∴ I_3 & I_4 are also sharing the stall cycles even though they are independent.

I_1	IF	ID	ALU	Mem	WB
I_2		IF	ID	-	-	ALU
I_3			IF	-	-	ID
I_4				-	-	IF

To avoid above problem, schedule the inst^n, Scheduling causes out-of-order execution or reorder execution, i.e. $I_1 - I_3 - I_4 - I_2$

Out-of-order execution creates two more dependencies in the pipeline, Anti Dependency & Output Dependency.

as I_3 is executed before I_2, I_3 writes result in r_4 & I_2 have r_4 as i/p, then I_2 will read

☆☆① Consider the program segment where inst^n j follows inst^n i in the program order.

Anti-Dependency exists when the inst^n j tries to modify the registers before inst^n i reads it.

e.g.

inst^n I_3 modifies reg. r_4 before I_2 reads it.

☆☆ O/p dependency :- it exists when inst^n j tries to write the reg. before inst^n i writes it, ∴ the reg is loaded with old value.

e.g. inst^n I_4 writes the reg. r_3 before inst^n I_2 writes it.

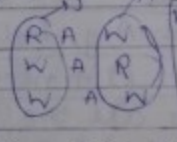
Kitty
Date:
Page No.

★ 2012 GATE Due to the registeo conflict, the Anti & o/p dependencies are present in the pipeline, reg. renaming concept is used to avoid the anti & o/p dependency, i.e use the temporary storage (reorder buffers) to store the o/p of the I_3 & I_4.
After receiving the exception signal from the inst^n I_2, the reorder buffer values are updated into the reg. file, ∴ data is safe.

Hazards :-

Hazard is created when the dependency exists in the pipeline. Hazard is a delay. Delay creates the extra cycles. Extra cycles are known as stalls.
Hazards are classified into 3 types based on the order of read & write op^n, i.e :- j , i

① RAW (Read after Write)
② WAR (Write after Read)
③ WAW (Write after Write)



When j performs these op^n before i performs these op^n, there is risk of hazard.

- RAW :- This hazard is created when the inst^n j tries to read the data before inst^n i writes it (True Data Dependency).
- WAR :- This hazard is created when the inst^n j writes the data before inst^n i reads it. (Anti-Dependency).
- WAW :- This hazard is created when the inst^n j writes the data before inst^n i writes it. (o/p dependency).

- **Performance Evaluation of pipeline with stalls**

① Speed up (S) = $\frac{\text{Avg. instn. execution time of non-pipeline}}{\text{Avg. instn. execution time of pipeline.}}$

Date
Page No.

(2) $S = \frac{\text{Avg. CPI}_{\text{Non-pipe}} * \text{Cycle time}_{\text{Non-pipe}}}{\text{Avg. CPI}_{\text{pipe}} * \text{Cycle Time}_{\text{pipe}}}$

(3) The ideal CPI of pipeline processor is always almost 1, so $\boxed{\text{avg. CPI}_{\text{pipe}} = \text{ideal CPI} + \text{no. of stall cycles/Instn}}$

$\therefore \boxed{\text{avg. CPI}_{\text{pipe}} = 1 + \text{no. of stall cycles/Instn}}$

under this -

$$S = \frac{\text{Avg. CPI}_{\text{Non-pipe}} * \text{cycle-time}_{\text{Non-pipe}}}{(1 + \text{no. of stall cycles/instn.}) * (\text{cycle time})_{\text{pipe}}}$$

Q. Assume that pipelining & non-pipelining circuits are perfectly balanced, i.e. no skew time / setup time overhead, pipeline & non-pipeline cycle times are equal.

$$\boxed{S = \frac{\text{Avg Instn FI}_{\text{Non-pipe}}}{(1 + \text{no of stall cycles/Instn})}}$$

When all the instn are taking same no. of cycles, then 1 instn execution time = the no. of stages in the pipeline.

Under this condition -

$$\boxed{S = \frac{(\text{no. of stages in pipeline})}{1 + \text{no of stall cycles/instn}}}$$

If there is no stalls, the processor is operating with 100% efficiency,

$\boxed{S = \text{Pipeline Depth} = K.}$

chp 1

(2)

Q12> $\frac{n}{(k+n-1)t_p} = \frac{1}{t_p} = 1$

Avg Execution Time = (1 + no. of stalls/Instn.) * Cycle time

$= (1 + 4 * 0.2) * 10\,ns = 18\,ns$

Date

05.05.12 V.V.Imp.

Date: Kitty
Page No.

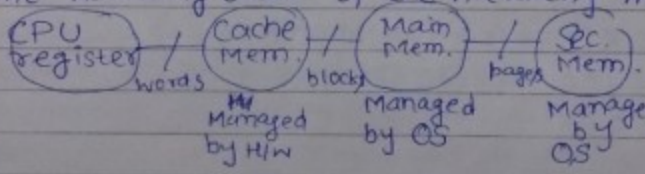
Memory Organisation

- **Cache Memory :-** Acc. to the memory hierarchy, the system supported memory standards are defined below:-

①	Level :	1	2	3	4
②	Name	Register	Cache	Main mm.	Sec Mem
③	Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
④	Access Time(ns)	0.25-0.5	0.5-25	80-250	50,00,000
⑤	Bandwidth(MB/sec)	20k-100k	5k-10k	1k-5k	20-150
⑥	Managed by	Compiler	H/W	OS	OS
⑦	Backed by	Cache (taking data from cache)	Main Mem. (Data to cache Mem. is taken from main mem.)	Sec. Mem.	CD
⑧	Implementation:	Multiport port	Onchip/offchip SRAM	DRAM (capacitor)	Magnetic

① Acc to the above standards, the memory hierarchy is designed to minimize the speed gap b/w the CPU & Memory.

② The Accessing order of the hierarchy mem. system is



- The data transfer b/w Sec. Mem. & Main mem. is controlled by OS.

- If CPU requires data, then it checks data in registers, if it is present in reg., then event is a hit, otherwise it is a miss.
- If the data is not present in reg, then it checks in the cache memory

- The data transfer b/w Cache Mem. & Main mem. is controlled by Computer Organisation.

- When CPU access the memory, it checks the availability of data in registers, if data is available, opⁿ is hit, then the req. data is transferred to ALU, otherwise opⁿ is a miss. When the opⁿ is miss in the register, then the next reference is given to the cache memory.
- If opⁿ is hit in cache, the data is transferred to the CPU in the form of words. If it is a miss in cache, the reference is forwarded to main memory.
- If opⁿ is hit in main memory, then the req. data is transferred to the cache in the form of blocks & to CPU in form of words.
- Miss in main memory, the reference is forwarded to Sec. memory. The opⁿ is always hit in sec. mem., ∴ the corresponding data is transferred to main mem. in the form of pages, transferred to cache mem. (via main mem.) in the form of blocks & to CPU (via Cache mem.) in the form of words.

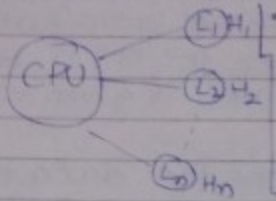
* Locality Of Reference :- The CPU performs read & write opⁿ only on the cache, the cache mem. maintaining the image of main mem. This property is called as the principle of locality of reference.

→ Based on the type of accessing, the mem. organisation is divided into 2 types :-

1. Simultaneous / Indebendent Mem. orgⁿ.
2. Hierarchial Mem. Orgⁿ.

→ Independent Mem :- In this orgⁿ, the CPU directly communicates with all the levels of

mem. directly, still it follows accessing order from level 1 to level n



- If data is not present in L_1, then reference is forwarded to L_2, if it is a hit in L_2, then data is transferred directly to CPU, w/o transfer of data from L_2 to L_1.

- Here $H_1, H_2, \ldots H_n$ indicates the hit ratio of corresponding levels of mem. Hit ratio = $\frac{\text{no of hits in mem}}{\text{total no. of accesses to that mem}}$

- **Hit ratio + Miss ration = 1**

- $T_1, T_2, \ldots, T_n$ are the accessing times of the respec. levels of mem.
- In this org^n, there is no communication establishes b/w the levels of memory
- If there is a miss in L_1, data is transferred from L_2 to CPU without involvement of L_1.
- If there is a miss in L_2, data is transferred from L_3 to CPU w/o involvement of L_2 & L_1.
-

The avg. access time of memory is

independent/simultaneous accessing time.

$$T_{avg} = H_1 * T_1 + (1-H_1) H_2 * T_2 + (1-H_1)(1-H_2) H_3 T_3 + \ldots + (1-H_1)(1-H_2) \ldots (1-H_{n-1}) H_n T_n$$

where $H_n = 1$

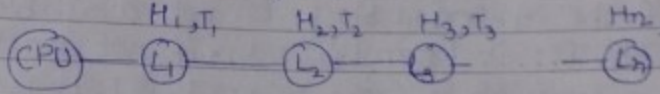
- T_{avg} - time req. to access one word from the memory.

1 word $\longrightarrow T_{avg}$

Efficiency of mem. system = $\frac{1}{T_{avg}}$ words/sec, no of words accessed in 1 sec.

Date / Page No.

↔ Hierarchial Mem. Orgⁿ :- In this orgⁿ, the CPU can perform read or write opⁿ only on the level 1 memory. If the opⁿ is miss in level 1, take the data from the higher levels into L_1, later the CPU can perform the read/write opⁿ on the L_1 memory, i.e.



Where $H_n = 1$.

Hierarchial Mem. System :-

T_{avg} (for hierarchial mem. system) =

$H_1 * T_1 + (1-H_1)H_2(T_1+T_2) + (1-H_1)(1-H_2)H_3(T_3+T_2+T_1) + \dots + (1-H_1)(1-H_2)\dots(1-H_{n-1})H_n(T_n+T_{n-1}+\dots+T_1)$.

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

$= H_1T_1 + H_2T_1 + H_2T_2 - H_1H_2T_1 - H_1H_2T_2$

$= H_1T_1 + T_1 + T_2 - H_1T_1 - H_1T_2$ [$H_2 = 1$]

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

Q1. In a 2-level memory, the Level 1 mem. is 5 times faster than level 2 & its access time is 10ns less than T_{avg}, let L_1 access time be 20ns, what is the hit ratio? (no hierarchy word, ∴ independent.

Ans. $30 = H_1 * 20 + (1-H_1)H_2 * 100$

$30 = H_1 20 + (1-H_1) \times 100$

$30 = 20H_1 + 100 - 100H_1$

$70 = 80H_1$

$H_1 = 7/8 \approx 0.9$

★ $S = \frac{\text{Performance of } L_1}{\text{Performance of } L_2} = \frac{1/\text{Access time of } L_1}{1/\text{Access time of } L_2} = \frac{T_2}{T_1}$

★★ If the question doesn't say that mem. is hierarchal, ∴ in that case, take it as independent.

The complete block must be transferred which comprises

Q2. 3 level memory has the following specifications:-

Level	Access time/word	Block size (words) in	Hit ratio
1	20	—	0.8 T_1 = 20 ns
2	50	2	0.9 T_2 = 100 ns
3	100	4	1 T_3 = 400 ns

If the referred block is not available in L_1, transfer it from L_2 to L_1, if not available in L_2, transfer it from L_3 to L_2 & L_2 to L_1, what is Tavg to access the block.

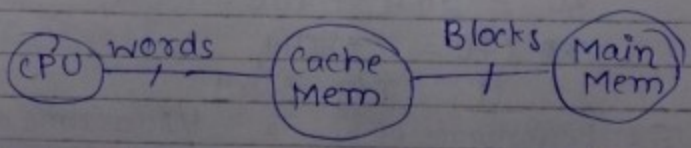
Ans.

$$T_{avg} = 0.8 \times 20 + 0.2 \times 0.9 \times (120) + 0.2 \times 0.1 \times (520)$$
$$= 16 + 24 \times 0.9 + 10.4$$
$$= 16 + 10.4 + 21.6$$
$$= 48 \text{ ns}$$

☆ We have to access the block, it takes 50 ns to access 1 word from level 2, so it will take 100 ns to access a block in L_2 (which is of 2 words length)

• Cache Memory:-

Cache Mem. acts as intermediate mem. b/w CPU & main memory. Therefore, CPU performs read & write op^n only on the cache data.



The basic elements of cache mem. is:-

① Mem. Org^n
② Mapping techniques
③ Replacement Algo.
④ Uptaking Techniques.
⑤ Multilevel Caches

★ 6a

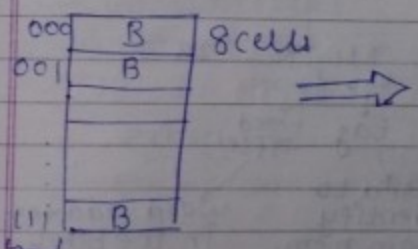
Date / Page No.

- **Memory Orgⁿ:-**
The data is transferred from the main memory to Cache memory in the form of blocks, ∴ the main mem. & cache mem. both are organised into blocks.

→ **Cache Mem. Orgⁿ:-**
The Cache Mem. is divided into equal parts based on the block size. Each part is called as Cache block or Cache line, ∴ No. of Cache blocks = $\frac{\text{Cache mem. size}}{\text{Block size}}$.

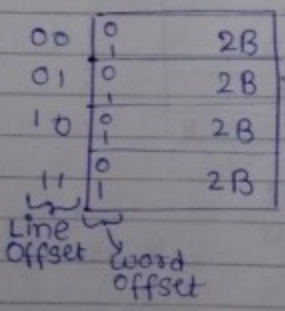
Consider 8 byte Cache mem. & 2 byte block size, no. of Cache blocks = $\frac{8}{2} = 4$

★ **Before Orgⁿ**
8 block Cache
8 byte Cache



After Orgⁿ

# of cache blocks = $\frac{8}{2} = 4$



★ No change in capacity but the internal Orgⁿ changes. (internal structure changes)

address of the block present in the Cache mem Orgⁿ. | 2 bit line offset | 1 bit word offset |

2 bit line offset → no. of blocks → 4 blocks
∴ size of line offset = $\log_2 4 = 2$

1 bit word offset → block size (no. of words in blocks) → 2
Size of word offset = $\log_2 2 = 1$.

3 bit Cache address is interpreted as 2 bit line offset & 1 bit word offset.

Date:
Page No.

* Before Main Mem. Orgⁿ :-

The main mem. is also divided into equal parts based on the block size. Each part is called as main mem. block. The main mem. block no. is identified by "tag". No. of main mem. blocks = $\frac{\text{main mem. size}}{\text{block size}}$

- consider 16 byte main mem. & 2 byte block size, the no. of main mem. blocks = $\frac{16}{2} = 8$.

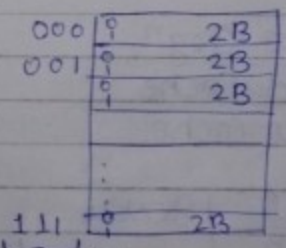
Before Orgⁿ

16 byte main mem.

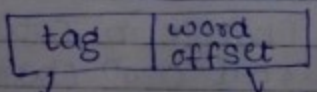
After Orgⁿ

no. of main mem. blocks = $\frac{16}{2} = 8$

8 * 2 byte = 16 B.



Physical Address



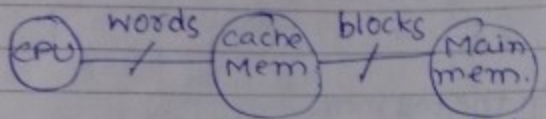
indicates main mem. block no.

$\log_2$(no. of main mem. blocks)
$= \log_2(8)$
$= 3$ bit

indicates word address in block

$\log_2$(Block size)
$= \log_2 2$
$= 1$ bit

★ In this org", there is no change in the capacity, but internal structure changes, ∴ the main mem. controller interprets the physical address as tag field & word offset field. The format is described earlier.



Block size = 2 byte

8 byte

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

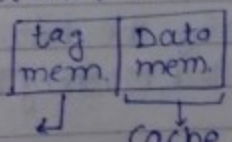
16 byte

000	B0
001	B1
	B2
	⋮
111	B7

LO	WO
2 bit	1 bit

Tag	WO
3 bit	1 bit

★ Cache controller :



extra bit in each cache line.

Cache (3 bit address cache addr.)

Q5. Consider the cache mem. size is 64 kB & block size is 64 bytes, if CPU generates 20 bit physical address, how many blocks are present in the cache mem & main mem?

Ans. Blocks in Cache = 1024

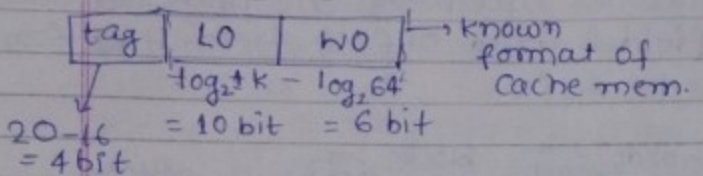
Blocks in main mem.

Mem size of main mem. = 2^{20} = 1 MB

Blocks in main mem. = $\frac{2^{20} \text{ Bytes}}{2^6 \text{ Bytes}} = 2^{14}$

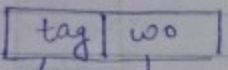
Date: Kirty
Page No.

★ The 20 bit physical address interpretation at cache controller is :-



↳ these tags are stored in the tag mem. in the cache controller.

★ 20 bit address interpretation at main mem. :-



$\log_2 2^{14} = 14$ bits $\log_2 64 = 6$ bits

Mapping techniques :-

The process of transfering the data from main mem. to cache mem. is called Mapping. There are 3 mapping techniques used :-

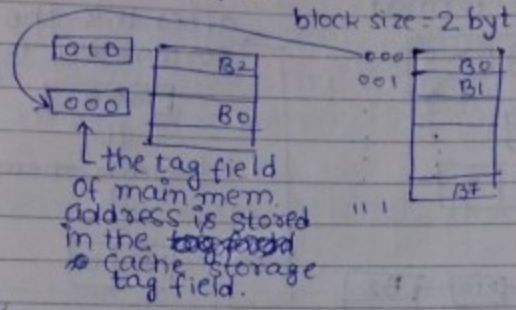
1. Associative Mapping
2. Direct Mapping
3. Set Associative Mapping

- **Associative Mapping** :- In this mapping technique, no mapping function is used, i.e. anyone of the main mem. block can be placed into anyone of the cache mem. lines, i.e. there is cache addressing is not used.
- The main mem. address format is used at the cache controller.

Date
Page No.

- The complete data along with tag is transferred from main mem. to cache mem.

Associative:- CPU / cache / MM
words 8 bytes blocks 16 bytes
block size = 2 byte.



B0 Main memory → No mem. func^n → Line 2 (Anyone of the cache line)
B0 Main mem.

* The address

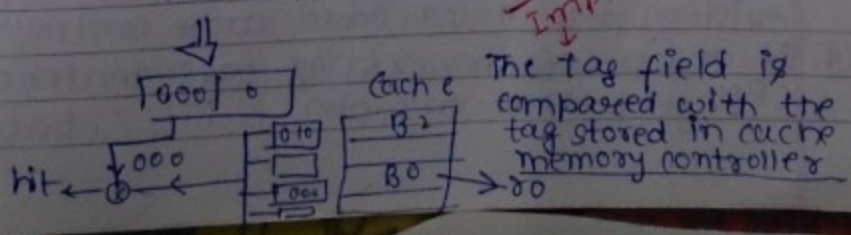
- Consider the program segment
  I_1: MOV r0, [0000]
  I_2: MOV r1, [1000]
  I_3: Add r0, r1

I_1:- CPU generates the Memory Address

[0000]

Associative cache known format is Main mem. format.

tag	W0
3 bit	1 bit



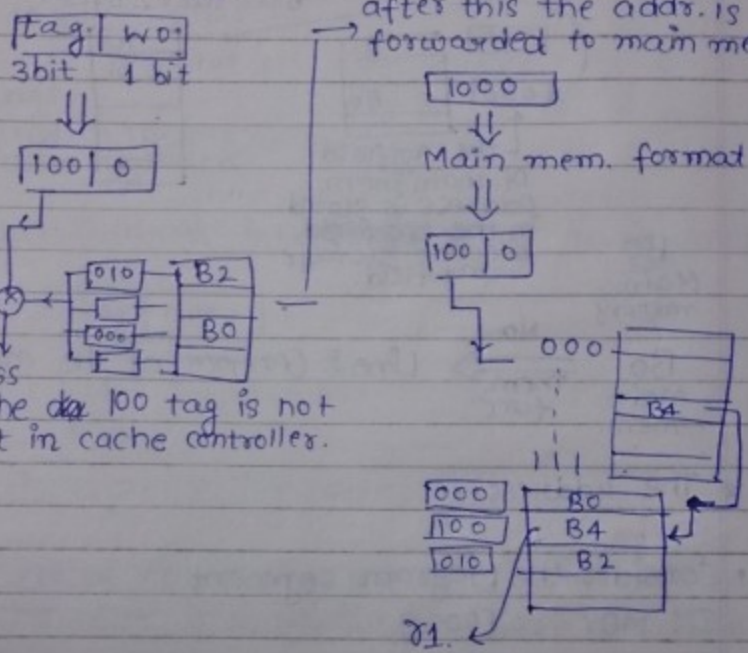
Date:
Page No.

I_2 :- CPU generates Mem. Addr.

1000

⇓

Associative & cache known format is main mem. format.



★ ① During execution of the program, whenever CPU identifies the mem. ref. inst^n, it generates the mem. address & transfer to the cache memory.

② The Associative Cache interprets the mem. addr. into its known format, i.e.

| tag | word offset |

③ The CPU generated tag is compared with the existing tag in the ~~cache~~ cache controller.

④ If anyone of the existing tags matches with CPU generated tag, o/p is hit, ∴ based on the

Date
Page No.

word offset, the respec. word is transferred to the destⁿ.

⑤ If none of the existing tags maches with the CPU generated tag, opⁿ is miss, so the ref. is forwarded to the main memory.

⑥ Acc. to the main mem. address format, the corresponding block maps into anyone of the cache line, later word is transferred to the CPU.

⑦ Tag mem. size = no. of cache lines * no. of tag bit at each line.

⑧ Data mem. size = no. of cache lines * no. of bits in one block

Tag mem. size = 4 * 3 = 12 bits
Data mem. size = 4 * 16 = 64 bits
Cache size = Tag mem. size + Data mem. size
∴ Cache size = 12 + 64 = 76 bits.

• Direct Mapping:-

① In this mapping technique, mapping function is used to transfer the data from the main mem. to cache mem.

② ∴ there is a binding that takes b/w the main mem. block no. to cache mem. line no. So, cache addressing is used.

③ The mapping function is K MOD N = i.

K:- main mem. block no. (Main mem. block no.)
N:- no. of cache lines (no. of cache lines)
i:- Cache mem. line no. (line no. in Cache)

Date: Page No.

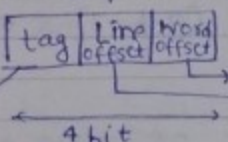
e.g.

0 MOD 4 = 0	4 MOD 4 = 0
1 MOD 4 = 1	5 MOD 4 = 1
2 MOD 4 = 2	6 MOD 4 = 2
3 MOD 4 = 3	7 MOD 4 = 3

imp.

① The above mapping function shows the reln. ship b/w the main mem. block no. to cache mem. line no., cache mem. addressing is used.

② The address format at the cache controller is



$\log_2$(Block size) = $\log_2 2 = 1$

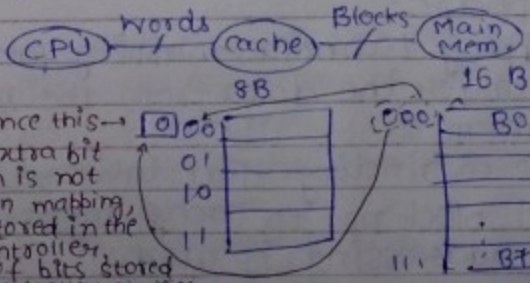
$\log_2 4 = 2$ bit

(*$\log_2$(no of blocks))

no. of main mem. blocks that are possible in each line = 2^(tag field size) = 2^1

② → this means that 2 blocks of main mem. blocks can be stored at one line in cache mem., in our example block 0 & block 4 can be stored in line 0, but not at the same time, i.e. only 1 block from main mem. can be stored in 1 line.

Direct



Since this is extra bit which is not used in mapping, it is stored in the cache controller, the no. of bits stored in the cache controller signifies that how many blocks from main mem. can be mapped into that particular line in cache (though only 1 block can be stored at a time).

Main Memory → Cache

B0 [000] → 0 Mod 4 → 0 (00)

* which makes the tag field at cache of 3 bits. The "0" extra bit is stored in cache controller.

In this case, we get 0 after doing 0 MOD 4, so the B0 from main mem. is stored in 00 (binary for 0) line no., now the tag field is 2 bit (00), the extra bit '0' is also stored*

Date: Page No.

*① During mapping, complete data is transferred from main mem. to corresponding cache line along with the tag.

② During this process, some of the tag bits maps into the cache mem. address logic & remaining bits are stored in the cache controller.

→ Consider program segment :-

I_1 : MOV R0, [0000]
I_2 : MOV R1, [1000]
I_3 : Add R0, R1

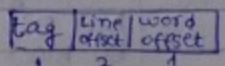
I_1 CPU generates mem. address

⇓

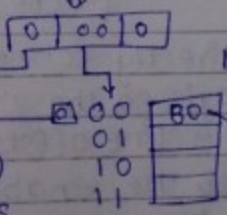
0000

⇓

Direct Cache known format



⇓

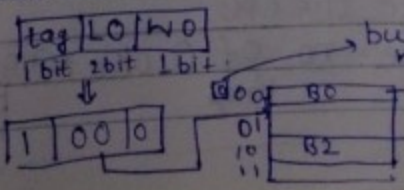


I_2 CPU generates mem. address.

1000

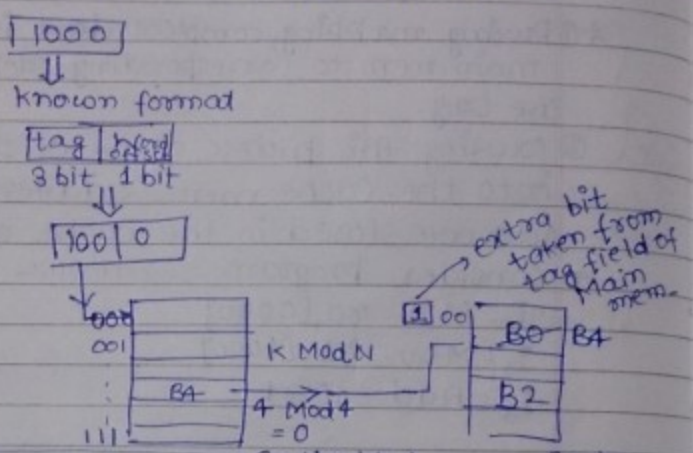
⇓

Direct Cache known format



① Compulsory Miss
② Capacity Miss
③ Conflict Miss

So, the address is forwarded to Main mem.



① CPU generated address is transferred to the direct cache.

② The direct cache interprets the req. based on its known format.

tag	L0	W0

So, the block B4 is transferred to the 0th line no. of cache mem. [0 → decimal, 00 → binary.] but 00 contains B0 block, so it is overwritten in the cache mem. by B4.

③ The line offset is directly connected to the cache memory address logic, so the corresponding cache line is enabled w.r.t. line offset.

④ The existing tag of the enabled line is compared with CPU generated tag.

⑤ When both matches, opⁿ is hit. Then data is transferred to CPU based on the word offset.

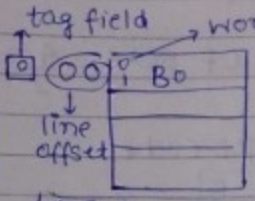
⑥ When both doesn't match, opⁿ is miss, so map the respective block from main mem. to cache memory based on the tag, later the data is transferred to the CPU.

★ Tag mem. size = no. of cache lines * no. of tag bits in each line.

Date: Page No.

The disadvantage in the direct mapping is only 1 block is possible in each Cache line at a time, i.e. conflict Miss increases.

Prog. I_1 MOV r0,[0000]
I_2 MOV r1,[1000]
I_3 MOV r2,[0001]
I_4 MOV r3,[0011]
I_5 Add r0,r1
I_6 Add r2,r3



★ I_5 & I_6 do not have to do anything with cache as it is just reg. to reg. opⁿ, mem. is not involved.

(i) During I_1, there will be a hit.
(ii) During I_2, the ~~100 14~~ 4 MOD 4 = 0, but 0 line no. contains B0 & not B4, so there will be a miss & B4 need to be transferred to line no. 0, now line no. 0 contains B4.
(iii) During I_3, 0 MOD 4 = 0, but when tag field is compared, it doesn't matches, ∴ it will be a miss & B0 will be transferred from main mem. to cache.
(iv) During I_4, 4 Mod 4 = 0, but when tag field is compared, it doesn't matches & there will be a miss & B4 will be transferred from main mem. to cache.

★ To avoid the above problem, there is a need of alternate cache organisation which is used to hold multiple blocks in the same line at the same time. This alternate approach is called as set-associative Cache Organization.

Set Associative Mapping:-

In this mapping, each cache line can hold the multiple blocks, i.e. the Cache mememory is organised into no. of sets based on the no. of blocks present on the same line.

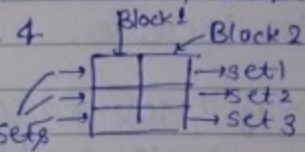
No. of sets S = $\frac{N}{P\text{-way}}$ (N: no. of cache lines), where N indicates no. of cache lines, P indicates no. of blocks in the same line..

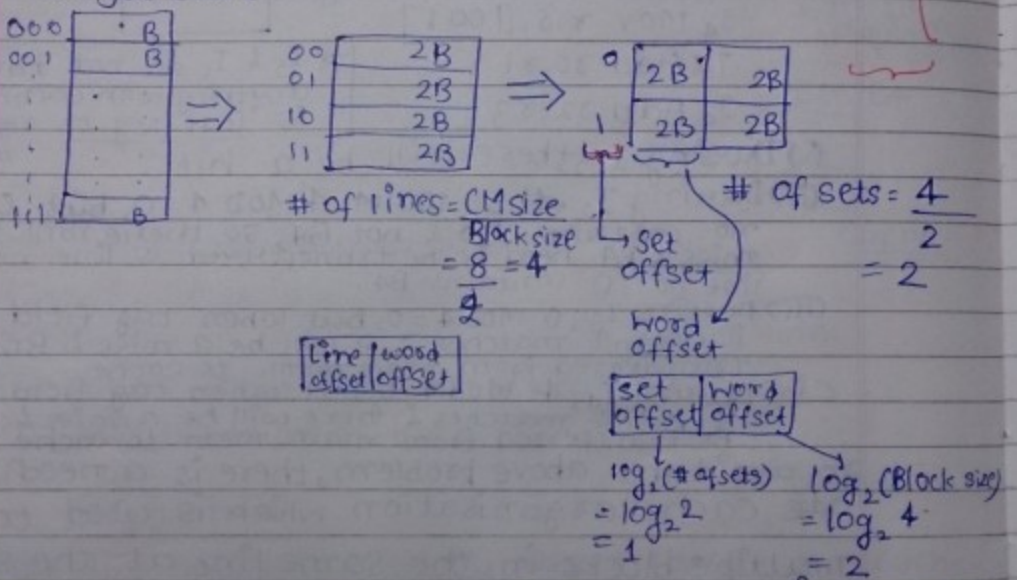
Date: Kitty
Page No.

• Consider 8 byte cache & 2 byte block using 2-way set associative cache orgn. The no. of sets is:-

no. of cache lines = $\frac{8}{2} = 4$

∴ no. of sets = $\frac{4}{2} = 2$.





★ Too many sets → fully associative cache memory.

★ To little sets → direct cache memory.
∴ 2 way & 4 way set associative cache gives the best result in the cache design.

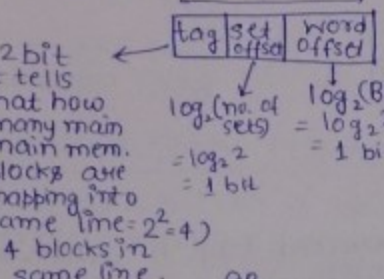
Mapping fuction:- In this technique, mapping fn is used to transfer data from main mem. to cache mem, i.e. there is a binding provided with the main mem. block no. to cache mem. line no.

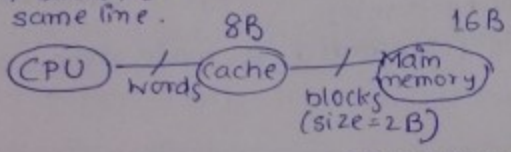
② The mapping fn. is k MOD S = i, where k:- main memory block no., S:- no. of sets in cache, i:- cache memory set no.

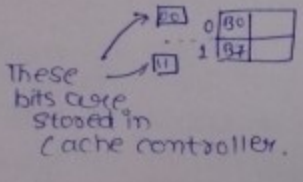
0 mod 2 = 0 4 mod 2 = 0
1 mod 2 = 1 5 mod 2 = 1
2 mod 2 = 0 6 mod 2 = 0
3 mod 2 = 1 7 mod 2 = 1

The above mapping fn. shows the relnship b/w main memory block no. & cache mem. set no., cache mem. addressing is used.

- The physical address can be interpreted at set associative cache controller is :- ⟵ 4-bit (physical address)







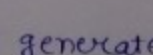
Main Memory
B0 [000] 0 mod 2 = 0
B7 [111] 7 mod 2 = 1

Cache
0 → this is the set no. where B0 will be stored
1 → this is the set no. where B7 will be stored

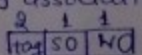
During this mapping process, the complete data along with tag is transferred to cache mem. Some of the tag fields maps into cache mem. address logic & remaining in Cache controller.

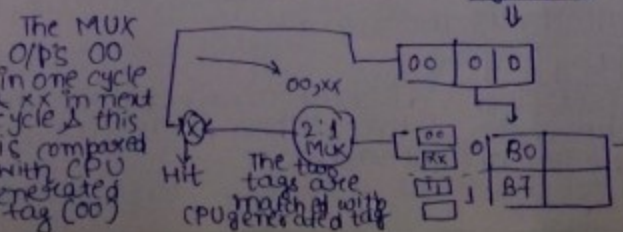
e.g. I_1: MOV R0[0000]
I_2: MOV R1,[1000]
I_3: Add R0,R1

I_1: CPU generates the Mem. address.



2-way associative cache known forma is





I_2

CPU generates Mem. Address

1000

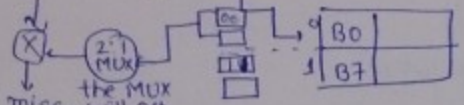
⇓

2-way set-Associative cache known format

tag	S0	W0

⇓

10	0	0



miss

the MUX will o/p 00 in 1st cycle & xx in next cycle & these o/p's will be compared with CPU generated tag '10', which doesn't matches, hence it will be a miss.

then, the address is forwarded to main mem.

1000

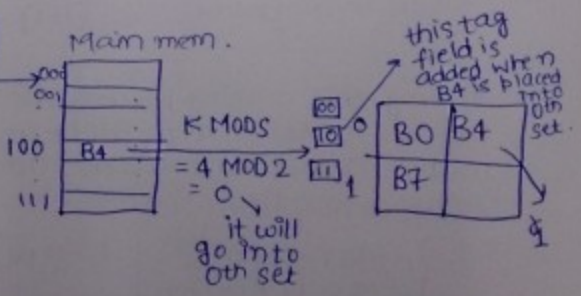
⇓

Main mem. known format

tag	W0
3	1

⇓

100	0



① The CPU generated mem. req. is initially referred in the cache mem.

② Acc to 2-way set-associative address format, the cache controller interprets the req. i.e.

tag	S0	W0

③ The Set offset field is directly connected to the addr. logic of cache mem., ∴ the respec. cache line is enabled based on the set offset value.

④ The enabled set contains 2 possible tags, ∴ MUX is req.

to compare these tags one after another to the CPU generated tag. If anyone is matching op^n is hit, the respec. data is transferred to the dest^n based on the word offset value.

⑤ When none of the tags are matching, op^n is Miss, then map the respec. block into the respec. set using the mapping function. Later the data is transferred to the CPU.

The tag mem. size = no. of cache lines (or no. of sets) * no. of blocks in each set * no. of bits in one tag.

If some more bits are there, then we will add it in tag field.

Types Of Cache Misses

There are 3 types of Cache Misses possible:-

① Compulsory Miss/First reference miss/cold-start :-
The very first reference to the cache is itself a miss, called as compulsory miss.
e.g. When Cache is initially empty.

② Capacity Miss:-
Because of the limited capacity of Cache, during the execution of the program, some of the cache blocks are replaced & later retrieved.

③ Conflict Miss/Collision/Inference Miss :-
Too many blocks when mapped on the same line is known as conflict miss.
e.g.
Direct mapping & set associative mapping creates conflict miss.

Types Of Locality :-

Based on the way of accessing reusable data, the locality of reference is divided into 2 types:-

① Temporal Locality. → same word in same block can be referenced by CPU in near future.
② Spatial Locality. → adjacent word in same block can be referenced by CPU in near future
③ Sequential "

Temporal Locality means the same word in same block can be referenced by CPU in the near future.

e.g. I_1: r0,[0000]
I_2:
I_3:
I_4: MOV r0,[0000]
I_5:
I_6:
I_7: Mov r0,[0000]

Spatial locality means the adjacent words in the same block can be referenced by CPU in near future.

e.g. I_1 MOV r0,[0000]
I_3 Mov r0,[1000]
I_5 MOV r0,[0000]

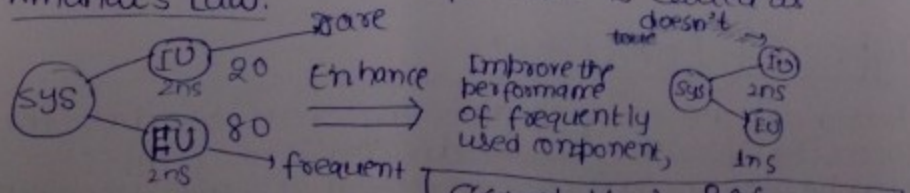
Replacement Algorithm :-

When the cache is full, then replacement algorithm are req. to replace the cache block with new blocks.
There are 2 replacement algo used :-

① FIFO (First in First Out) :-
In the FIFO, replace the cache block which is having longest time stamp.

② LRU (Least Recently Used)
In LRU, replace the cache block which is having less no of references along with the longest time stamp.

- Quantitative principle to improve the performance
Quantitative principle states that improve the performance within the cost & price limit by making the Common Case Fast. This principle is called as Amdhal's Law.



ETprog = 40ns + 160ns
= 200ns

S (speed up performance gain) = Performance of enhanced sys / Performance of old system.

Amdhal's law focus on performance gain after the enhancement.

$$S = \frac{\frac{1}{ET_{new}}}{\frac{1}{ET_{old}}} = \frac{ET_{old}}{ET_{new}} \quad -①$$

ET_{new} = $ET_{unenhanced\ portion}$ + $ET_{enhanced\ portion}$

ET_{new} = ET_{1U} + ET_{FO}

To calculate the ET-new, there is a need of two factors, i.e.

① fraction enhance (F)
② Speedup enhance (S)

- F indicates how much portion of system undergoes enhancement
- Speedup enhance indicates how many times the enhanced portion is running faster than old portion.

$$S = \frac{\text{performance of new portion}}{\text{performance of old portion}} = \frac{1/\text{Exec}^n \text{ time of new portion}}{1/\text{Exec}^n \text{ time of old portion}}$$

→ of Floating Point Unit (as it is the only unit that undergoes enhancement.)

$$S = \frac{(F)ET(Old)}{(F)ET(New)}$$ → of frequent case only.

ET_{new} = $ET_{unenhanced\ portion}$ + $ET_{enhanced\ portion}$

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

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

$ET_{old}(F)$ → enhanced portion (decrease of execution time in the enhanced portion.)

$ET_{old}(1-F)$ → unenhanced portion (no change in its execution time)

Let $ET_{old} = 1$

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

- $ET_{new} < 1$
- $S_{overall} > 1$

ET_{old} :- of complete system

★ If system contains multiple frequent cases :-

$$S_{overall} = \left[\left(1 - \sum_i F_i\right) + \sum_i \frac{F_i}{S_i}\right]^{-1}$$, i :- no. of frequent cases

A syste

Q. A system consists 2 functional units, i.e. Floating point & Integer which is used in mathematical model simulation. The FP unit is enhanced (it is running 2 times faster) but actually 10% inst^n are floating point, what is the performance gain.

Ans. $F = 0.1$ $\quad 2 = \frac{ET_{old}(F)}{ET_{new}(F)}$

$1 - F = 0.9$

$S = 2$

$ET_{new\ Soverall\ (complete\ system)} = ET_{unenhanced} + ET_{enhanced}$

$= 0.9 \times ET_{old} + \frac{ET_{old}(F)}{2} = 0.9 \times ET_{old} + \frac{0.1 \times ET_{old}}{2}$

$= 0.9$

$$S_{overall} = \left[(1-F) + \frac{F}{S}\right]^{-1} = \left[0.9 + \frac{0.1}{2}\right]^{-1} = 1.05$$

☆ When rare case undergoes enhancement, there is no considerable gain.

Q. 3 enhancements are proposed for a new architecture with following speedups:-

$S_1 = 30$

$S_2 = 20$

$S_3 = 15$

If enhancement 1 & 2, each usable of 25% of time, how much % of the time enhancement 3 will be used when the Soverall = 10.

Ans. $$10 = \left[1 - (50+x) + \frac{25}{30} + \frac{25}{20} + \frac{x}{15}\right]^{-1}$$

$$10 = \left[1 - (50+x) + \frac{250 + 375 + 20x}{300}\right]^{-1}$$

$$10 = \left[1 - (50+x) + \frac{625 + 20x}{300}\right]^{-1}$$

$10 = [1 -$

15 | 30, 20, 15
2 | 2, 20, 1

$\frac{1}{0.37}$

$0.7x$

$\frac{ET_{old}}{ET_{new}}$ $\quad \frac{ET_{mm}}{0.7 \times 10 \times ET_{mm}}$

$0.7 \times 10 \times ET_{mm}$

$0.7 ET_{mm} + 0.3 \times ET_{mm}$

$\frac{10}{0.37 ET_{mm}}$

$0.8 \times 4 t$

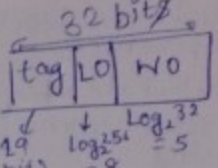
0.2×10^6

$3.2 t$

2000

Chapter-5

Ans3. Cache Size = 8KB
Block Size = 32B
no. of blocks in cache = $\frac{8 \times 2^{10}}{2^5} = 256$



no. of blocks in main mem. = 2^{32}

32

0 MOD 256 = 0

8 bits

24 bits + 2 bits

26 bits

256 × 26 = 1536 + 512 = 6656

(19+2) = 21 × 256 = 5376 bits

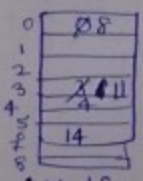
Ans1. S = 10
F = 0.7
Amdhal's law

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

$= \left[(1-0.7) + \frac{0.7}{10}\right]^{-1} = 2.7$

System with ~~cast~~ cache runs 2.7 times faster than sys. w/o cache.

Ans2.



no. of cache lines = 8

2-way
no. of sets = $\frac{N}{P\text{-way}} = \frac{8}{2} = 4$

① + ① + ① + ① + ① + ① + ① + ①

7 × 8 = 56

~~0 Mod 2~~

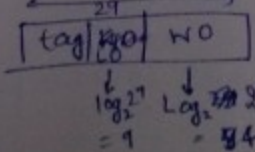
0 mod 4 = 0
14 mod 4 = 2

③ + ① + ① + ① + ①

= 7 × 10 = 70

~~Ans7.~~

Ans6. no. of cache lines = 2^9
block size = 2^4 Bytes
mem. size = 2^{29} Bytes



$0.8 \times 4ns + 0.2 \times (1msec + 4ns)$

16×2^9

Q Consider 4 block cache, with following main memory block references.

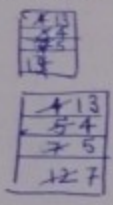
4, 5, 7, 12, 4, 5, 13, 4, 5, 7

assume cache is initially empty, identify hit ratios by using

(a) FIFO
(b) LRU
(c) Direct Mapped Cache (Associative)
(d) 2-Way set Associative with LRU.

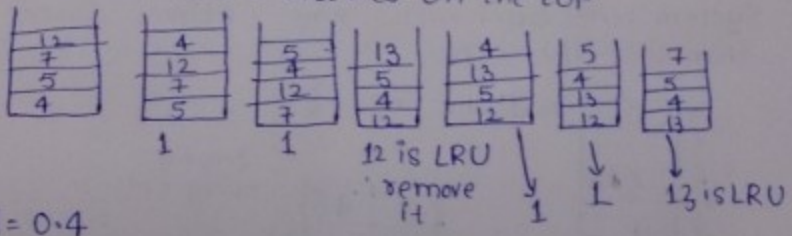
Ans. Total = 10

(a)



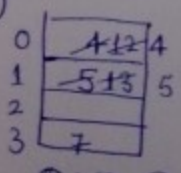
$H = \frac{2}{10} = 0.2$

(b) keep all recent entries on the top



H = 0.4

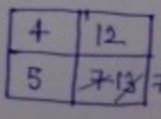
(c)



4 MOD 4 = 0
5 MOD 4 = 1
7 MOD 4 = 3
12 MOD 4 = 0
4 MOD 4 = 0
13 MOD 4 = 1
4 MOD 4 = 0
5 MOD 4 = 1
7 MOD 4

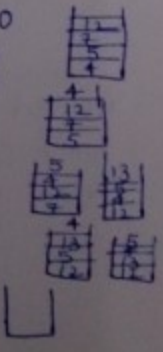
H = 0.3

(d)



4 MOD 2 = 0
5 MOD 2 = 1
7 MOD 2 = 1
12 MOD 2 = 0
13 MOD 2 = 1

H = 0.4



Ans1.

$\frac{80 \times 4}{10} = 32$ ns

$T_{avg}(old) = 80$ ns

$T_{avg}(new) = 112$ ns

$S = \frac{T_{MM}}{T_{CM}} = 9$

$T_{MM} = 9 \times T_{CM}$

~~$80 = 112 = H_1 \times T_{CM} + (1-H_1) \times 1 \times 9 \times T_{CM}$~~

$80 = 0.7x + 0.3 \times 1 \times 9 \times x$

$80 = 0.7x + 2.7x$

$80 = 3.4x$

$x = 21.62$ ns

$112 = H_1 \times 21.62 + (1-H_1) \times 1 \times 9 \times 21.62$

$112 = 21.62x + (1-x) \times 194.58$

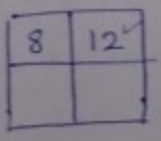
$82.58 = 172.94x$

$x = \frac{82.58}{172.94}$

194.58 − 112.00 = 82.58

15
21.62
× 9
194.58
21.62
21620
194.58
− 21.62
172.94

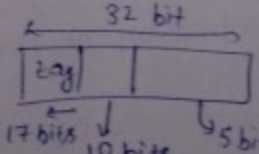
Ans8.



①+①+①

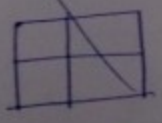
$\frac{2^5 \times 2^{10}}{2^5} = 2^{10}$

Ans10.



Ans11. Block size = 64 byte → One time accessing it generate 48 bytes, 1 time

Ans12.



no. of Set lines = $\frac{32 \times 2^{10}}{32 \times 2} = 2^9$

Ans11. 1 time accessing it generates = 48 bytes

1 time accessing time = $\frac{K}{I}$ ns + 80 ns

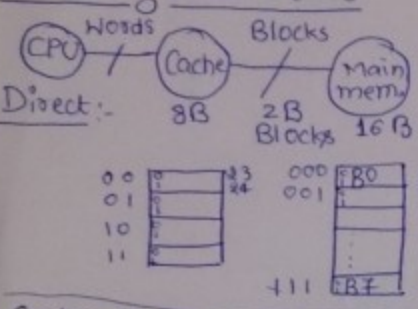
= 92 ns

Because of 64 byte cache line, there is a need of 2nd time accessing.

To transfer 64 B from MM to cache Mem, it req. 2 × 92 = 184 ns.

Date
06.05.12

Updating Techniques



I_1 : MOV r0, #23 ; r0=23
I_2 : MOV r1, [0000] ; r_1 = 23
I_3 : Add r0, r1 ; r0 ← r0+r1, r0=46
I_4 : MOV [0000], r0 ; CM[0000]=46

I_2 :- is a hit opⁿ, so the content will be available from cache. (Cache Mem.)
I_4 :- is a write hit opⁿ, so the data is transferred or written to cache.

Cache Memory - [0000] = 46
Memory (Main) [0000] = 23

When the same address contains diff. values at diff. places (i.e. Main mem. & Cache mem.) then it is called "coherence".

① The same address contains two diff. values at 2 diff. places is called "coherence".
② The coherence causes the loss of data.

I_5 : MOV r0, [1000] : Cache mem. block 0 [B0] is replaced by block B4.
I_6 : -
I_7 : - (updated block → updated data is lost)
I_8 : MOV r1, [0000] : r_1 = 23 (old value)

During the execution of I_5, the block 4 is transferred from the main mem. to cache mem. line 0. The cache mem. line 0 contains the updated block.
② Due to the conflict miss, the updated block is replaced with new block, updated data is lost. After sometime if CPU wanted to access the updated data which is nowhere present.
e.g.
During execⁿ of I_8 CPU want to read the updated value present in the block 0, because of replacement, the updated data is lost, the old data is reflected.

To avoid the above problem, there is a need of updating techniques. There are 2 updating techniques used, named as :-

① Write through
② Write back

Write through :- ① The CPU performs read & write opⁿ only on the cache. When the CPU performs read opⁿ, check the availability of the block in the cache. If it is available opⁿ is read hit, then CPU reads the data from the cache.
② If block is not available in cache, opⁿ is read miss, ∴ there is a need of read allocate, i.e. transfer the respec. block from main mem. to cache mem. for the read opⁿ.

Write Opⁿ :- ① When CPU performs write opⁿ, it checks the availability of block in cache mem.

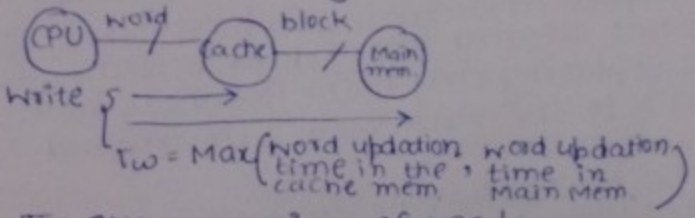
② If the block is available, opⁿ is write hit, so the CPU performs the simultaneous write opⁿ in the cache mem & main mem.

③ If block is not available, opⁿ is write miss, ∴ there is a need of write allocate, i.e. transfer the respec block from main mem. to cache mem. for the write opern later, the CPU performs simultaneous updation in cache & main memory.

Inclusion Property :-

Inclusion means the low level memory data is always subset of the high level memory data, i.e. low level mem. must be a write through memory.

Inclusion Property is not applicable for write back.



The avg. access time of read :-

$T_{avg(read)} = H_R T_C + (1-H_R)[T_M + T_C]$

(H_R: read hit; T_C: read data; $(1-H_R)$: read allocate; $T_M + T_C$: read data (from cache))

$T_{avg(write)} = H_W T_W + (1-H_W)(T_M + T_C)$

(H_W: write hit; T_W: because of simultaneous write opⁿ; $(1-H_W)$: write miss; write allocate; $T_M + T_C$: simultaneous write opⁿ [max. time for write opⁿ in each mem.])

e.g. T_{access} of L_1 = 10ns, T_{access} of L_2 = 20ns ∴ T_W = 20ns

The avg. access time of write through mem. when considering both read & write opⁿ is :-

$$T_{avg_{WT}} = F_R T_{avg(read)} + F_W T_{avg(write)}$$

($T_{avg(write\ through)}$; F_R: freq. of read opⁿ; F_W: freq. of write opⁿ)

• The efficiency of the write through is :-

$$\frac{1}{T_{avg(WT)}} \text{ words/sec}$$

Note: ① In the write through mechanism, there is no coherence at all.

② Write through mechanism is also applicable in independent mem. organisation.

$T_{avg(write)} = H_W T_2$ $H_W = 1$ (for independent)

★★ If H_W is not given, then $H_W = 1$, then it is independent
If H_W is given, then H_W = given value, it is hierarichal.

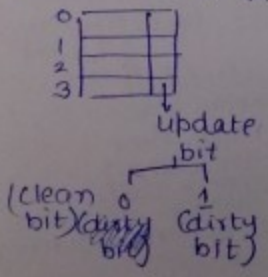
If write through mechanism consists, Hit ration for write opⁿ is 1, then it is independent mem.

Write Back Technique:-

In this technique, the CPU performs the write opⁿ only on Cache, which implies coherence is still present, but this coherence is not creating loss of data problem because before replacing any cache line, the updated data is write back to the main mem. ∴ the updated data is safe.

Write Back Cache maintains one extra bit at each cache line called Update bit, if it is "1" (Dirty bit) it means that the block is updated, otherwise (clean bit) → the block is not updated.)

Before replacing the cache lines with new blocks, read the % of dirty bits, take those updations into main mem., later replace the block.



When the CPU performs the read opⁿ, it checks the availability of blocks in the cache, if it is available, opⁿ is read hit, then CPU directly read the data, otherwise opⁿ is read miss, ∴ there is a need of read allocate, which implies transfer the block from main mem. to cache mem. for read opⁿ So there is a possibility of replacement, ∴ before read allocate, take the updations back

the main memory. After read allocation, CPU directly read the data from the cache.
The avg access time of read cycle is:-

$T_{avg(read)} = H_R T_C + (1-H_R)[\%\text{ of dirty bits}(T_m + T_M + T_C) + \ldots$

H_R: read hit; T_C: read data from cache; $(1-H_R)$: read miss; % of dirty bits: no. of updations to be made in main mem.; T_m: write back time (time taken to update blocks in main mem from cache); T_M: read allocate (time taken to put the req. block from main mem. to cache); T_C: read data

$\ldots \%\text{ of clean bits}(T_M + T_C)]$

% of clean bits: no updations; T_M: read allocate; T_C: read data from cache

When the CPU performs the write op^n check the availability of block in cache, if it is available, if it is available op^n is write hit, then CPU performs the write op^n only on cache (still coherency is present).
Otherwise, op^n is write miss, ∴ there is a need of write op^n. Due to write allocate, there is a possibility of replacement, so before write allocate take the updations back into the main mem. The avg. access time of write cycle is:-

$T_{avg(write)} = H_W T_C + (1-H_W)[\%\text{ of dirty bits}(T_M + T_M + T_C) + \%\text{ of clean bits}(T_M + T_C)]$

($(1-H_W)$: write miss; % of dirty bits: updations)

H_W: write hit; T_C: write data; % of clean bits: no updation; T_M: write allocate; T_M: write back; T_C: write data; write data; write allocate

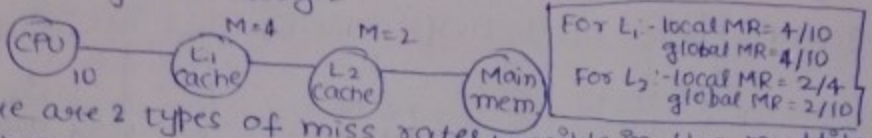
The avg access time of write back cache when considering both read & write op^n is:-

$$T_{avg(WB)} = F_R T_{avg(read)} + F_W T_{avg(write)}$$

$$\text{efficiency} = \frac{1}{T_{avg(WB)}}$$

Multilevel Caches

To reduce the miss penalty, multilevel caches are used in the system design



There are 2 types of miss rates possible in the multilevel caches :-

① Local Missrate = no. of misses in the cache / no. of accesses to that cache

② Global Missrate = no. of misses in the cache / total no. of CPU generated references

- Avg. access time of memory is :-
- T_{avg} = Hit Time$_{L_1}$ + Miss Rate$_{L_1}$ * Miss Penalty$_{L_1}$
- Miss Penalty$_{L_1}$ = Hit Time$_{L_2}$ + Miss Rate$_{L_2}$ * Miss Penalty$_{L_2}$ (Miss Rate$_{L_2}$ → local miss rate)
- Miss Penalty$_{L_2}$ = Main Mem. Access Time

The no. of mem. stalls/instⁿ = Misses in L_1/instⁿ * Hit time$_{L_2}$ + Misses in L_2/instⁿ * Miss Penalty$_{L_2}$

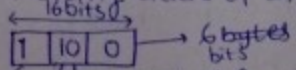
Chapter-5

⑰ ~~2500~~ $\frac{2500}{64}$ = 3

No. of blocks req. to store the array = $\frac{2500}{64}$ = 40 blocks

Physical Addr : 16 bit

Starting addr of an array = 1100H



5 bits

00010 00 0100 000000
tag LO WO

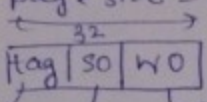
0	B28			
1	B29			
2	B30			
3	B31			
4	B0	~~B32~~	B0	B32
5	B1	~~B33~~	B1	B33
6	B2	B34	B2	B34
7	B3	B35	B3	B35
8	B4	~~B36~~	B4	B36
9	B5	~~B37~~	B5	B37
10	B6	~~B38~~	B6	B38
11	B7	~~B39~~	B7	B39
12	B8	40 miss	8 miss	8 miss
13	B9			
⋮	⋮			
31	B27			

56 miss

Ans 20.

CM size = 64 KB
Block size = 16 B
no. of blocks = $\frac{64 \times 2^{10}}{2^4} = 2^{12}$

Main mem. size = $2^{32} \times 4$ KB = $2^{32} \times 2^{12}$ B = 2^{44} B
page size = 4 KB



No. of blocks = 2^{12}
No. of sets = $\frac{2^{12}}{2} = 2^{11}$

17 bit
$\log_2$(no of sets) = 11 bit
$\log_2$(block size) = $\log_2 16$ = 4 bit

17×2^{12} bit
$2^{10} \times 17 \times 4 = 68$ KB

8 bytes $\times 2^{20}$
$= 2^{23}$ bytes

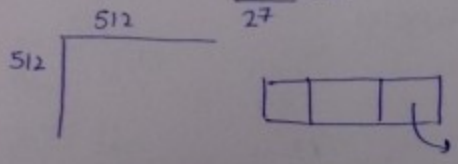
0xff000

11

one array element size = 8 B
one block size = 16 B
one block holds 2 array elements
one row req. 512 blocks

Ans 14.

CM size = 32×2^{10} B
Block size = 128 B
no. of blocks = $\frac{2^{15}}{2^7} = 2^8$



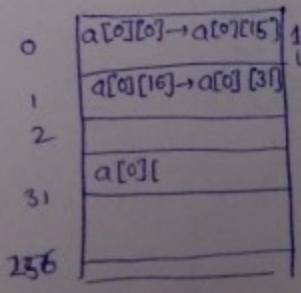
1 row → 512 elements
32 blocks
32 miss opn
512 rows → $32 \times 512 = 2^5 \times 2^7 = 2^{12}$

no. of blocks to store $512 \times 512 = \frac{512 \times 512 \times 8}{128} = \frac{2^{21}}{2^7}$
$= 2^{14}$ blocks

each block contains → $\frac{128}{8} = 16$ elements

$\frac{512}{16}$
$\frac{2^9}{2^4} = 2^5 = 32$

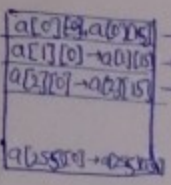
$\frac{512 \times 512}{16} = \frac{2^{18}}{2^4} = 2^{14}$ blocks



1 miss
15 hits

32
32
32
128

No. of misses = 16384



a[256][0] → a[256][15]

a[511][0]

512 miss → To access 1 column

To acess 512 columns → 512 * 512 miss

$\frac{2^{19}}{2^{18}} = \frac{1}{16}$

Ans 24. L_1 :- local = $\frac{40}{1000}$ = 0.04

Global = $\frac{40}{1000}$ = 0.04

L_2 :- local =

25. Tavg. = 1 + 0.04 * (10 + 0.5 * 100)

= 1 + 0.04 * (10 + 50)

= 1 + 60 × 0.04

= 1 + 6 × 0.4 = 3.4

26. Stall = Misses in L_1/instn. * Hit time of L_2 +

Misses in L_2 /instn. * Miss penalty of L_2

1 instn req. → 1.5 mem ref.

1000 mem. ref → $\frac{1}{1.5} \times 1000$ instn

Stall = 40/667 * 10 + 20/667 * 100

= 3.59 cycles ≈ 3.6 cycles.

27. 64 Bytes

2^6 Bytes

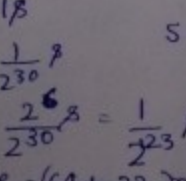
2^{30} Bytes → 1 s

1 byte → $\frac{1}{2^{30}}$ s

64 bytes → $\frac{2^6}{2^{30}}$ s = $\frac{1}{2^{23}}$ s or $\frac{64}{10^9}$ s = 64 ns

total time = (64 + 32) ns = 96 ns

FBFC / 512



31 | B27

Ans28. ~~16x 4W~~

4 x 20 = 80ns

2 L_1 - Miss
20 L_2 - Hit (4W)
2 L_1 - Hit
20 L_2 - Miss Hit (4W)
2 L_1 - Hit
20 L_2 - Hit (4W)
2 L_1 - Hit
20 L_2 - Hit (4W)
~~L_1 - Hit~~
~~L_2 - Hit (4W)~~
~~88ns~~

L_1 - Miss
L_2 - Miss
MM - Hit (4W)
L2 - Hit
L_1 - Hit (but miss as complete 16 words are not transferred)
L_2 - Miss
MM - Hit (4W)
L_2 - Hit
L_1 - Hit (but miss also)
L_2 - Miss
MM - Hit (4W)
L_2 - Hit
L_1 - Hit (but miss also)
L_2 - Miss
MM - Hit (4W)
L_2 - Hit
~~Hit Hit~~

800
+ 80
+ 8

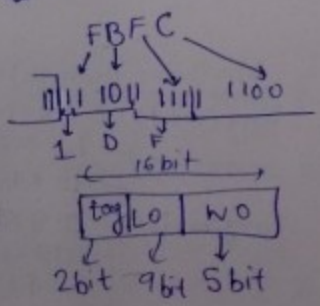
Ans30. set lines = $\frac{16}{4}$ = 4 set lines

0	~~048~~	~~132~~	~~8~~	~~216~~ 92
1	1	133	129	73
2				
3	255	3	159	63

$\frac{255}{4}$

34. no. of blocks

1011
1100



Ans31. ~~xx 2^8 x 2^10~~ = ~~xxx~~ $2^8 \times 2^{10}$

$x = 8$

Ans32.

0	~~0~~	16
1	5	~~9~~
2		
3	~~535~~	7

①+①+①+①

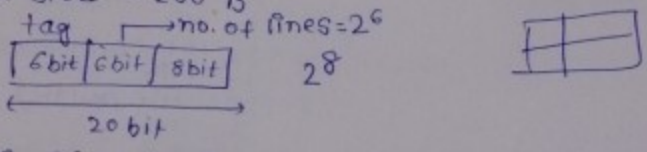
33.

0	~~808~~	12
1		

Ans 37. $T_{avg} = 0.9 \times 50 + 0.1 \times [500 + 50]$
$= 45 + 0.1 \times 550$
$= 45 + 55$
$= 100$ ns.

Ans 36. Lock up free cache :- used to hold outstanding miss information, it is also called as miss information status handling register.

Ans 38. Main mem. size $= 2^{20}$
line size $= 256$ B



2^8

no. of blocks in main mem. $= \frac{2^{20}}{2^8} = 2^{12}$

Ans 39. K MOD S = i

V

S :- no. of sets
K - ~~1set~~ lines in 1 set

Ans ~~10~~ 43. T_{avg} (write through)

T_{avg} (read) $= 0.8 \times 20 + 0.2 \times (200 + 20)$
$= 16 + 0.2 \times (240)$
$= 16 + 48 = 60$ ns

T_{avg} (write) $= 0.9 \times 100 + 0.1 \times (200 + 100)$
$= 18 + 24$
$= 42$ ns 120 ns

~~$T_{avg(WT)} = 64 \times 0.7 + 42$~~

$T_{avg(WT)} = 60 \times 0.7 + 120 \times 0.3$
$= 78$ ns.

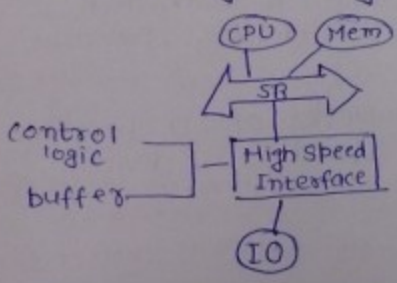
$\therefore$ Efficiency $= \frac{10^9}{78}$

Ans 44.

Input-Output Configuration

① I-O devices are very slow devices, ∴ they are not directly connected with the CPU.
② High speed interface logic is used b/w the CPU & I/O devices.
③ High speed interface logic is req. because I/O devices are electromagnetic devices & CPU is electronic device, ∴
   ① there is a diff. in operating modes.
   ② there is a diff. in word format.
   ③ there is a diff. in data transfer rate.

To avoid all the above problems, high speed interface logic is used. All the I/O devices are connected to the CPU through the high-speed interface.



High speed interface consists control logic & buffer.

① High speed interface control logic receives the CPU generated request & interpret it. Based on the address & control signal, the high speed interface enables the corresponding device for opⁿ, after preparing the data, the device transfers it to the buffer.
② When the data is present in buffer, the high speed interface generates the request to the CPU & waits for the acknowledgement.
③ After receiving ACK from CPU, the high speed interface transfers the data with high rate, ∴ speed gap is minimized.

Types Of Interfaces :

- Non-programmable (configuration is pre-defined during the design time) eg buffer, latch
- Programmable (configuration is defined in the command word register (CWR)) like that in 8255

Programmable

- Serial Interface (Bit-by-bit) eg 8251 USART
- Parallel Interface (more than 1 bit) e.g. 8255 PPI, 8259 AIC, 82377 / 8257 → DMA

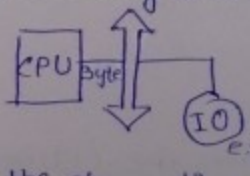
I/O Transfer Modes :-

There are 3 I/O transfer modes used to transfer the data from I/O devices to CPU/memory :-

1. Programmed IO
2. Interrupt Driven IO
3. DMA

Programmed IO :-

1. In this mode, no high speed interface logic is used, which implies I/O devices are directly connected with CPU, ∴ processor utilization is inefficient.
2. Processor undergoes waiting until completion of IO operation. This waiting time depends upon the speed of IO device.



e.g. 10 KB/s

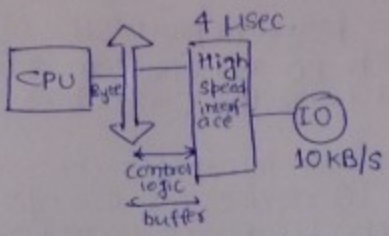
In the above diagram 10 KB/s is connected to the CPU in the programmed IO mode, if data transfer is byte level, the CPU requires

$10\ KB \longrightarrow 1\ s$

$1\ B \longrightarrow \frac{1}{10^4} = 10^{-4}\ s$ (or 100 μs)

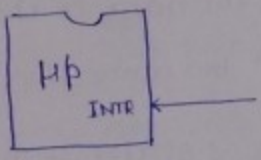
Interrupt Driven IO :-

1. In this mode high speed interface logic is used b/w the CPU & basic IO devices, ∴ processor utilization is good.
2. This implies that CPU is not directly communicating with IO, it only communicates with high speed interface, ∴ executing time depends on the latency of the high speed interface.



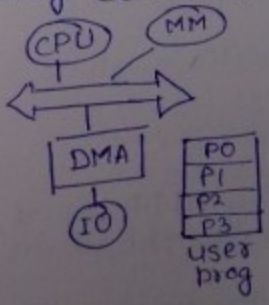
20KB
4μsec + 1B + 4μsec =

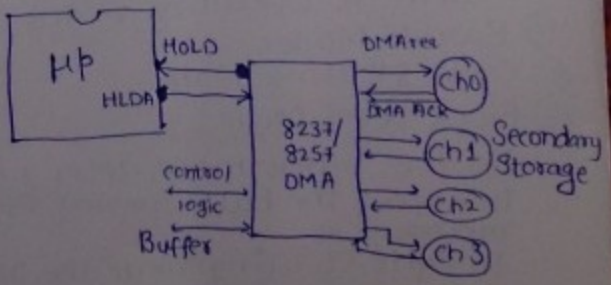
When the CPU want to read/write 1 byte data from the 10KB/S device which is connected to the CPU in the interrupt mode, it requires 4μsec.



DMA(Direct Memory Access):-

1. In this mode the bulk amount of data is transferred from the I/O device to main memory through the DMA W/O involvement of the CPU
2. DMA is a module, all the secondary storage devices are connected through the DMA.
3. When the user program capacity is greater than the main memory capacity we the virtual memory concept to increase the address space.
4. This implies that store the user program into the secondary storage device.
5. During execution of user program





① During the execution of user program part 0, the CPU initializes the DMA along with IO address, Mem. address, control signals & count values. Later, the CPU is busy with user program.
② Simultaneously, DMA control logic interprets the req. & enables the corresponding IO device for the opⁿ. After preparing the data, the IO device transfers it to the buffer present in the DMA.
③ DMA enables the HOLD signal to gain the control of the bus & waits for the acknowledgement.
④ After receiving the ACK, it transfers the data to the main memory until the count becomes 0.
⑤ After the opⁿ, the DMA reestablish the bus connection to the CPU.

→ During DMA opⁿ, CPU may be in 2 states, i.e. busy state & blocked/hold state.
→ Until preparation of data, the CPU is busy & until transfer of data, CPU is blocked.
→ The preparation time depends on the speed of device (disk) & transfer time depends on the latency of the memory.
→ Suppose, prepⁿ time denoted by x
& transfer " " " y
then % of time CPU is busy = $\left(\frac{x}{x+y}\right) \times 100$

% of time CPU is blocked = $\left(\frac{y}{x+y}\right) \times 100$

→ DMA is operating under 3 modes:-
① Burst Mode
② Cycle stealing Mode
③ Block Level Mode

In the burst mode, after receiving HLDA signal, DMA transfers the bulk amount of data to the main memory.
In cycle stealing mode, the DMA forcefully suspends the CPU & take the control of the bus & transfers very imp. data to the main memory.
Block level mode is intermediate b/w burst & cycle stealing. In this mode data is transferred blockwise