Class-1 Computer Organisation

Syllabus

Module 1: computer functionality
Module 2: computer components
module 3: Fixed point (vs) floating point formats

Books
① William Stanling
② Moris Mano

Ref. Books

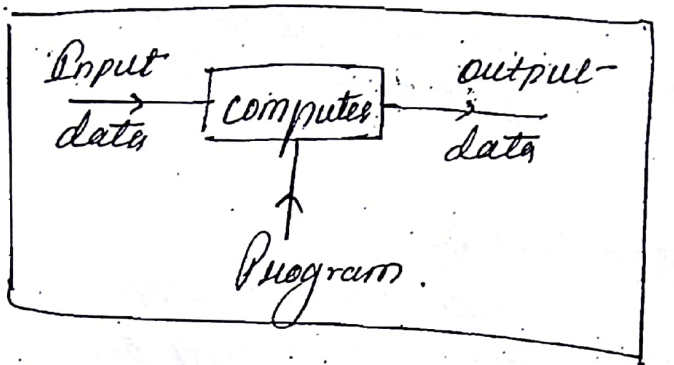
Computer organization - william stallings
Computer Architecture & organisation - Morris Mano

Duration: 65 hrs

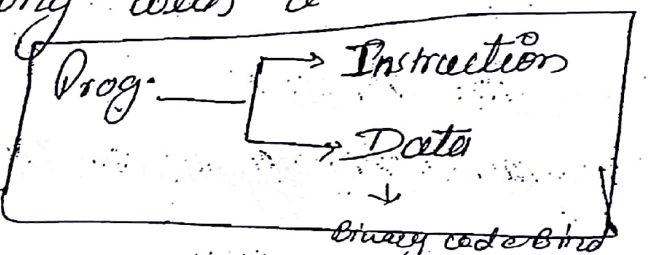
Faculty: Sagar Pingile
E-mail: Sagar26-2003@yahoo.co.in

Module 1 - Computer Functionality

* Computer → It is a computational m/c used to process the data under the control of a Program. ∴ computer s/s functionality is program execution.



• Program → It is a sequence of instruction along with of the data.



• Instruction → It is a binary code w/c is designed & inside the processor to perform some task.

Binary code - Bind - operation with

Binary code bind with op

Ex X-CPU supports 8 operation

$$\text{Opcode size} = \log_2 8$$
$$= 3 \text{ bit.}$$

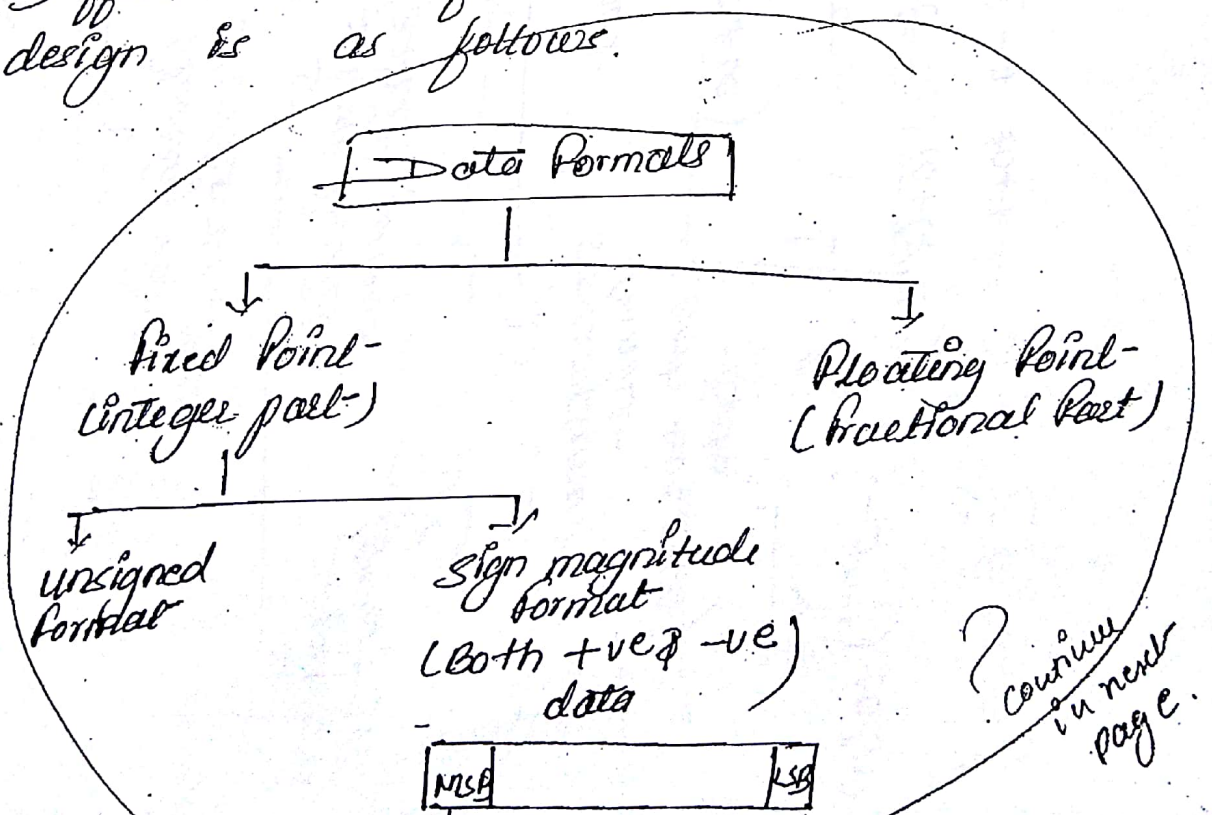
OP code		Op^n
000	——	+
001	——	*
010	——	/
⋮		⋮
111	——	AND

decided by the designer of a x-CPU
↓
Stored in the ROM.

- Data → It is a Binary code w/c is associated with a value based on the data formats.

Binary code	–	Bind with	–	value

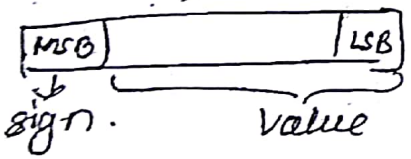
Different data formats used in the computer design is as follows.



Continue in next page.

***** Data formats

- Fixed point (integer part)
  - magnitude format
    - unsigned format (only +ve data)
      - e: {0 to b^2-1)}
      - : {0 to 15}
    - sign-magnitude format (Both +ve & -ve data)



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

$\{-7$ to $+7\}$

In sign MF → MSB is used to indicate sign. if MSB is +ve then, MSB is zero & value is +ve. If MSB is -ve, then MSB is 1 & value is -ve.

  - complement format
    - 1's complement format (Both +ve & -ve) data

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

$\{-7$ to $+7\}$

    - 8's complement format (Both +ve & -ve) data

$\{-(2^{n-1}$ to (

$\{-8$ to $+7$

- Floating point (fractional part)
  - Single precision (32-bit) format.
  - Double (64-bit) for

it

ent data values associated with a
t binary code is as follows.

unsigned data	sign-magnitude data	1's compl data	2's compl data
0	+0	+0	+0
1	+1	+1	+1
2	+2	+2	+2
3	+3	+3	+3
4	+4	+4	+4
5	+5	+5	+5
6	+6	+6	+6
7	+7	+7	+7
8	-0	-7	-8
9	-1	-6	-7
10	-2	-5	-6
11	-3	-4	-5
12	-4	-3	-4
13	-5	-2	-3
14	-6	-1	-2
15	-7	-0	-1

Not in use data redundancy

Not in use.

→ MSB bit ③ +ve data never store in the Complement system ^MSB bit -ve

there is need to take complement.

→ Signed data always rep^n with 2's complement

→ for Signed magnitude data we always take 2's complement bcoz in 2's complement redundancy is not there.

→ in signed magnitude data & 1's complement data redundancy is there +0 & -0 not possible.

1's complement

1000 → 0111 (7)
1001 → 0110 (6)

2's complement

1000 → 0111
+ 1
1000 = 8

1001 → 0110
+ 1
0111 = 7

unsigned data

expressible data

overflow

design to indicate the range exceeding condition of a unsigned arithmetic.

condition : Is there an extra bit out of MSB → T = Set = 1 = C (carry) (true); F = reset = 0 = NC (false) / No carry.

extra bit generated out of MSB data is exceed.
extra bit not generated out of MSB data is not exceeded
set → extra bit required
reset → extra bit not required.

In w/c case given data's exceeding range

Ex ① 5 − 0101 (no carry)
6 − 0110
11 / CY=0
1011 / CY=0 → carry flag.

Justify

PSW (It is combination of two registers)
- Accumulator → 1011
- Flag registers → CY → 0

Ans 1011 (11) 4 eleven.

Signed data represents in computer in w2C format
2's complement

Ex ② 8 − 1000 (carry 1)
9 − 1001
17 / CY=1
0001 / CY=1

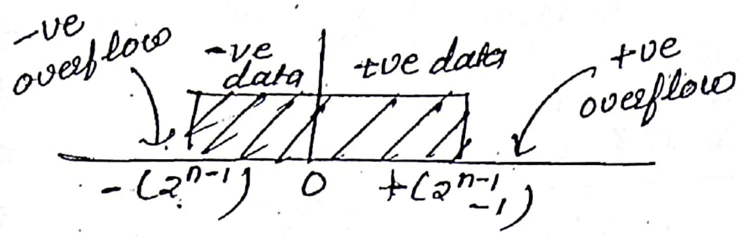
Justify: PSW
- ACC → 1011
- Flag register → CY

CPU → it is fully tested w/w.
5th bit taken from carry bit.

Ex ① 1111 (carry 1111)
1110
5 with no carry 0101

Appl
Adv DS, compiler
CN
TOC, DS, EMath
COS
OS
DED
DBMS
English

4) Signed Data



same sign no are added range is exceeded.
+6+7 = +13
is +13 is in range, no.
so +ve overflow
extra bit require in MSB position
+ve OF → extra bit should be zero.
-ve OF → extra bit should be 1.

Note: Overflow flag is used in the CPU design to indicate the range exceeding condition of a sign arithmetic

My Ex ① (no carry out of MSB is ⊗; carry in MSB Ex.①)

x → 0111 : +7
y → 0110 : +6
z → 1101 +13

Ex ② (carry out of MSB ①; no carry in MSB ⊗)

-8 : 1000
-7 : 1001
-15 0001

Ex ③

1000 -8
0111 +7
1111 -1

Ex 4 (carries ① ① 1)

0111 : +7
1110 : -2
0101 +5

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

condition → There is a carry into the MSB & No carry out of the MSB (or) vice versa → T(true) = set = 1 = OV (overflow)
→ F(false) = reset = 0 = NV (No overflow)

In carry	out carry [w.r.t MSB]	
0	0	NV(0)
0	1	OV(1)

MSB of a operand1 | MSB of a operand2 | MSB of a operand3

Ex 1.

	①11
+7	0111
+3	0011
+10	1010
OV = 1	OV = 1

Justify: PSW

- Acc → 1010
- Flag reg. → OV → 1 → +ve overflow

Ans: (OV) 0 1010 (+10)

Ex 2

	①①11
+7	0111
−3	1101
+4	0100
OV = 0	OV = 0

Justify: PSW

- Acc → 0100
- flag reg → OV → 0

Ans 0100 [+4]

Ex 3.

	11
−7	1001
+3	0011
−4	1100
OV = 0	OV = 0

Justify PSW

- Acc → 1100
- flag reg. → OV → 0

Ans 1100 [−4]

Ex 4.

	①
−7	1001
−3	1101
−10	0110
OV = 1	OV = 1

Justify PSW

- Acc → 0110
- Flag reg. → OV → 1 → −ve overflow

Ans (OV) 1 0110 [−10]

Q1) consider the foll. signed data, perform the addi^n ⑤
o/p^n what is the status of a OV flag in the flag register.

① 10010011 −19
11011011
01101110

$x y \bar{z}$ = OV = 1.

when Binary data given check directly.
MSB

Q2) consider the foll. 8 bit data computa^n (−62) + (−24) wt is the status of OV flag in the flag register after processing the data.

8 bit Range.

$\{-(2^{8-1}) \text{ to } +(2^{8-1}-1)\}$
$\{-128 \text{ to } +127\}$

when raw data given check range.

−62
−24
−86

OV = 0

3220 → sup^n bar.

−62 62 → 00111110

11000001
+1
11000010
11101000

$xy\bar{z}$

−24 24 → 00011000

Computer architecture → Blueprint

① Based on the Program storage, computer architecture is two types

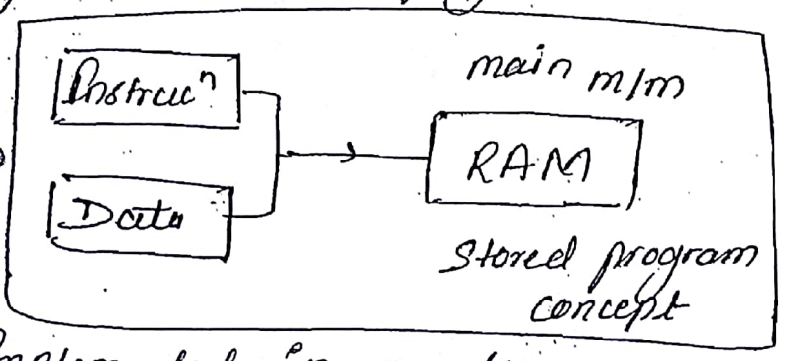
if we don't have von neumann archi. we need one room for comp. SIs one comp. for music play, 1 comp for video play

① Single m/m Arch: [Ex. PC. one SIs can do multiple work at a time]
[von-Neumann Arch.]

② Multi m/m Arch.
[Horvard Arch]

① von-Neumann Arch → In this Arch. user program is always required in the main m/m.

- Main m/m is a volatile m/m, so we can load different Application program into a main m/m.
- CPU always execute the "main m/m part of a prog. by generate the physical address.



ismai I & D ek hi m/m mai hota hai →

app^n prog uses 1 m/m. RAM

Implemented in a Microprocessor Design.

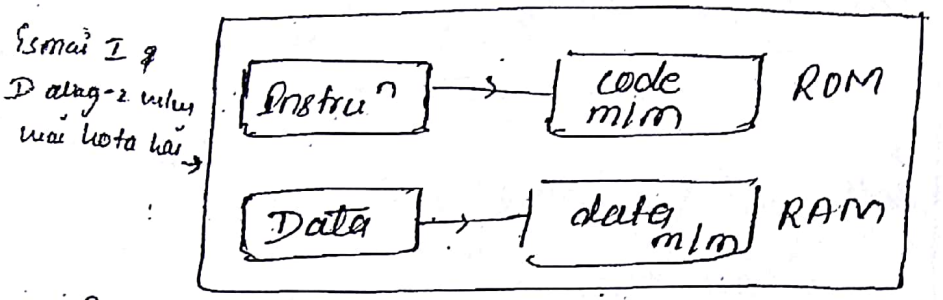
Ex 8085 (8bit) MP → 64KB RAM
8086 (16bit) MP → 1MB RAM

This arch. used in personal computer.

② Harvard Arch

✓ In this arch. program logic is stored in the permanent m/m (ie ROM) & data is stored in the volatile m/m (ie RAM)

✓ In this arch. CPU always execute the predefined prog. in the ROM chip on a different data elements.



Esmai I & D alag-2 milen wai hota hai →

App^n prog uses two milen ROM & RAM

Implemented in a μcontroller Design.

Ex. 8051 μ controller → 4KB ROM
→ 128B RAM

used in the intellegent s/s Design.

Ex. Embedded S/S.

Note → CO is a implemention of the von-neumann arch.. so computer contain 3 fundamental components.

① CPU (processing)
② Memory (storage)
③ IO (External communication)

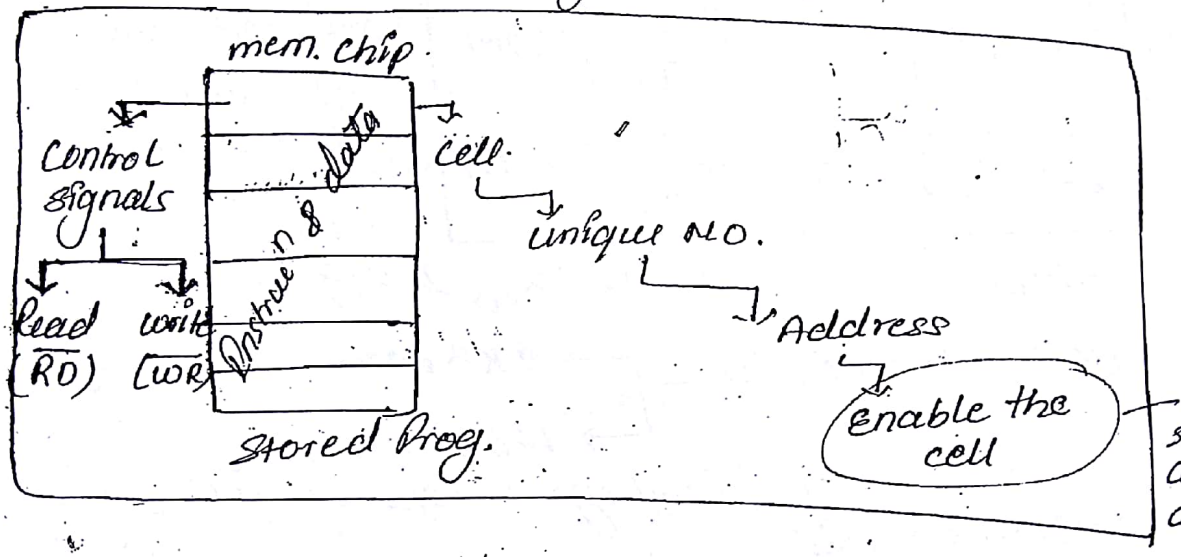
++ Computer Organisation

According to a ~~of~~ von-neumann arch. wer

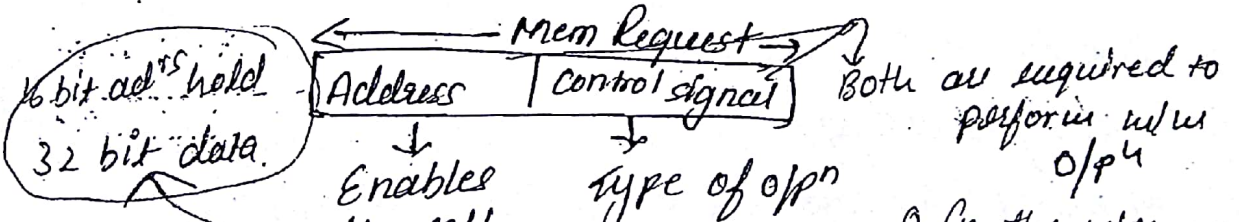
* Memory organisation

In CO → main means → main memory.

- m/m is a storage element in the computer.
- m/m chip is divided into a equal parts called as "cells".
- m/m cell is identified with a unique no. called as add^rs
- m/m chip recognises the control signal to indicate the the type of o/p^n.



- CPU generates the m/m request to access the user prog. from the m/m chip i.e



Q In the m/m req. How many control signal.
A → 2 (Read & write)

- M/m add^rs depends on the capacity of m/m chip.
- m/m chip configuration is super Q How many bits of

it is consider 64K×8 m/m chip to analyse the (7)
cof configura^n.

ad^rs space — (64K)×(8) — data space → m/m chip looks like this.

these are two parameters ① ②

1st parameter define ↓ # cells in the chip i.e 64K cells.
↓
addrs bits required to access the chip.
i.e $\log_2 64K = 16$ bits.
↓
Ad^rs space/ad^rs range (m/m ad^rs repre^n by hexadecimal format)
m/m map of a m/m chip

→ How many bits of data stored in the cell?
↓
cell size {# bits stored in the cell}
↓
data space in the cell.

→ No. of

range → min & max.

[0000 0000 0000 0000] → min
[1111 1111 1111 1111] → max

[0000 FFFF]16/H (0x → Base
↓ Hexa decimal.
(→ in high level lang. 0x is used
→ Hexadecimal denota^n in the C-lang.)

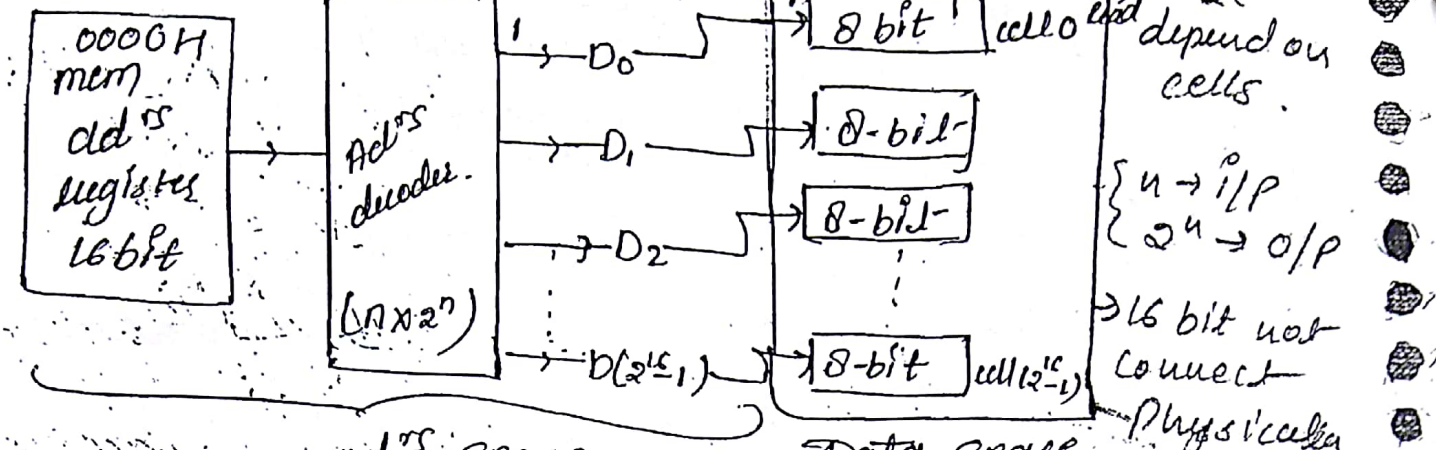
Ex
256K×4
128M×6
32G×8
256×8
32M×4

{ 1K = 2^{10}
1M → 2^{20}
1G → 2^{30}
1T → 2^{40}
1P → 2^{50} }

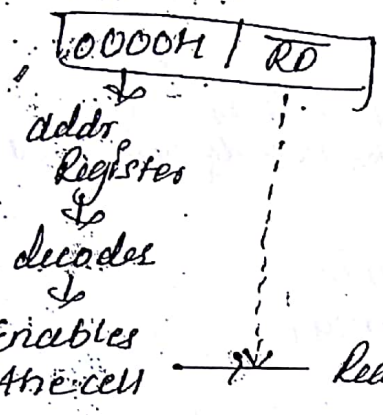
27 bit ad^rs cell hold 6 bit data — 128M×6

$2^x \times y$ → x bit ad^rs cell hold y bit data
m×n → $\log_2 m$ bit ad^rs cell hold n bit data.

data → Physical
ad^rs → logical



Ex. CPU generated mem request.



in the mem we are storing the data.

- In the computer sys design data is always refered in a form of binary. Different refferential elements of a binary data is as follows.

Processor	Bit	Nibble	Byte(B)	word (no. of bits of data Processed by CPU at a time)
8 bit up	1 bit	4 bit	8 bit	8 bit
16 bit up	1 bit	4 bit	8 bit	16 bit
32 bit up	1 bit	4 bit	8 bit	32 bit
n bit up	1 bit	4 bit	8 bit	n bit

Binary

word size = word length = # bits processed in the CPU at a time.

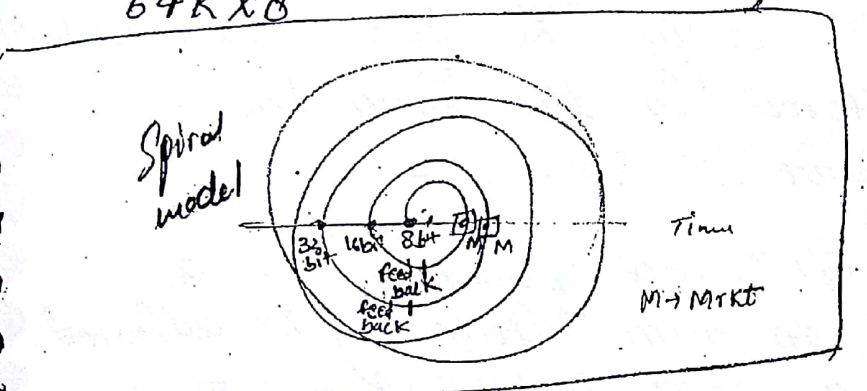
- Based on the data storage configuration it four types.

① Bit addressable m/m → Bit, Nibble, Byte all the entitial unit.
② Nibble " "
③ Byte " "
④ word " "

→ Highest common denomination is Byte.
→ We can communicate one Byte & word.
→ o/p wise denomination ↳ word
↳ variable.
→ By grouping cells in m/m we can make word.
→ w varies → it is individual/indivisible parameter.
→ How data store in m/m
→ How data is accessed from m/m

Ex 64KB → 64KX8 → 64KX8

Ex 64KW → 64KX16 → 64KX? (w depend on user not fixed difficult to find)



- Byte addressable vers vs word addressable m/m

when the m/m cell contain 8 bit data space then the corresponding ad's space named as byte address

Ex 64X8 : 6 bit address
256X8 : 8 bit "
1KX8 : 10 bit "
2^nX8 : n bit "

6 bit ad's hold 8 bit data
8 bit ad's hold 8 bit data
10 bit ad's hold 8 bit data
n bit ad's hold 8 bit data

→ cell size must be 8 in 1 Byte
→ ad's size may be any.

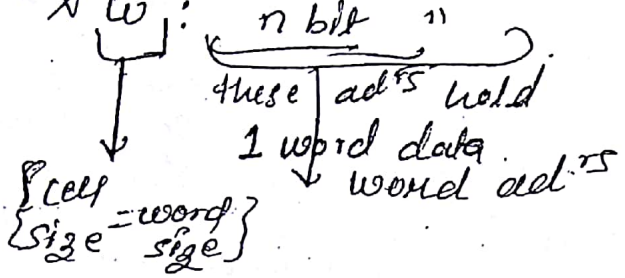
Byte ad's, it doesn't matter.

{ cell size = 8 bit } ∴ matter.

When the m/m cell contain 1 word data Informa then the corresponding adrs space is called as word adrs.

Ex
$64 \times w$: 6 bit adrs
$256 \times w$: 8 bit "
$1K \times w$: 10 bit "
$2^n \times w$: n bit "

Particular Byte at the mem all there.



Note → ① default m/m configuration in the computer design is byte addressable so data is stored in the m/m in a byte wise sequence

② In the CPU o/p" are performed on a word formats, so m/m interfacing is adjusted according to a word length of a CPU ∴ CPU will be accessing the data from the m/m in a form of words, by refering the multiple of cells in parallel.

③ data storage sequence in the m/m chip is byte wise.

Data accessing sequence from the m/m chip is word wise

21/7/2017 ⑨
Friday.
Class-2

Class-2

Ex 8086 (16-bit) µP

No. of op^n → 2^6
64 op^n.

Ins^n: | MOV | AX | [2000] |

- OP code → Type of op^n → Data transfer op^n
- Dest^n → Register [Reg. is referred with Name] → AX → 16 bit
- Source → m/m [m/m is referred with ad^{rs}] → By default value is Byte ad^{rs} of m/m. → [2000] → Byte ad^{rs} → compiler give ad^{rs} alloc^n to data

Execu^n

AX ← M[2000] : Even Bank cell
AX ← M[2001] Odd Bank cell.
→ Everytime two cells coming → colum

AX : 15 14 13 ...... 8 7 6 ... 0
15–8: HB (higher byte)
7–0: LB (low Byte) (Lower Byte)

while execution of a above instruction two possible outcomes are generated due to lack of a order of the data storage in the m/m i.e

Mem. contents

Address	Content
0000	
⋮	
2000	28H
2001	12H

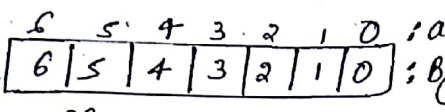
① If [2000] contain LB then, AX = 1228H
② If [2000] contain HB

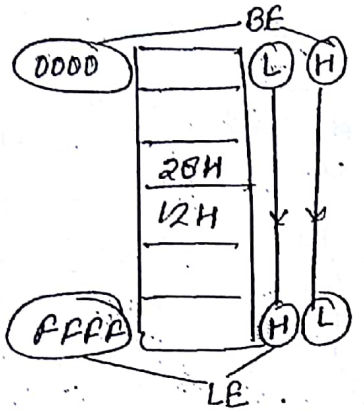
due to order ~~problem~~ missing associated outcome are only 2.

- To handle the above problem m/m ad^{rs} interpretation (Endian-ness) techniques are used.
- There are two types.

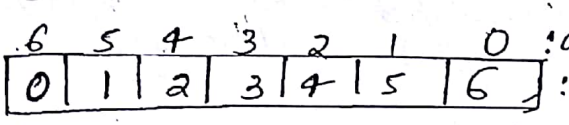
① little endian [Asending order]
② Big endian [descending order]

① In the little endian "lower ad^{rs} contain lower Byte & Higher ad^{rs} contain higher byte".



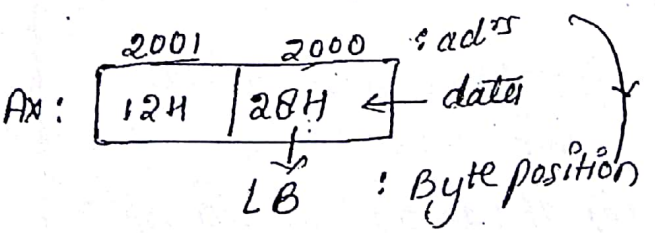


② In the big endian technique "lower ad^{rs} contain lower Byte & Higher ad^{rs} contain lower byte".

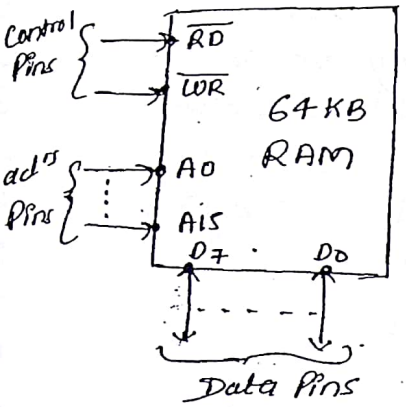


- Default ad^{rs} interpretation is little endian.

∴ above instruc^{n} generates the foll. o/p.



AX = 1228H

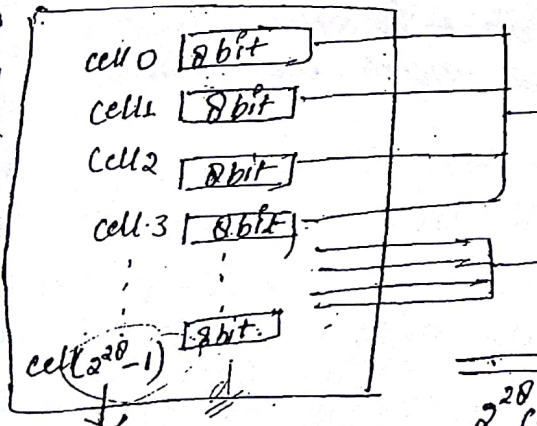


Q1) consider a hypothetical Processor with a word length of 32bit w/c supports 256 MB m/m. Processor is enhanced with a word addressable m/m space. How many adrs bits are saved in the enhanced CPU.

X
- s/s old → 256 MB m/m
  enhance → 32bit → new s/s adrs
- old s/s adrs
- new s/s adrs

s/s old

256 MB m/m



$\log_2 2^{28}$

↓

28 bit

Ans

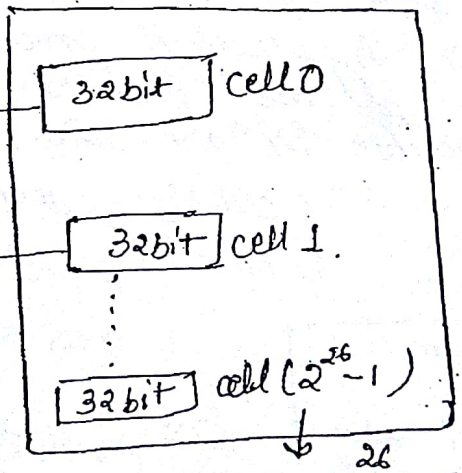
2^{28} cells

→ 4 cells

$= \frac{2^{28}}{2^2}$

$= 2^{26}$ cells

s/s new



$\log_2 2^{26}$ → 26 bit adrs

adrs

Bit saved = 28 - 26

= 2 bit

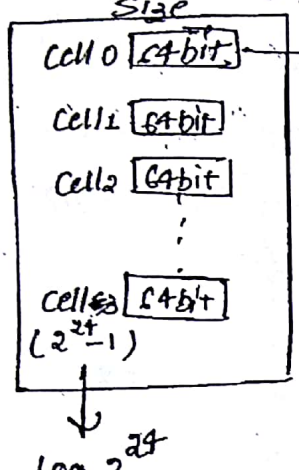
supports 16M word m/m. Process is enhanced with a Byte addressable m/m space. How many no. of adrs bits are required in the new CPU to suffer the new m/m modelle.

Ans

S/S Old

16M W m/m

Word = 64 bit



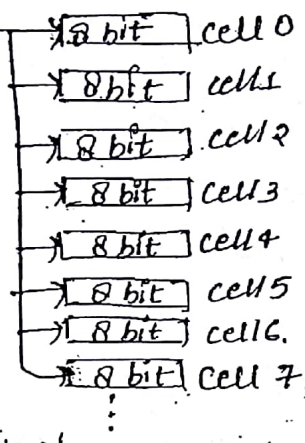
$\log_2 2^{24}$

↓

24 bit adrs

S/S new

- Byte addressable m/m
- cell size = 8 bit



No. of cell in the new module = 2^{24} cell * 8 cells

$= 2^{24} \times 2^3$ cells

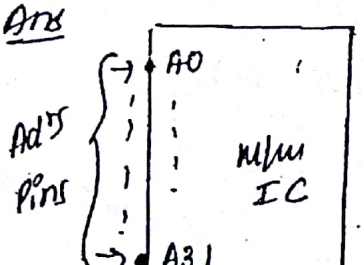
$= 2^{27}$ cells

$= 27$ bit adrs Ans

Q3 A hypothetical word addressable m/m IC contain 32 adrs pins & 24 data pins. m/m chip is interfaced to a hypothetical processor. What is the word length of the processor & How much main m/m is pacsible in the CPU.

$$\begin{cases} \text{cell size} = \text{word size.} \\ \text{m/m cell} = 24 \text{ data} \end{cases}$$

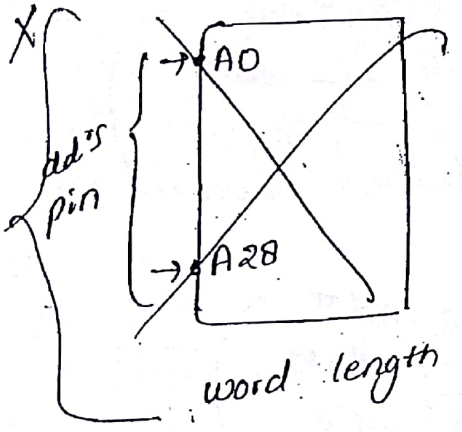
Ans



- word addressable m/m chip
- cell size = word size
  ↓
  24 bit = word size
  ∴ word size = 24 bits
- m/m size: Depends on add^{rs} pin
  ⇒ 2^{32} cells
  ⇒ $2^2 \times 2^{30}$ cell
  ⇒ 4 G cell → 4GW Ans

Q4) A hypothetical m/m IC contain 29 add^{rs} pins. It is interface to a μP by organise the m/m chip into a 4 logical Bank in 4th. what is the word length of a processor & how much main m/m possible in the s/s.

A



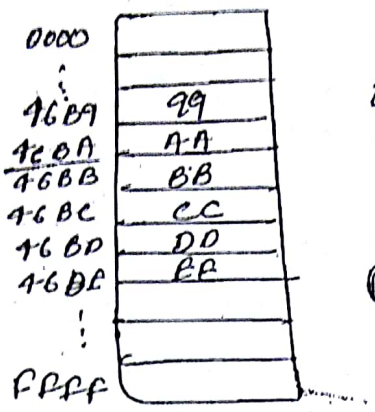
word length of process is 32 bits.

- Dfault m/m configura^{n} is Byte add^{rs}ble
  - cell size = Byte = 8 bit
  - 4 cell are interfaced to μP.

∴ word length = 4 * 8 bits
= 32 bits

- m/m size = 2^{29} cells
  ↓
  = $2^9 \times 2^{20}$ cells
  ↓
  512M cells

Q5) consider a 32 bit- hypothetical computer that is accessing the data word from the m/m chip with a starting address of $(46BA)_H$ onwards. m/m contents is as follows

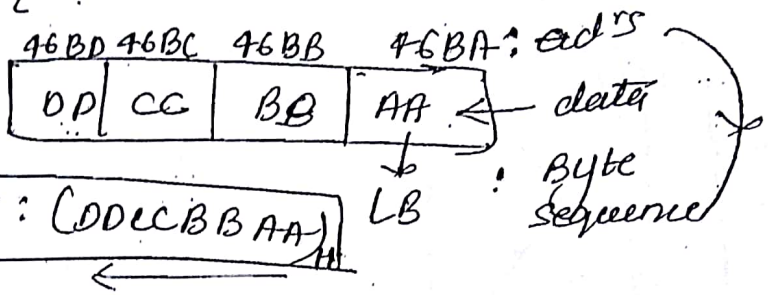


wt is the data word fetched from the m/m when m/m is designed with

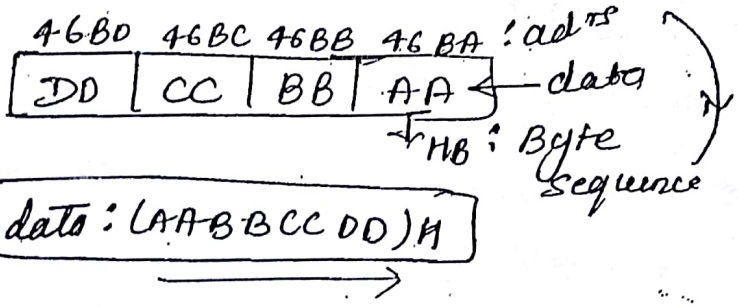
(a) little endian
(b) Big-endian.

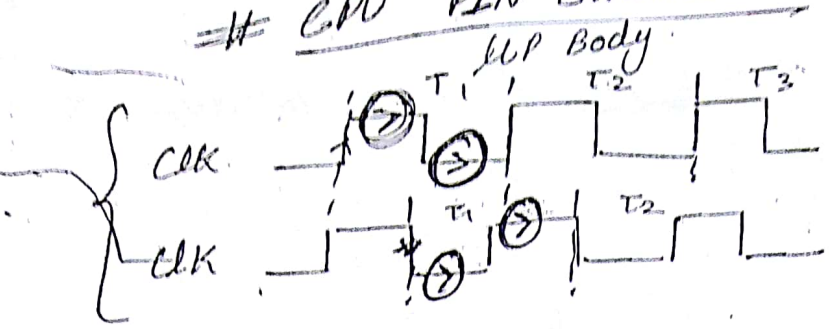
Ans

(a) {32 bit- ⇒ 4B}



(b)





Pin IC of 8085
Pin IC of 8086

Pin who bar Active

- Processor contain set of hiw pins used to high. perform the various pins externally connected components m/m & IO.
- Hiw Pins are categorise into 3 tools.
  - (a) Active low
  - (b) Active high
  - (c) time multiplexed Pin

(a) Active low

This pin is enabled when the clock pulse is in the low state.

It is denoted as. $\overline{\text{Pinname}}$ (Pin name with bar)

Ex $\overline{RD}$, $\overline{WR}$, $\overline{INTA}$ etc

(b) Active high

This pin is enabled when the clock pulse is in high state.

It is denoted as Pinname (no bar).

Ex. ALE, INTR, HOLD, HLDA etc.

(c) Time multiplexed Pin.

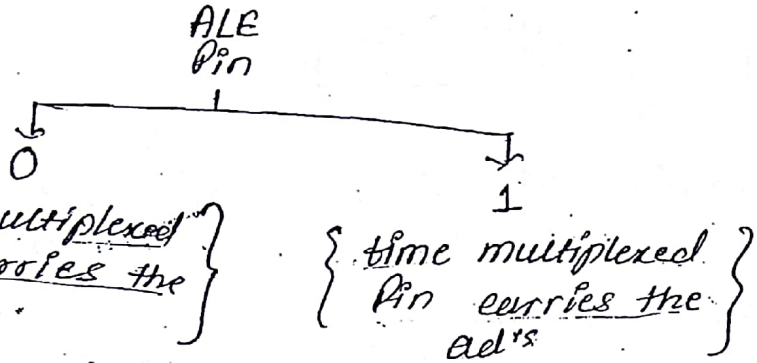
This pin is used to carry the add^{rs} and data, but not at the same time.

Ex. In 8085
AD7 - AD0
↓ adrs ↘ data

Ex In 8086
AD15 - AD0

Advantage → Reduces the H/W Pins.

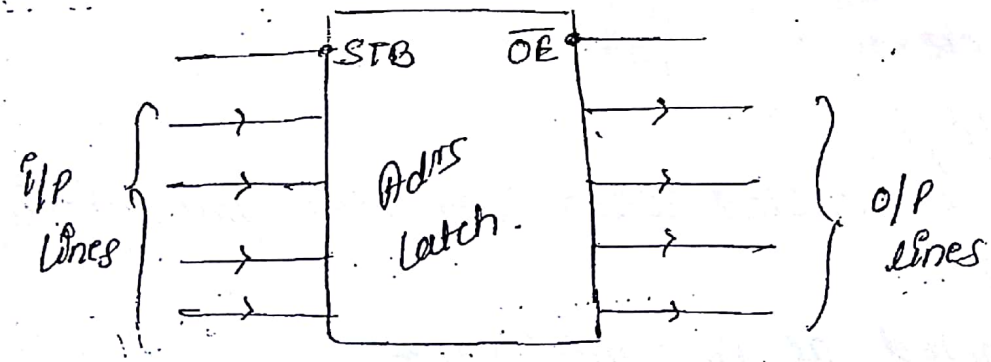
- In this implementation ALE pin is used to differentiate the add^rs & data i.e.,



- In this implementation add^rs latch is used to lock the add^rs & contain two control pins used to control the i/p & output line of a latch.

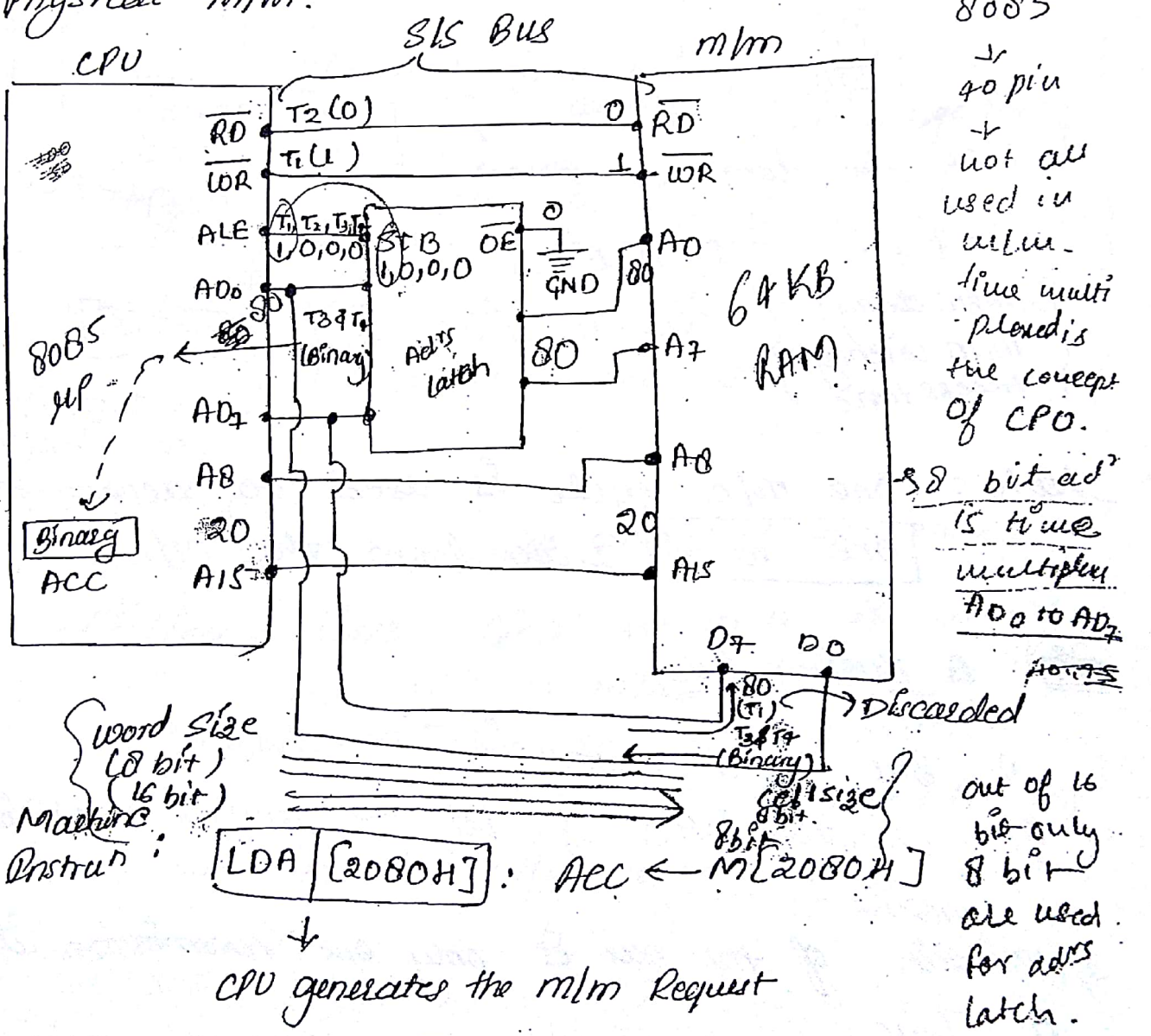
① STB (Strobe) Pin → 0 : i/p lines are disabled
→ 1 : i/p lines are enabled.

② $\overline{OE}$ (o/p enable) Pin → 0 : o/p lines are Enabled
→ 1 : " " " disabled.



Memory Interfacing

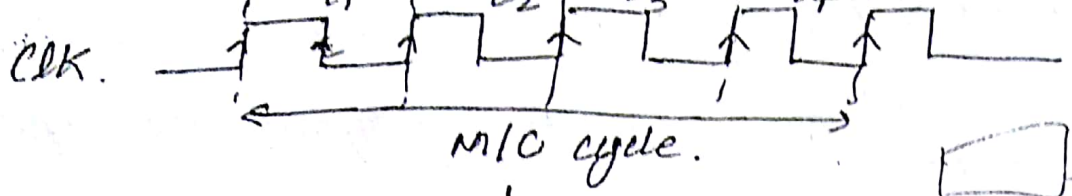
- This concept is used to integrate the CPU and m/m.
- In this process pins are mapped b/w the CPU & m/m respectively.
- let us consider 8085 processor as a reference CPU.
- It is a 8 bit processor supports 64 KB Physical m/m.



8085
↓
40 pin
↓
not all used in multiplexed is the concept of CPO.

8 bit add is time multiplexed AD0 to AD7

out of 16 bit only 8 bit are used for add latch.



Actions:

(1) Send the address.

T_1 : ALE = 1

80 ⇒ AD_7 - AD_0

20 ⇒ A_{15} - A_8

(2) Send the control signal (CS)

T_2 : ALE = 0

$\overline{RD}$ = 0

$\overline{WR}$ = 1

(3) Read the data

T_3 & T_4 : ALE = 0

{ m/m delay / m/m latency / Access time } $D_7 - D_0 \Rightarrow AD_7 - AD_0$

Note: one m/c cycle is used to access one word data from the m/m.

Study

m/m interfacing:

in 8085 & 8086

STA | 2050

ACC

2050 | WR

$AD_7 - AD_0 \Rightarrow D_7 - D_0$

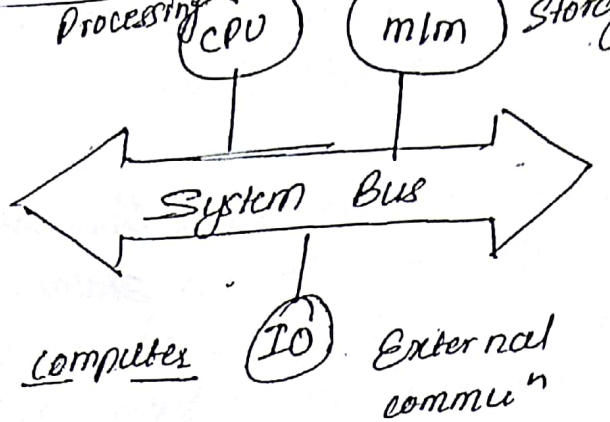
4T0

1T0

4T m/c

✱) System Bus

1) Bus is a communication channel.
2) characteristic of bus is shared transmission media.
3) Limitation of the bus is only one transmission at a time.
4) A bus m/c is used to provide the communication



Classroom as a S/S Bus

	Functionality	Speed		
CPU	Processing	Fast →	Master CU, ALU, all the internal comm.	
m/m	Storage	Slow →	I/O slave Internal Storage RAM capacitor ↳ periodic charge require	• Adrs • CS's
IO	External comm^n	Very-2 Slow	~~Sometime~~ slave. • Usually	• IO adrs = Port adrs • Control signals. CCS's

- S/S bus contain 3 category of the line used to perform the comm^n named as
  1. adrs lines
  2. Data lines
  3. control lines. → to decide capacity of internal m/m.
  4. Adrs lines → these lines are used to carry the adrs to m/m & IO.
     - so adrs lines are unidirectional.
- Based on the ~~the~~ no. of adrs pin (lines) in the CPU we can determine the capacity of a m/m S/S..

Ex In 8085

$A_{15} - A_8$ $AD_7 - AD_0$

↓

2^{16} cells

$A_{19} - A_{16}\ AD_{15} - AD_0$

↓

20 AL's ↳ add^{s} lines

↓

2^{20} cells

↓

1M cells

↓

1MB.

line carry 1 bit status.

③ Data lines (DL) → These lines are used to carry the binary sequence blw the CPU, mlm & IO.

- 8O data lines are bidirectional.
- Based on the data lines we can determine the word length of a processor.
- Based on the word length we can measure the performance of a CPU.

Ex① In 8085,

$AD_7 - AD_0$

↓

8 DL's are line multiplexed with AL's

↓

word length = 8bit

↓

Operations are performed on a 8bit data formats

Ex② In 8086

$AD_{15} - AD_0$

↓

16 DL's

↓

word length = 16bit

↓

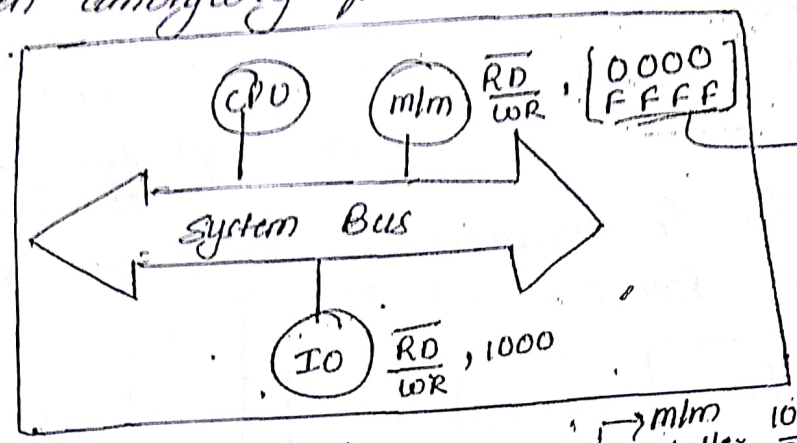
Operations are performed on a 16 bit data formats.

③ control lines → • These lines are used to carry the control signal & timing signal.

- I/O o/p^n & m/m o/p^n with a CPU clock.
- These are unidirectional.

Note: whenever clock signal for CPU same for I/O & m/m

Note i) when these is a common bus, common control s/s & signals & common adr^s space is used in the m/m & I/O, then ambiguity problem is occurred i.e describe below.



CPU generates the m/m request:
- 1000 → AL's
- RD → CL's

1000	RD

→ m/m controller: 1000-valid, RD-valid } m/m o/p^n → conflic / collision
→ IO controller: 1000-valid, RD-valid } IO (ambi-guity) op^n

Ex. In a class, same roll no. allot to two students two students generate ans. teacher confused.

↓ Ex. based on above concept. Ambiguity is there.

- To handle the above problem bus configuration are used in the computer design & are three types

① IOP (input-output processor)
② Isolated IO (IO-mapped-IO)
③ mem. mapped-IO.

① IOP (input-output processor) → this config^n uses the common control signals common adr^s space but diff. buses for both m/m & IO.

Additional hlw is required the mlm bus & IO bus so it is expensive. ∴ It is suitable in the high performanced CPU design.

(2) Isolated IO → This configuration uses the common bus & common adrs space but different control signals for both mlm & IO.

control signal
- mlm Based
  - $\overline{MEMRD}$: Load
  - $\overline{MEMWR}$: Store
- IO Based
  - $\overline{IORD}$: In
  - $\overline{IOWR}$: Out

- $IO/\overline{M}$ → Pin is required in the CPU to implement the Isolated IO. i.e.,

$IO/\overline{M}$	$\overline{RD}$	$\overline{WR}$	CS
0	0	1	$\overline{MEMRD}$
0	1	0	$\overline{MEMWR}$
1	0	1	$\overline{IORD}$
1	1	0	$\overline{IOWR}$

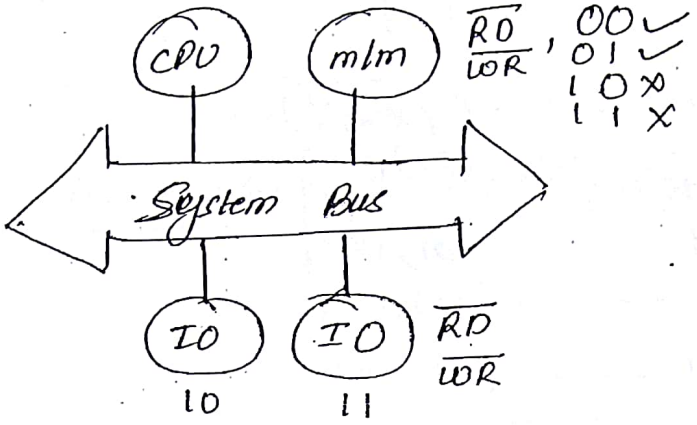
Active low enable → disable.

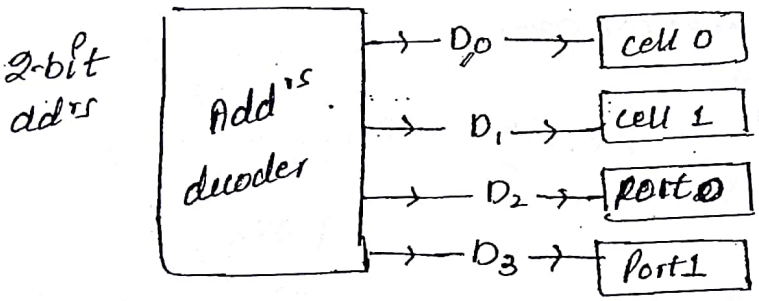
- It is less expensive, suitable in the General Purpose computer design.

(3) Mlm mapped IO → This configuration uses the common bus & common control signal but unique adrs space.

- In this design some of the mlm adrs are assigned to a IO ports. the corresponding adrs of the mlm cell are disable.
- In this design port adrs becomes the part

- So limitation is complete m/m space is not on the use.





Q1) Consider a hypothetical processor w/c supports 16MB m/m. Processor uses m/m mapped IO in w/c, when the 3 MSB bits of the adrs is one then use them for IO ports. How many no. of port adrs & m/m adrs are possible in the processor.

Ans

16MB m/m
↓
16M x 8
↓
$\log_2$ 16M cells
↓
$\log_2 2^{24}$
↓
24 bit adrs

Adrs

A_{23} A_{22} A_{21} | A_{20} A_{19} ..... A_0

3bit | 21bits

↓

0	0	0	mem
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	IO
1	1	1	

Port addrs = 1×2^{21}

[111 0000 - - - - (21) 0's
to
111 1111 - - - (21) 1's]

No. of (I/O) m/m addrs = $(7 * 2^{21})$

[000 0000 - - - - - - (21) 0's
to
000 1111 - - - - - (21) 1's]

[001 0000 - - - - - (21) 0's
to
001 1111 - - - - (21) 1's]

⋮

Total Space = I/O space + m/m space

$= (1 * 2^{21}) + (7 * 2^{21})$

$= (1+7)\, 2^{21}$

$= 2^{24}$

Q2) A hypothetical prop processor contain 28 time multiplexed bits to carry the addrs & data & also 6 addrs pins. calculate the foll.

(a) m/m capacity (b) processor word length

(c) Processor uses m/m mapped I/O in wch port addrs all the Highest Byte addrs pins are connected to logic zero. How many no. of port addrs are possible in the CPU.

Solⁿ (a) m/m size : depends on addrs pins.

$A_{33} - A_{28}$ $AD_{27} - A_{00}$

↓

34 ALs

↓

2^{34} cells

↓

$2^4 \times 2^{30}$ cells

(a) 16G cells → 16GB.

(b) word length = # DL's in the CPU → $AD_{27} - AD_0$ → 28 bit.

(c) Add rs:

$A_{33}\ A_{32}\ A_{31}\ A_{30}\ A_{29}\ A_{28}\ AD_{27}\ AD_{26}$ | $AD_{25} - - - - AD_0$

8bit | 26 bits.

↓

2^8 codes

- only one code contain all 0's → used it for IO → $(1 * 2^{26})$
- 255 codes are different → use them for mlm → $(255 * 2^{26})$

22/July/2017
Saturday
Class-3

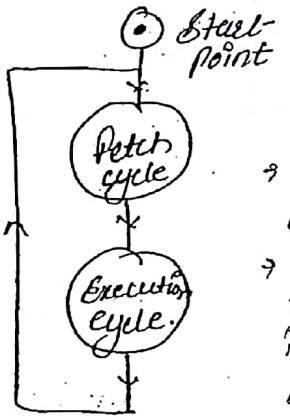
Instruction cycle

PC describes the execution sequence of a program.

It contain two subcycle.

1. Fetch cycle
2. Execution cycle.

State diagram is



→ Instruction who will give instructions at user.

→ Instruction present in main memory

→ Fetch → reading ↳ read the data

→ Initially Instruction is required

→ Data fetching required in Execution cycle.

→ it is a ∞ loop dia. no end point.

→ where is ending

in comp end point logically by halting the prog.

PC → Prog. control register / Program counter

1. Fetch cycle objective of a fetch cycle is Instruction fetch. In this process CPU reads the instruction from the m/m based on the PC (Program counter)
   → when CPU touch with main memory
   → storage ↔ sequence → Bytewise access from m/m → word.
   instruction 1 word → 1 time m/m refer required.
2. PC is a mandatory register in the CPU used to control the program execution flow.
3. PC holds the starting add's of a Instruction immediately point the next instruction add's in a sequence. Here, starting add's is supplied by the programmes & the next instruction add's is automatically generated by the CPU by incrementing the PC with a step size.
4. Step size is depends on the word length of a processor. It is a fixed constant.

during the execution of a current Instru^{n}..

used to trace step by step exe^{n}

g → go → it execute all give o/p of last one

Best is t.

If Inst^{n} fetch

-t {trace} = starting add^{rs} of a Program : executable Stmt

↓

PC → CPU generates the m/m Request s/s Bus { AL's CL's } → m/m Binary → s/s Bus {DL's} → CPU. ↓ decode

PC ← PC + step size.

AL's : add^{rs} lines
DL's : data lines
CL's : control lines

- Analysis associated with above concept
- Fixed length Inst^{n} = Inst^{n} size = word size. RISC
- variable length Instru^{n} CISC
  - (a) opcode size = word size
  - (b) opcode size < word size
  - (c) opcode size > word size never possible.

out of 3 Byte one for fetch 2 for decoding

1) When the processor supports fixed length Instru^{n} where **Instruc^{n} size = word size**. then during the fetch cycle the complete instruc^{n} is transferred to CPU so PC will be incremented to a next Instruction adrs.

2) When the processor supports variable length Instru^{n}
   (a) where **opcode size = word size**. then during the fetch cycle only the opcode is accessed from the m/m.

adrs during the decoding stage.

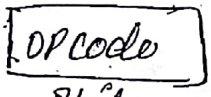
(b) where opcode size < word size, then during the fetch cycle, 1 word ~~Instruct~~ Information is transferred to CPU Internal storage.
later CPU will be accessing the opcode part from the Internal storage.

(c) Opcode ~~size > word size~~ not exist.

8085 μP (8-bit) CPU
{word size = 8bit
opcode size = 8bit}

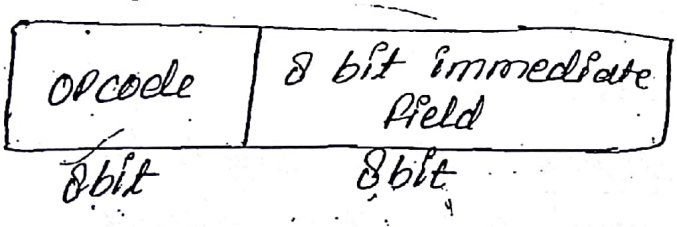
Instruction Design

(1) One Byte (word) Instruⁿ



Ex RIM
SIM

(2) two Byte (word) Instruⁿ



Ex: MVI 23H

(3) three Byte (word) Insⁿ

Ex LDA [2080H]

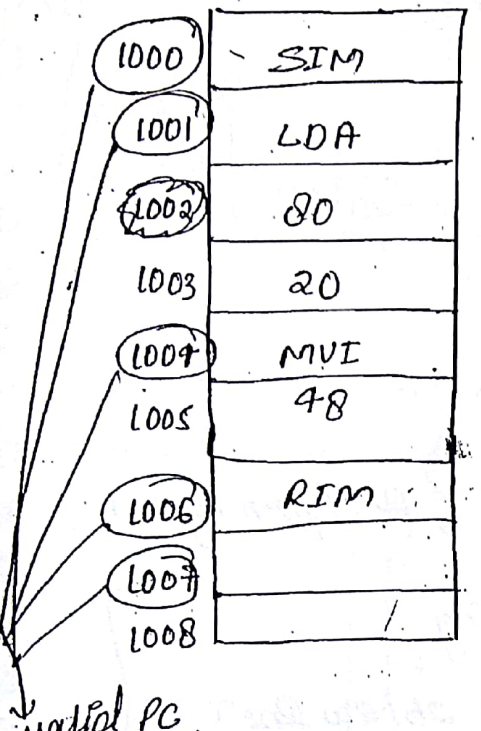
Operational state

User mode

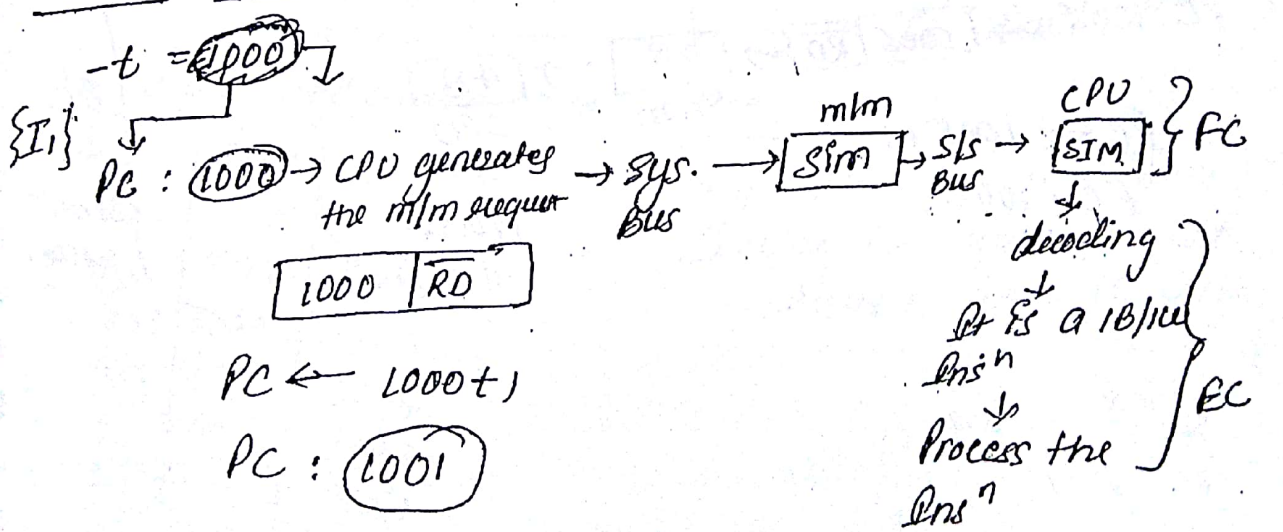
Org 1000 → origin

I_1 : SIM
I_2 : LDA [2080]
I_3 : MVI 48H
I_4 : RIM
⋮ ⋮

Mem Storage



Execn sequence.



{I2}

PC: 1001 → CPU generates the m/m req. → 1001 | RD → S/S Bus → m/m: LDA → S/S Bus → CPU: LDA } FC

↓ decoding

↓ It is a 3B/3W Instⁿ

↓ 2 m/m sequence required.

PC ← 1001+1

PC ← 1002

PC: 1002 → 1002 | RD → S/S Bus → mem: 80 → S/S Bus → 80 CPU

PC ← 1002+1

PC ← 1003

PC: 1003 → 1003 | RD → S/S Bus → m/m: 20 → S/S Bus → 20 CPU

→ Process the Instⁿ } FC

PC ← 1003+1

PC: 1004

{I3}

PC: 1004 → 1004 | RD → m/m: MVI → CPU: MVI } FC (fetch cycle)

↓ decoding

↓ It is a 2B/2W Instⁿ

↓ 1 mem sequence required.

PC ← 1004+1

PC: 1005

PC: 1005 → 1005 | RD → 40 m/m → 40 CPU

↓ Process the Instⁿ

PC ← 1005+1

PC: 1006.

EC ↓ (Execⁿ cycle)

IMR	2MR	—
IMR	IMR	—
IMR	—	—

fetch consumes
7 m/m references (MR)
(7 m/c cycle)

a 32 bit hypothetical processor used to the foll: program

Insⁿ	size (in words)
I_1	3
I_2	2
I_3	1
I_4	3
I_5	1
I_6	2
I_7	1
I_8	

ⓐ assume that program stored in the m/m with starting adⁿ of 3000 decimal onwards, what could be the value present in the PC during the execution processing of a I_6 Insⁿ

addressable m/m space:

3000 – 3011
3012 – 3019
3020 – 3023
3024 – 3035
3036 – 3039
3040 – 3047
3048 – 3051

Prog. Given in word format
Storing in Byte format.
convert word to Byte.

12 × 4
= 48

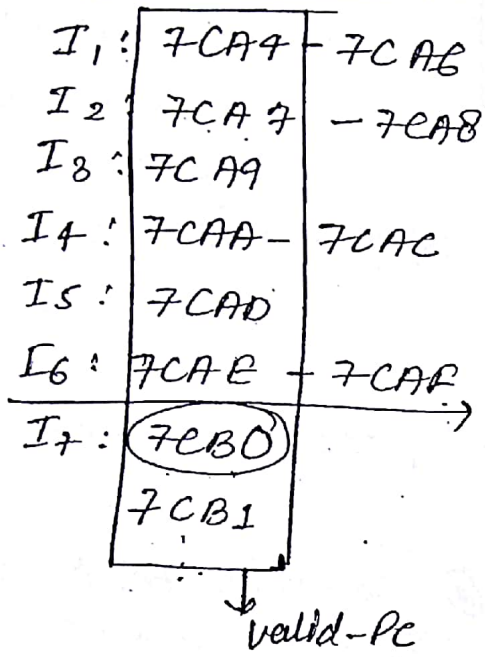
3000
3048

(b) assume that Prog. is stored in a word addressable mlm with starting adrs of (7CA4)$_H$ onwards what could be the value present in the PC during the execn of I_6 Instn.

Every cell capable to hold → 1 word data

7CA4

word addressable space



Q A CPU has 24 bit Instn. Prog. is loaded in the mlm with a starting adrs of 300decimal onwards w/c one of the foll. is a valid Program counter.

(a) 400
(b) 500
(c) 600
(d) 700

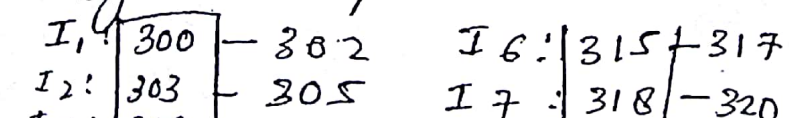
600 is valid PC.

mlm adrs → 300

Soln → Instn size = 24 bit
= 3B

Instn takes 3 cells of mlm space.

Byte addressable space



Q A hypothetical 64 bit processor supports 3 cate gory of Ins^n i.e., cat-1 → 1w long 20
cat-2 → 2w long 40
cat-3 → 3w long 30
Prog. contain 20, 40, 30 Ins^n respectively. How many Bytes of mm space is required to store the prog.

Ans
Program size → 20 cat1
40 cat2
30 cat3.

cat-1 = 1w long (word size = 64 bit = 8B)
= 8B long.

cat-2 = 2w long
= 16B long

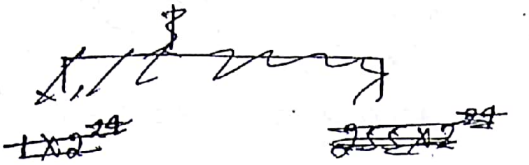
Cat-3 = 3w long
= 24B long

Prog. size in Byte.

cat-1 Ins^n = 20*8B
cat-2 Ins^n = 40*16B
cat-3 Ins^n = 30*24B
Total Space = 1520 B

Q consider a hypothetical processor w/c supports 24 bit add^{rs} bus to refer the mm space & 8bit data bus. Instru^n is design with two field OP code & add^{rs}. Processor supports 200 Ins^n.
a). How many bits are required to encode the

b) When the prog. contain 100 Ins^n. then how many words of mim space is required to store the prog.



word size depend on data bus.

* Ins^n size is 32 bit. $\leftarrow$ 24 add^r bus, 8 data bus.

Prog contain 100 Ins^n every Ins^n is 4 B long.

Ins^n

Ins^n format: | OPcode | Addr | — $\log_2 200$ — 24 bit

$\log_2 200$ → 8 bit

∴ Ins^n size = 32 bits

Prog. size = 100 Ins^n

= 100 * 4 B

= 400 Bytes

Data bus = 8 bit = 1 word

∴ Prog size = 400 words

W/O In^t layout processing Cu's take place

word size depend on data bus
8 bit → 1 word

Processor - 2 - Pro → Inst format change

CI 11 1 →

Execution cycle

① Objective of a execution cycle is Processing of a currently fetched Instruc^n

② Opcode is required to process the instruc^n.

③ Instruc^n format specifies the opcode info^n

④ Instruc^n format is classified into 5 types based on the CPU organization.

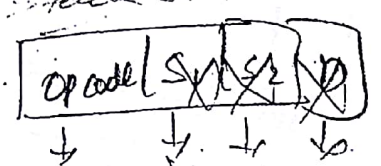
CPU organization is divided into 3 types based on the availability of a ALU operands name as

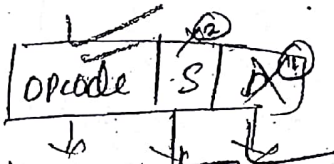
ⓐ Stack CPU → all are implicit.
ⓑ Accumulator CPU
ⓒ General register CPU

} ISA (Instruc^n set arch.) m/c.

ALU require for data manipula^n i/p^u.

ⓐ My: Stack CPU

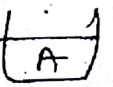




A * B

M M

Stack-CPU

I_1 : push A : 

I_2 : Push B 

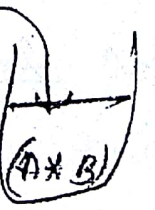
I_3 : MUL

↓

Pop } OF
Pop }
* } PD

A B

A * B



① Stack CPU

① In this organizⁿ ALU o/pⁿ performed on a stack data. so operands are always required in the stack.
After the processing result is also placed into a stack.

② stack is a part of the m/m initialize with a stack pointer (SP register).
- In the stack insert & delete o/pⁿ are always take place at the same end i.e top of stack (tos) it become a default locaⁿ, so implicit -t argument in the instruⁿ design.

③ compatible Insⁿ format is

: Zero adrs Insⁿ format.

↓
type of dpⁿ
↓
ALU o/pⁿ : S1 S2 Destⁿ
TOS TOS TOS

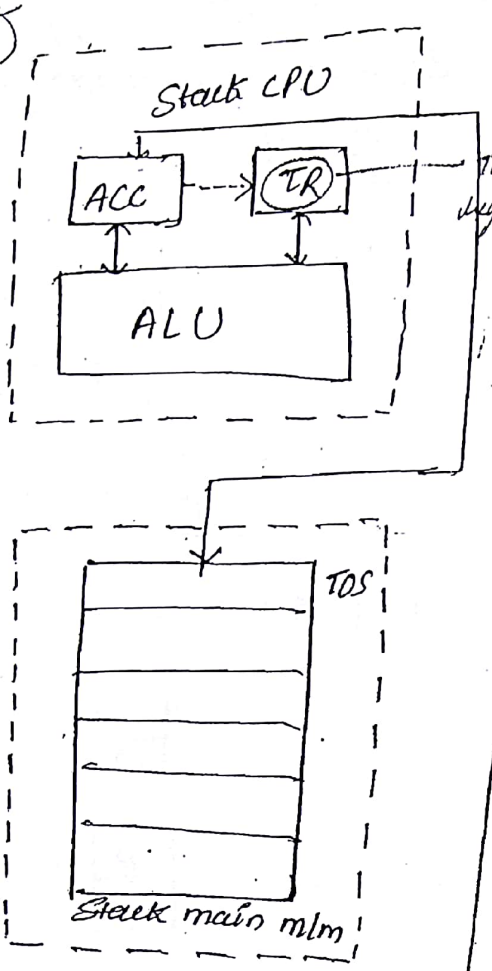
↓ Invokes → If pop give → (second operand)

POP	→ To get the "S2"
POP	→ To get the "S1"
ALU o/pⁿ	→ Process the data
Push	→ Push back the result

↓
TOS ← TOS (ALU o/pⁿ) (TOS) → Top of stack

In this organisaⁿ data transfer Insⁿ are (push & pop). design as 1 adrs instruⁿ.

steps (N)



(2) Prog:

Ins^n: ADD; memory

↓

ADD: Stack-CPU

PC ← PC+ss

↓

ADD : Ins^n format

↓

ALUO/P^n

↓

+

↓

Pop

Pop

+

Push

↓

TOS ← TOS + TOS

FC

EC.

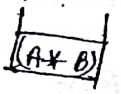
(3) Ex (A*B)+C

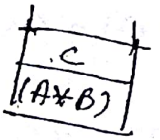
variables are in memory.

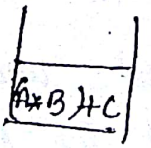
code:

I_1 : Push A ; 

I_2 : Push B ; 

I_3 : MUL ; A*B 

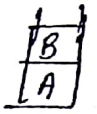
I_4 : push C ; 

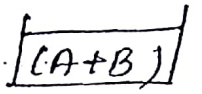
I_5 : ADD ; (A*B)+C 

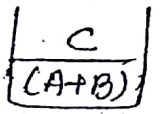
Q $X = (A+B) * (C+D)$ variable ... mic Instⁿ are required to evaluate the stmt on a stack-CPU.

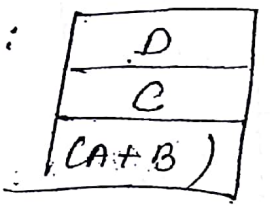
Code:

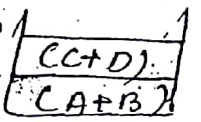
I_1: Push A : 

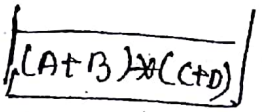
I_2: Push B : 

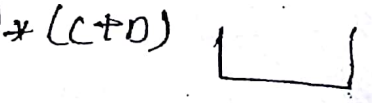
I_3: ADD : A+B 

I_4: Push C ; 

I_5: Push D : 

I_6: ADD : C+D 

I_7: MUL ; (A+B) * (C+D) 

I_8: Pop X ; X = (A+B) * (C+D) 

Q Construct the following code, running on a stack-CPU (stack is initially empty)

Code:

I_1: Push 1B b 3B (b) 4B

I_2: Push x 4B

I_3: Add 1B

I_4: popC 4B

I_5: pushC 4B

I_6: Push y 4B

I_7: Add 1B

I_8: Push C 4B

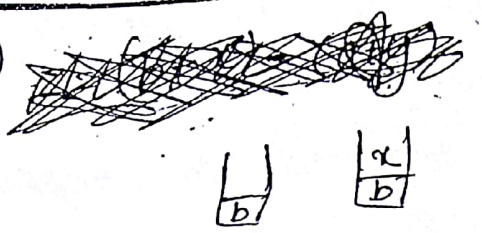
I_9: Sub 1B

I_{10}: pop Z 4B

31B Ans (b)

(a) what is the O/P

(b) If CPU contains (8bit) opcode, (24 bit) addr space then supports

(a) 

I_1 : b

I_2 : x, b

I_3 : b+x (b+x)

I_4 : c = (b+a)

I_5 : c

I_6 : y, c

I_7 : c+y (c+y)

I_8 : c, (c+y)

I_9 : c+y-c y

I_{10} : z=y

z location contain y & stack be empty.

operator → Zero add^{rs} Instn

Push pop are 1 add^{rs} Instn

(b) 31B.

D = (A*C)

(a) How many mc Ins^n are required to complete the prog on a stack CPU.

I1 Push C

I2 Push B

I3 Add B+C

I4 pop A

I5 push A

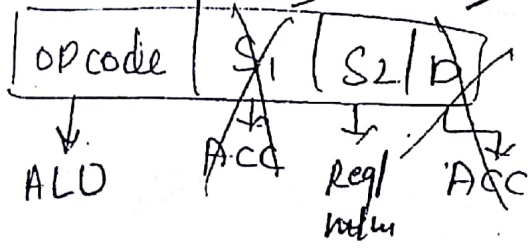
I6 push C

I7 mult (A*C)

I8: Pop D.

(b) Accumulator CPU → two operand are implicit 1 is explicit. 2/2

Implicit no need, remove.



(A + B)

M → M

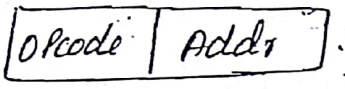
ACC-CPU

A & B in mem can use multiply in acc-CPU NO, because they r in mem.

- Accumulator is implicit we give mem addrs everywhere.
- In this organisation one of the ALU operand is always required in the accumulator & another operand is present either in the register or in the m/m.
- After the data processing result is placed into

② In the CPU design only one accumulator is present so it become implicit.

③ compatable instruction format is:



: one-Addr Instn format

↓

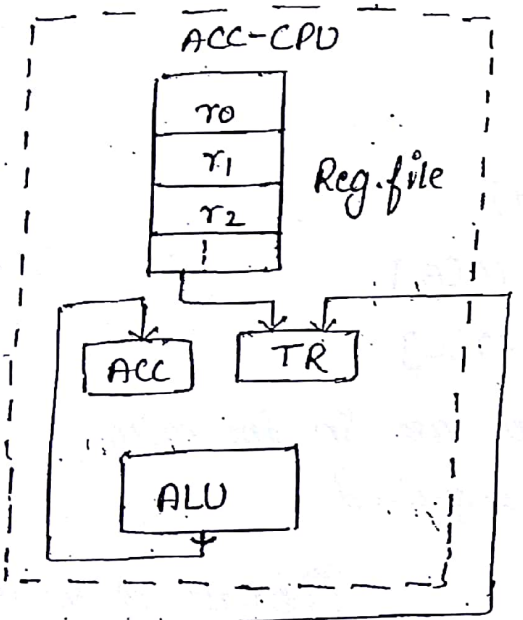
Type of opn

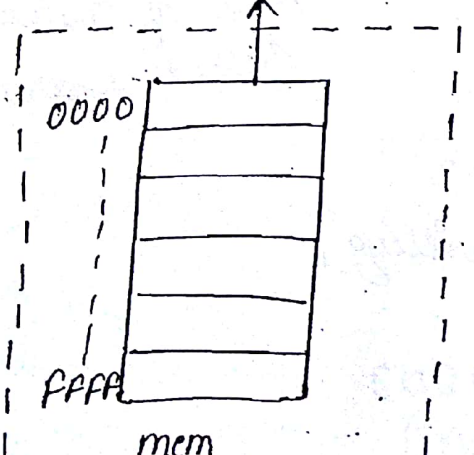
↓

Data transfer o/pn : Source: Mem, ACC; Destination: ACC (MemRD): Load; Mem (MemWR): Store

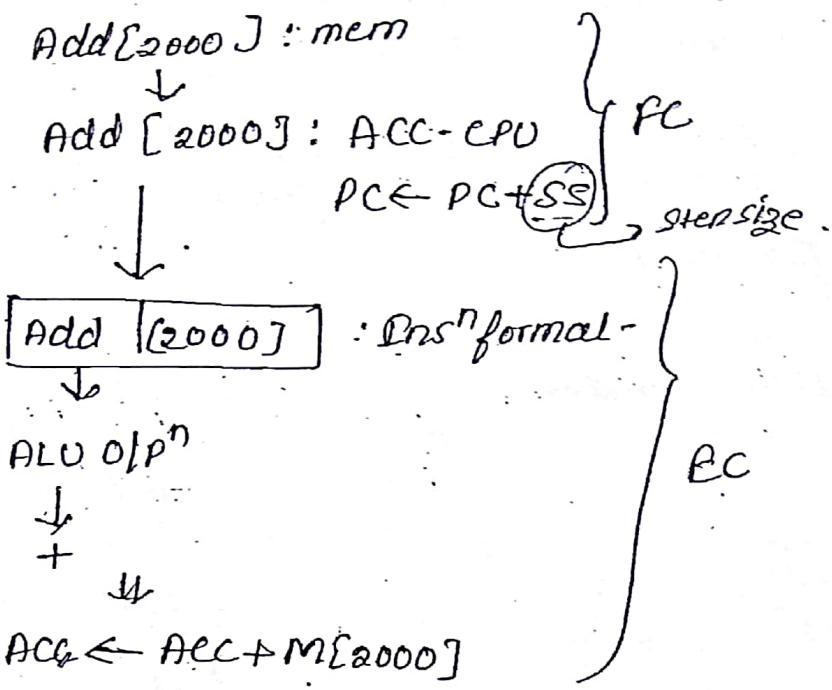
↓

ALU o/pn : S_1: ACC; S_2: Reg/mem; Destiu: ACC





Prog Qns?



Ex (A*B)+C

- variable are in the mem

code:

I_1 : Load A ; ACC ← M[A]

I_2 : MUL B ; ACC ← ACC * M[B]

I_3 : Add C ; ACC ← ACC + M[C]

Q1. X = (A+B) * (C+D) variables are in the mem. How many mle Qns^n are required.

Q2. X = (A+B)/(C+D) variables are in the mem.

(1) Incase of Multip^n order of Evalua^n doesn't matter.

Ans1

I_1 : Load A ; ACC ← M[A]

I_2 : ADD B ; ACC ← ACC + M[B]

I_3 : store T ; M[T] ← ACC [spilling]

I_4 : Load C ; ACC ← M[C]

I_5 : ADD D ; ACC ← ACC + M[D]

Ans 2. $I_1 \rightarrow$ Load C ; ACC $\leftarrow$ M[C]; ACC $\leftarrow$ C

$I_2 \rightarrow$ add D; ACC $\leftarrow$ ACC + M[D] ACC $\leftarrow$ (C+D)

$I_3 \rightarrow$ Store T M[T] $\leftarrow$ ACC; T $\leftarrow$ (C+D)

$I_4 \rightarrow$ Load A ACC $\leftarrow$ M[A]; ACC $\leftarrow$ A

$I_5 \rightarrow$ ADD B ACC $\leftarrow$ ACC + M[B]; ACC $\leftarrow$ (A+B)

$I_6 \rightarrow$ Div T ACC $\leftarrow$ ACC / M[T]; ACC $\leftarrow$ (A+B)/(C+D)

$I_7 \rightarrow$ Store X M[X] $\leftarrow$ ACC; $\leftarrow$ (A+B)/(C+D)

7 Instructions

$A/B \neq B/A$

Order of Evalu required.

In this case divid either (A+B) I St or (C+D).

Q① & ② both managable by user.

In ③ order matter.

Note:
Expand opcode technique is required in the fixed length instru supported cpu design to implement the instruc with various formats. In this process addr field will be appended to a free opcode to generate the low order instruc
∴ low order instruc opcode size n.

Analysis

① variable length Inst supported CPU

- OPcode = 8 bit
- Address = 8 bit

(a) one addr Instn design → High order

OPcode	Addr
8bit	8bit = 16bit.

Fixed → super comp. RISC

variable → general Purpose computer CISC

(b) Zero address Inst Design → low order

OPcode
8bit = 8bit

{no need of a expanded opcode technique}

② Fixed length Instn supported CPU

- opcode = 8bit
- addr = 8bits.

(a) One addrs Instn design

OPcode	Addrs

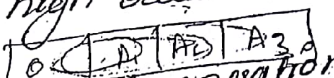
(b) zero add^{rs} Ins^{n} Design

Opcode
16bit = 16bit

{Expand opcode technique is used if & only if there is a free space}

Q. consider a hypothetical, $ CPU w/c supports 6bit Ins^{n} & 4 bit add^{rs}. If there exist two-one add^{rs} Instru^{n}, then how many zero add^{rs} Ins^{n} can be formulated?

Any step 1: Identifiy the high order Ins^{n} format in the CPU.



Step 2: Identify the total no. of operations & Instruc^{n} possible.

Step 3: Identify the free opcodes after allocate the existed Instruc^{n}.

Step 4: calculate the low order Ins^{n} possible in the CPU by multiply the free opcodes with a decoded form of a add^{rs} field.

1222

(1)

6bit	
Opcode	Add^{rs}
2bit	4bit

(2) #dp^{n} Ins^{n}(N) = 2^{opcode}

$= 2^{2}$

$= 4$

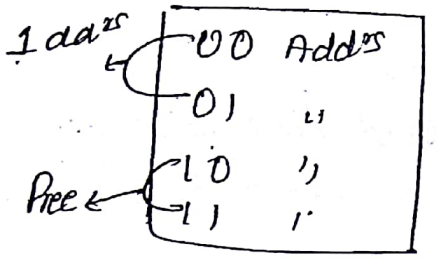
00 Addr.
01 Addr
10 Addr

(4) opcode w/c is derived after alloca^{n} of 1 add^{rs}

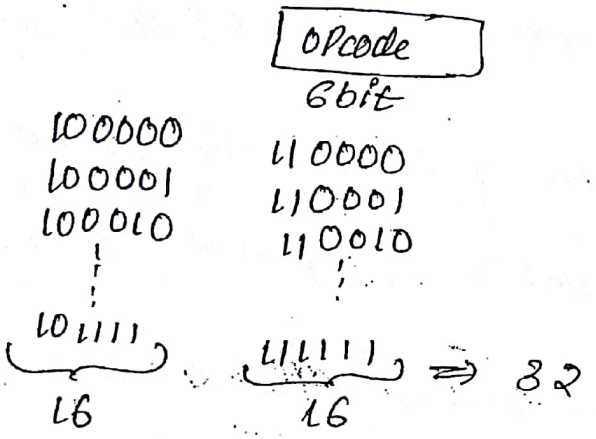
(3) ## free opcodes = (N − ## existed Instrucⁿ)

= 4 − 2

= 2



(4) ## zero addrs Instⁿ in the CPU.



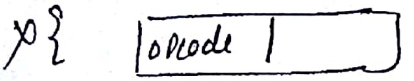
⇒ ## free opcodes * $2^{\text{addrs size}}$

⇒ $2^{*}2^{4}$

⇒ 32.

Q. Consider a 32 bit hypothetical processor w/c supports 1 word Instruction. Instruction is placed in a 256M word mlm. If there exist 8-1 addrs Instruction then how many zero addrs Instruction are formulated.

1 word = 32 bit $\frac{256 \times 10^{20}}{32}$



- word size = 32 bit
- Instⁿ " = 1 word
  = 32 bit

- m/m size = 256 MW
  ↓
  256 M x W
  ↓
  $\log_2$ 256 M x W
  ↓
  $\log_2 2^{28}$
  ↓
  28 bit Adrs

(1) Higher order format.

32 bit

opcode	Adrs
4 bit	28 bit

(2) # opⁿ = 2^4 = 16

(3) # free opcodes = (16 - 8) = 8

(4) # zero adrs Instⁿ = 8 * 2^{28} = 2^{31}

Total # Instruⁿ in the CPU = (# 1-adrs Instⁿ + # 0-adrs Instⁿ)

= 8 + 2^{31}

Q Consider a hypothetical processor w/c supports Instruⁿ format with two fields that is opcode field of 4 bit & 6 bit adrs bit.

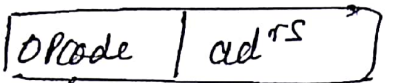
In the Instruⁿ design 4 opcodes are reserved for special Instruⁿ when the reserved opcodes

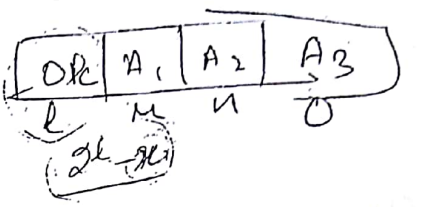
How many no. of Instruct are possible in the cpu.

Any three opcode are free out of 16.

$4 \times 2^6 = 2^8 = 256 + 12 = 268$

New old.





$2^4 = 16$ opcodes

4 opcodes for special Instrn

adrs field will be appended to opcode

So, # special Instrn $= 4 \times 2^6 = 256$

12 opcodes for old Instrn.

$\therefore$ Total Instrn = 12 old Instrn + 256 special Instrn

$= 268$

Q Consider a hypothetical processor WC supports both one adrs & zero adrs instruct. A 24 bit Instruct is placed in a 128 kB mem.

(a) What is the range of 1 adrs & zero adrs instruct possible in the cpu.

(b) If there exist x-1 adrs instruct then how

128 KB → 128×2^{10}

24 bit

① | Opcode | Addr |

7 bit — $\log_2 128$

↓

17 bit

② # opⁿ (N) = 2^7

= 128

Share the opcode blw 1 adᵉˢ & zero adᵉˢ Instⁿ.

	1-adᵉˢ		0-adᵉˢ
①	1	——	127
②	127	——	1

Range of 1-adᵉˢ Instⁿ : { 1 to 127 }

Opcode	Addr
7 bit	17 bit

Range of 0 adᵉˢ Instrucⁿ : $\{(1 * 2^{17}) \text{ to } (127 * 2^{17})\}$

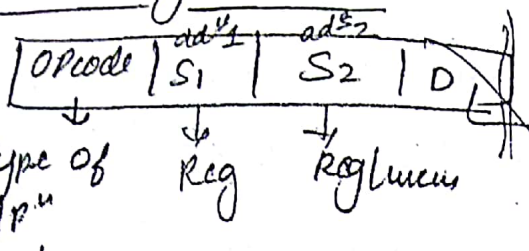
Opcode
24 bit

(b) # free opcodes = $(2^7 - x)$

↳ # zero adᵉˢ Instrucⁿ = $(2^7 - x)\, 2^{17}$

Opcode
24 bit.

② General register CPU



two add^{rs} format
1 opcode with 2 add^{rs}

ALU opⁿ

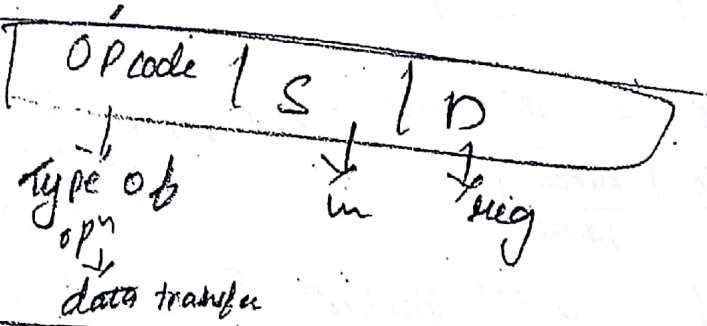
→ (A + B)

Register - m/m ref.

R-M ref.

to make it possible 1 operand shud be in Reg/mem.

ALU data transfer both are two add^{rs}



* Based on the no. of registers possible in the CPU, architecture is divided into 2 kinds.

① Register to m/m reference CPU (CPU with less no. of registers).

② Register to register reference CPU (CPU with more registers)

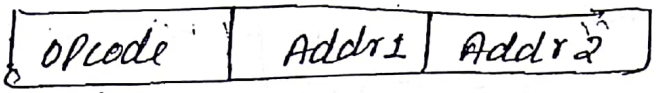
① Register to m/m reference CPU.

① In this organizar IST operand is always required in the register.

② If the 2nd operand is present either in

placed in a source 1 location.

compatible Instrn format is:



: two addrs Instrn format.

↓

Type of opn

↓

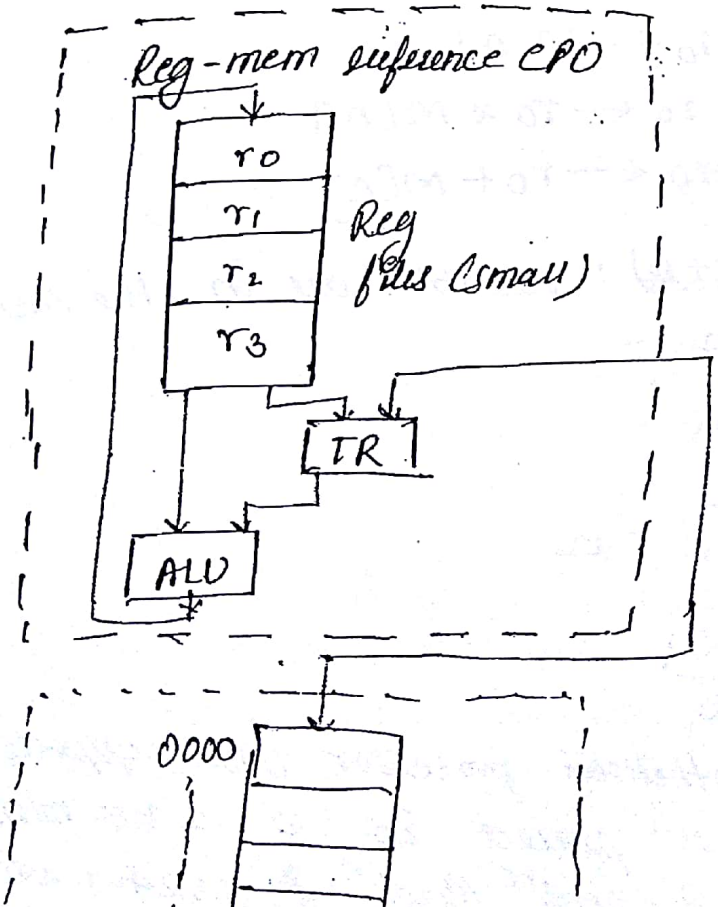
data transfer o/pn : Source — Destinan

mem — Reg
Reg — mem } mov
Reg — Reg

↓

ALU o/pn : S1 S2 Destinan

Reg Reg/mem S1.



Prog.

Instnⁿ: Add r0, [2000]; -mem

↓

Add r0, [2000]; CPU } FC

↓ PC ← PC + SS

Add	r0	[2000]
↓	↓	↓
ALU opn	S_1 & D	S_2

; Instrucⁿ Format

↓

\+

⇓

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

Ex: $(A*B)+C$

variables are in the mem.

code:

I_1 : MOV r0, A : $r_0 \leftarrow M[A]$

I_2 : MUL r0, B : $r_0 \leftarrow r_0 * M[B]$

I_3 : Add r0, C ; $r_0 \leftarrow r_0 + M[C]$

Q $X = (A+B)*(C+D)$: variables are in the mem

I_1 : mov r0, A

I_2 : Add r0, B

I_3 : mov r1, C

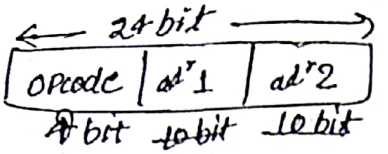
I_4 : Add r1, D

I_5 : mul r0, r1

I_6 : mov X, r0

Q consider a hypothetical processor wk supports 24 bit ~~ofp~~ instrucⁿ placed in a 1 KB mm.

Of these, what 2-addrs Instⁿ & 1024-1 addrs Instr



$\log_2 1K$ $\log_2 1K$

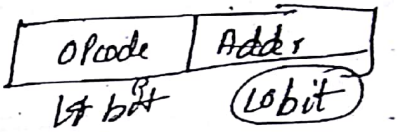
10 bit 10 bit Both are address

8×2^{10}
2^{10}

opn = 2^4
= 16

↳ # free opcodes after allocate = (16 - 8)
the 2-adrs Instn = 8

↳ # adrs Instruction possible



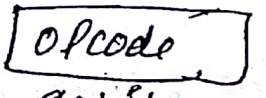
= 8 * 210
= 2^{13}

↓

free opcodes after allocate the 1-adrs Ins
= 8192 - 1024
= 7168

↓

zero adrs Instn possible



= $7168 * 2^{10}$ Ans.

Q consider the computer s/s w/c contain the Instruction format with 3 fields i.e opcode, Register adrs fields & m/m adrs field. s/s suppor. 60 Registers a 32 bit Instruc^n is placed in a 16 MB m/m. calculate the foll.

(a) How many Instructions are possible.

(b) If there exist "x" - 2 adrs Instru^n w/c refers both registers & m/m. Then how many 1 adrs Instru^n m/m reference are possible. asking keep

(c) If there exist x - 2 adrs Instru^n w/c in format refers both register & m/m & why register extra merg bit reference 1 adrs Instruc^n then how many zero adrs in ~~regis~~ Instru^n are formulated.

← 32 bit →

opcode	adrs reg	adrs mem

2 bit   60 R   16 MB

↓   ↓

$\log_2 60$ $\log_2 2^{24}$

↓   ↓

6 bit   24 bit

(a) ~~no of Instr 2^2 = 4~~, no of op^n $= 2^2 = 4$

(b) ~~no. of free opcode = (4-x)~~   ask mem reg

~~= (4-x) × 2^6~~ → extra merge bit

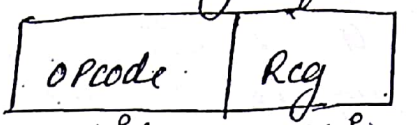
(c) (b) # free opcodes after allocate the 2-Addr Insn $= (4-x)$

↳ ~~##~~ 1-adr m/m ref. Ins^n

$= (4-x) * 2^6$

(c) # free opcodes after allocate the 2 adrs Instn = $(4-x)$

↳ # 1-adrs reg. ref Instn



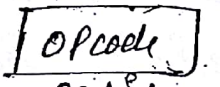
$= (4-x) * 2^{24}$

↓

free opcodes after allocate the 1-adrs reg ref. Instruc^n $= ((4-x)2^{24}) - c$

↓

2 adrs Instn

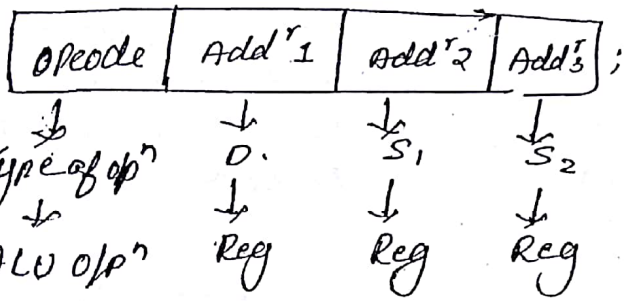


$= \left[\left\{(4-x)2^{24}\right\} - y\right] 2^6$

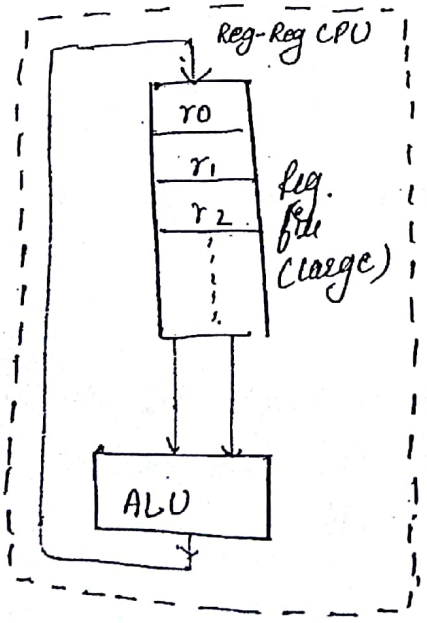
Register to Register reference CPU

23/July/2017
Sunday
class-4

- In this organisation ALU operation are performed on a register data so both of the operand are required in the register.
- After the manipulation result will be placed into a separate register.
- compatible instruction format is :



- In this organisation load & store instruction are used as a data transfer operation.
- To transfer the data b/w the m/m & ~~add^rs~~ Register respectively. These are assigned as a two add^rs Ins^tr^n



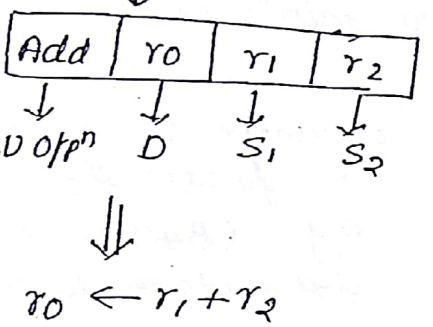
Prog Instrⁿ

Add r0, r1, r2 ; mem

↓

Add r0, r1, r2 ; CPU

PC ← PC + SS } FC



r0 ← r1 + r2

Ex (A*B)+C variables are in m/m.

code:

I1 : Load r0, A

I2 : Load r1, B

I3 : Mul r2, r0, r1

I4 : Load r3, C

I5 : Add r4, r2, r3

Q X = (A+B) * (C+d) variables are in m/m.

I1 : Load r0, A

I2 : Load r1, B

I3 : Add r2, r0, r1

I4 : Load r3, C

I5 : Load r4, D

I6 : Add r5, r3, r4

I7 : Mul r6, r2, r5

I8 : Store X, r6

load store data transfer instrⁿ in the O/Pⁿ

Q consider a hypo. processor used to execute the

the register. How many min. reg. required to evaluate the stmt w/o spilling.

↳ intermediate result always requires 1 register.

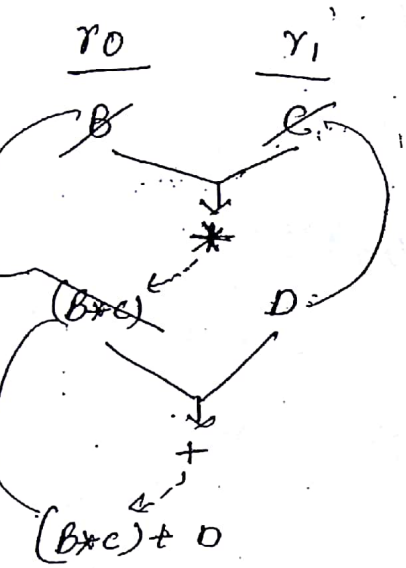
~~I_1 : load r_0, B~~

~~I_2 : load r_1, C~~

~~to minimise the regi.~~

Re-use the registers to minimise the register count.

r_0 contains B not required in future replace it by (B*C), Below operation store in register.



M[A] = (B*C)+D

I_1 : Load r_0, B

I_2 : load r_1, C

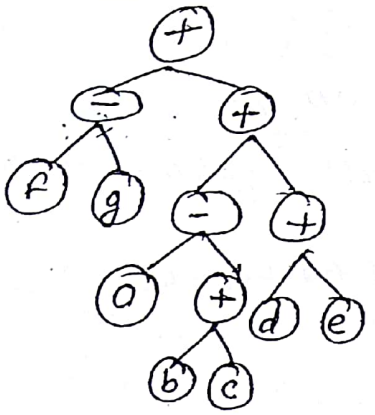
I_3 : Mul r_0, r_0, r_1

I_4 : load r_1, D

I_5 : Add r_0, r_0, r_1

I_6 : Store A, r_0

Q Consider the foll. expression tree, executed on a hypo. CPU. All the variables are present in the m/m. Operators in the tree always accepting the operands from the registers. If no intermediate

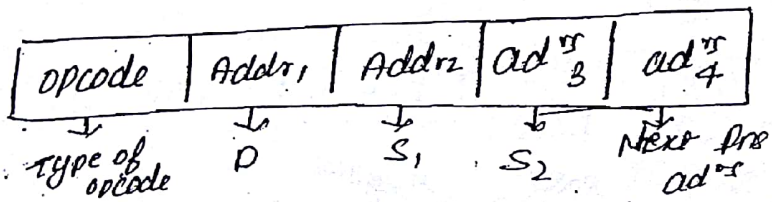


Evaluate from bottom to top

I_1 : load r_0, b
I_2 : load r_1, c
I_3 : $r_0 \leftarrow r_0 + r_1$ — $a+r_1$ stored in either r_0 & r_1 because b & c values not required further
I_4 : load r_1, a
I_5 : $r_0 \leftarrow r_1 - r_0$
I_6 : Load r_1, d
I_7 : Load r_2, c — 3 registers used
I_8 : $r_1 \leftarrow r_1 + r_2$
I_9 : $r_0 \leftarrow r_0 + r_1$
I_{10} : load r_1, f
I_{11} : load r_2, g
I_{12} : $r_1 \leftarrow r_1 - r_2$
I_{13} : $r_0 \leftarrow r_0 + r_1$

---

* 4 address Instruction Format



data transfer
data manipulation
transfer of control signal

Note → PC is a compulsory register in the CPU used to hold the adrs of a next Instruction during the execution of a current Instruction .: four adrs format is not in the practice

Instruction Execution Sequence

- To analyse the execn flow of an instrucn, let us consider accumulator CPU as a reference model.

Ex. 8085 µP

Expression: (A + B)
↓ ↳ mem
mem

variable → name of the mem cell.
A, B are variables here.

Code:

I1 : Load A : ACC ← M[A]
I2 : Add B ; ACC ← ACC + M[B]

Variable	Adrs	Cell
A	2000H	C9H
B	2040H	DAH

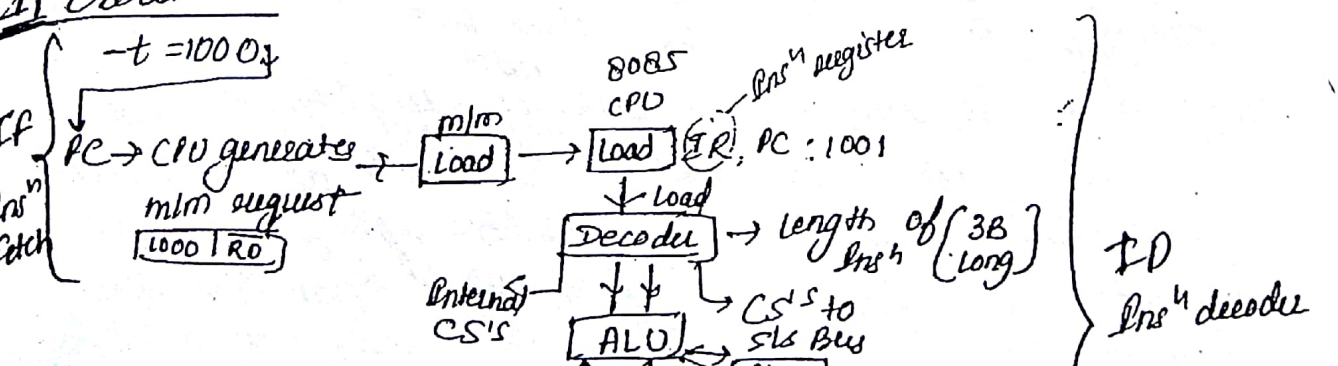
C prog. compiler job
prog. allocn
48H
24H

Prog storage

Org 1000

1000	Load
1001	00
1002	20
1003	A0D
1004	40
1005	20
1006	Next

I1 Execution



PC:1001 → [1001 | RD] → mem [00] → CPU [00] zReg.
PC:1002

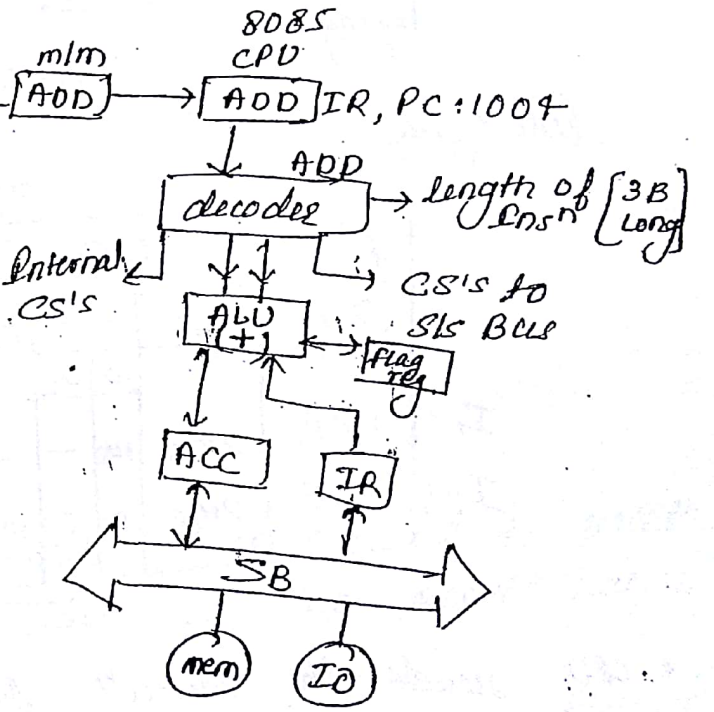
PC:1002 → [1002 | RD] → mem [20] → CPU [20] wReg
[PC:1003]

Operand fetch (OF)

IR[Adrs] → CPU generates the request → m/m [C9H] → CPU [C9H] Acc
{w = Regpair}
[2000 | RD]

I2 Exec

PC [1003] → CPU generates m/m request → m/m [ADD] → 8085 CPU [ADD] IR, PC:1004
[1003 | RD]

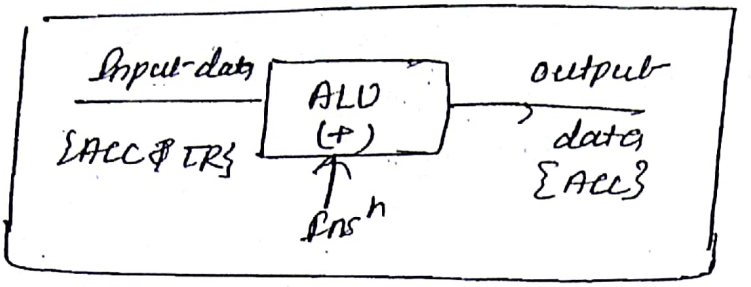


PC:1004 → [1004 | RD] → mem [40] → CPU [40] zReg
PC:1005

PC:1005 → [1005 | RD] → mem [20] → CPU [20] wReg
[PC:1006]

Operand fetch (OF)

Process data (PD) & write Back (WB)

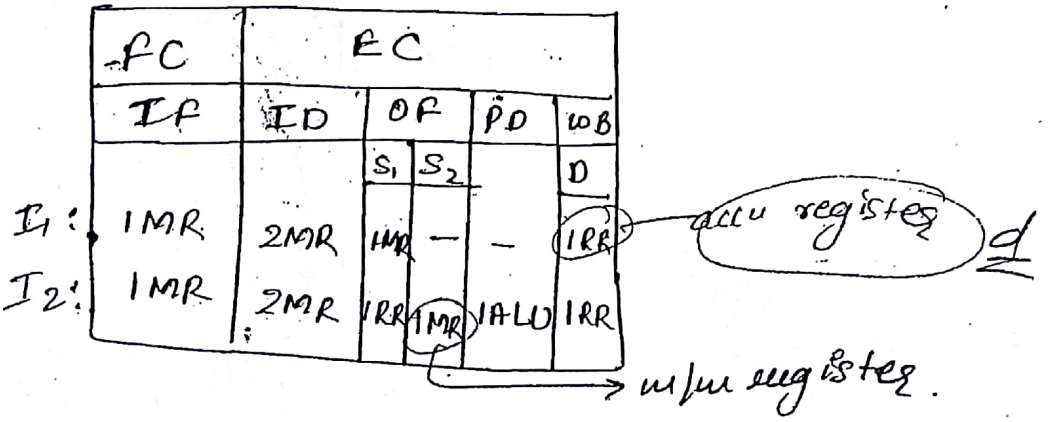


O/P

ACC : 1 1 0 0 1 0 0 1
TR : 1 1 0 1 1 0 1 0
ACC : (1) 1 0 1 0 1 0 1 1

↳ flag

Inst^n cycle



- CPU reads the instruⁿ from the m/m based on the PC called as Instruction fetch.
- IR is a mandatory register in the CPU, used to hold the currently fetched opcode to decod
- during the decoding process, CPU enables the corresponding H/W to perform the o/pⁿ & also fetch the ~~Data~~ remaining part of the Instruct from the m/m when the Instruⁿ size is greater

- During the operand fetch CPU read the data either from registers or from m/m based on the addressing modes.
- Input data is processed on a enabled h/w called as process data later result will be stored into a destination called as write back.
- In the CPU design extra storage is provided to hold the exception of a o/p'n called as flag register.

* PSW (Program status word)

PSW = ACC + Flag Register

- Flag is a flip flop.
- Flip flop is a Bistate component used to hold the 1 bit information.
- Flags are categorised into two kinds.
  1. conditional flags
  2. control flags
- (1) conditional flags → These flags are set or reset Based on the result nature of of a ALU.
- With reference to a 8086 PSW, 6 conditional flags are present name as:
  - (a) carry flag
  - (b) Parity flag
  - (c) Auxiliary flag.
  - (d) zero flag

Condi'n with flag. result compare.

(a) carry:

cond": " Is there an extra bit out of MSB " → T = set = 1 = C
→ F = reset = 0 = NC

- used in the unsigned arithmetic, to hold the range exceeding status.

write opprag Ko plflo lines on rain

(b) Parity

cond": " Is the ALU o/p (Acc) contain even no. of 1's ' → T = set = 1 = PE (even parity)
→ F = reset = 0 = PO (odd parity)

Ctrl + PrtScn

- This flag is used in the serial data communication to prepare the error correction & error detection code.

(c) Auxilary carry:

Oporage opposite (lines on rain
Kopflo

cond": " Is there an extra bit from lower nibble (3rd position) to higher nibble (4th position) → T = set = 1 = AC (Auxilary carry)
→ F = reset = 0 = NA (NO auxiliary carry)

- This flag is used in the bcd arithmetic to indicate the range exceeding condition of a lower nibble.
- it is set when the lower nibble is greater than F. (carry flag is also used in the BCD arithmetic to indicate the range exceeding condin of a higher nibble. it is set when the higher nibble greater than the F.

BCD format is used to represent the decimal data. It is 2 kind:-

(1) unpacked BCD
(2) packed BCD.

① In the unpacked BCD format each digit represent with 8 bit.

Ex 23: 00000010 00000011
2 3

In this format 246 codes are wasted so alternative is required i.e packed BCD.

② In the packed BCD format each digit represent with 4 bits.

Ex 23: 0010 0011
2 3

In this format 6 codes are unused. (A to F) so adjustment is required to report the result i.e add 6 to a nibble when the nibble greater than 9.

Ex 26 ——— 0010 0110
27 ——— 0010 0111
53 ——— 0100 1101 [4DH]

AC=0
CY=0

Lower Nibble {LN: 13} < F: AC=0
{HN: 5} < F: CY=0

Justify

PSW
- ACC → 4D
- Flag reg.
  - AC → 0
  - CY → 0

Any 4D

~~Ex~~ 4D

6

53

16 | 19 — quotient

1 (3) — remainder

(13) (16) — Base 16.

b) 19 | 1

16

3

Ex 28

29

57

0010 1000

0010 1001

0101 0001 [51H]

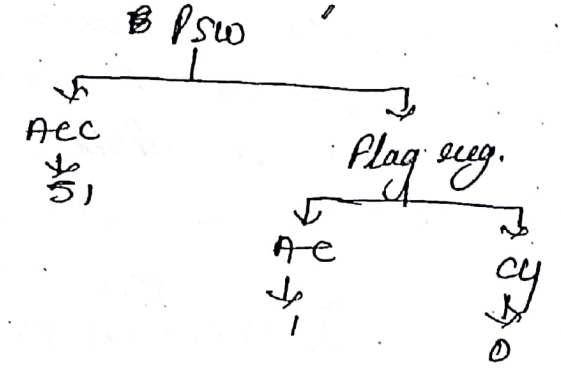
AC = 1

CY = 0

{LN : 7} > F ; AC = 1

{HN : 5} < F ; CY = 0

Justify



Ans 51

{ 1 < 9 & AC = 1 ∴ Actual > F
LN
so add '6' to LN

5 < 9 & CY = 0 ∴ Actual is 5
HN
No need to add '6' to HN }

51

6

57

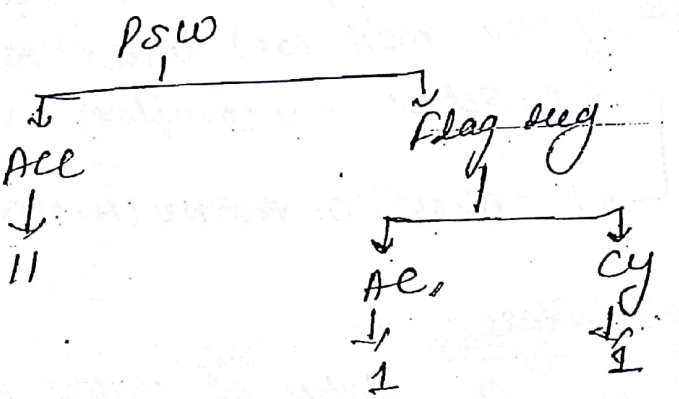
Ex

88	1000	1000
89	1000	1001
177	0001	0001 [11H]
		AC=1
		Cy=1

{LN; 17} > F; AC=1

{HN: 17} > F; Cy=1

Justify



Any 11

{ 1<9 & AC=1 ∴ Actual LN > F, So add 6 to LN

1<9 & Cy=1 ∴ Actual HN > F, So add 6 to HN }

11
66
(Cy) 177

(d) Zero flag

cond: " Is the ALU o/p (Acc) value zero " → T = set = 1 = Z
→ F = reset = 0 = NZ

value	status
0	1
≠0	0

(e) sign flag -

condⁿ → "Is the MSB bit of a ALU o/p (acc) contain "1" → T = set = 1 = NG (-ve logic)
F = reset = 0 = PL (+ve logic)

(f) overflow condⁿ

cond: there is a carry into the MSB & no carry out of the MSB (or) vice versa"

→ T = set - 1 = OV (overflow)
→ F = reset = 0 = ~~NO~~ NV (NO overflow)

(g) control flags

Based on the status of control flags some of the o/pⁿ in the H/w is controlled according to a 8086 PSW, 3 three control flags are present.

(h) TRAP flag → 1: single step prog. execⁿ (-t); trace
0 (At a time Prog. execⁿ (-g): go
at every step it generate trap.
want to see final result
It will reset trap flag. It does
generate trap flag in bio.

(i) Interrupt flag → 1: Enable the Interupt (EI)
0; Disable the Interrupt (DI)

Interrupt is signal m/c is generated by the low speed component out of 3 component, IO is low speed component.

Door is open Interrupt flag is 1
Door is lock " " 0 → stud stands o/s

Direction flag → 1 : Auto decrement (STD) ← set direct
0 : Auto Increment (CLD) ← clear direct

Q consider the foll. 2's complement data & perform the arithmetic add^n o/p^n. what is the status of a carry, zero, sign & overflow flag in the computation.

Data 01100101
10011011

0 1 2 3 4 5 6 7 → positions.

01100000 → Accumulator

Carry = 1 (CC)
Parity = 0 (PO) (odd 1 in accumulator)
AC = 1 (AC)
Zero = 0 (NZ) value not = 0 flag become zero
Sign = 0 (PL)
OV = 1 (OV) Any (1,0,0,1)

Q consider the foll. 8 bit data computa^n, what is the status of a carry, zero, & overflow flags in the computation.

(-121) + (-69)

11111001
11000101
0

64
32
96
16
112
8
120

121 : 01111001
↓
2's complement
↓

69 : 01000101
↓
2's complement
↓

(1) 1000 0111
1011 1011
01000010

Carry = 1
Parity = 1
AC = 1
Zero = 0
Sign = 0
OV = 1

Ans (1, 0, 1)

Addressing Modes (AM)

data location specified by AM | AM show the path of respective entity.

Instruction type
opcode
→ type of ins
→ type of mode used in the ins
→ where operand are present

(1) Addressing modes shows the location of required object. object may be the data or instruction.

(2) output of a AM is effective adrs (EA)

(3) Effective adrs is the actual adrs of a object ∴ object equal to EA.

Object = [EA] sq. bracket indicate content of EA

(4) In the instruction design AM is implemented

① Implicit : opcode itself indicate the type of AM.

② Explicit : mode field is used in the instru formal- to indicate the type of ~~opera~~ AM. i.e.,

opcode	"mode"	Addr

↓

~~$\log_2$ # AM~~

$\log_2$ # AM

Ex. 8 AM used

$\log_2 8 = 3$ bit

③ Different symbols are used to refer the various AM.

I. ~~#~~ / I — Immediate AM

II. RegName — Reg. AM

III. [ ] — Direct AM

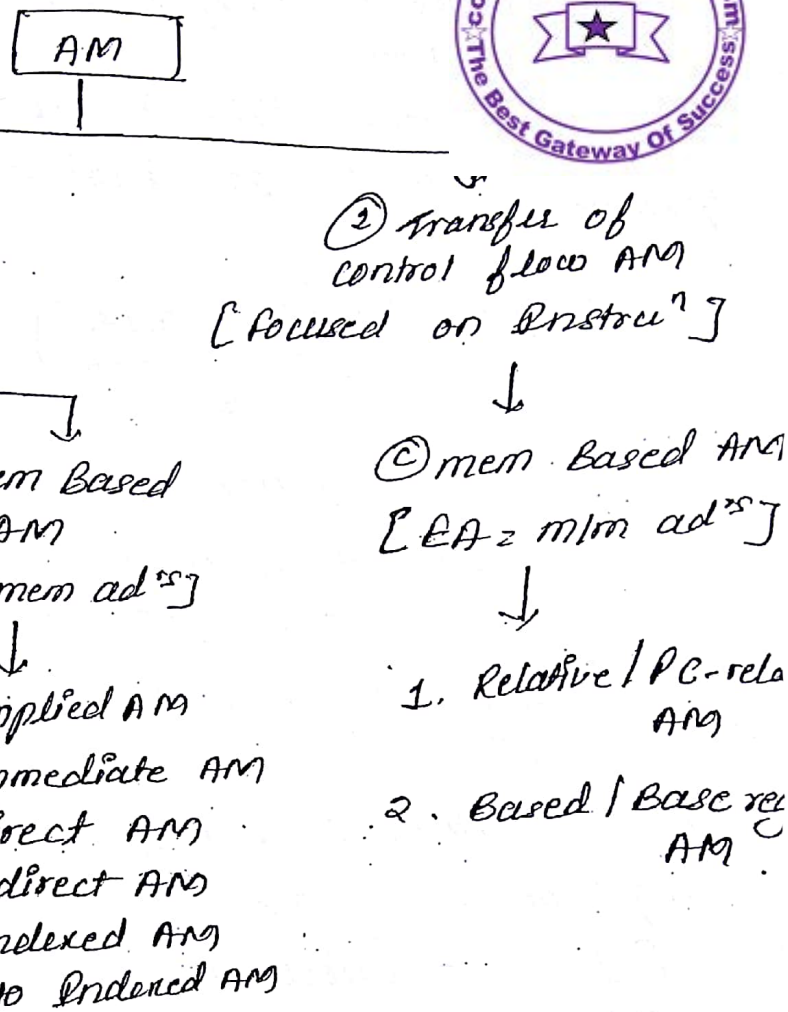
IV. @ / ( ) — Indirect AM

V. Index RegName — Indexed AM

VI. +/- — Auto Indexed AM

⑥ Different types of Addressing modes used in the computer s/s is as follows.

Flow chart in next page.



① Sequential control flow AM

- These modes are concentrate on the data flow so data transfer & data manipulation Instru are designed with a sequential control flow AM.
- Data is present in the computer in two possible places registers & m/m. so there are two types.

(A) Register Based AM.

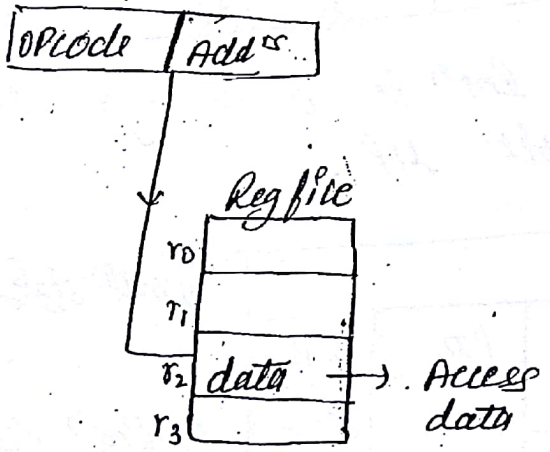
① Registes Based AM

- These modes are used to access the data when it is in the register. So EA is register name.
- In this category only 1 addressing mode is employed.

(1.1) Registes AM.

This mode is used to access the local variables. In this mode data is present in the registes, the corresponding register name will be maintain in the add^{rs} field of a Instru^{n}

Ins^{n} Design.

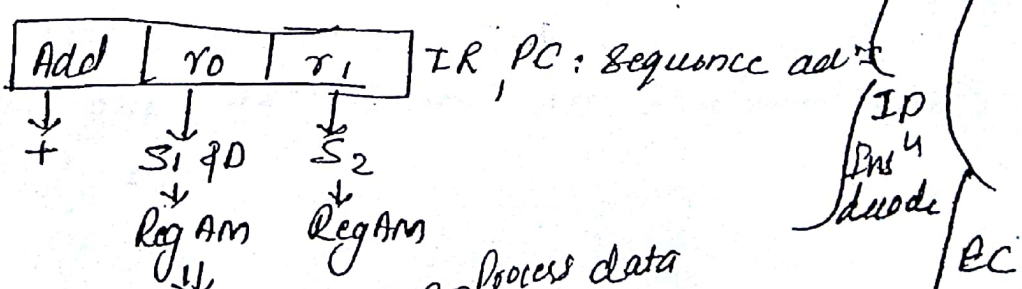


RR → register Reference
Register is internal comp. of CPU.
Same speed as CPU. w/c
So no cycle is required.
RR → negligible

Actions

IRR → to access the data

Ex 1 word ~~design~~. Ins^{n}



IC:

PC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
IMR	—	IRR	IRR	IALU	IRR

Q If a above Instⁿ is 5 word long.

FC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
IMR	4MR	IRR	IRR	IALU	IRR

Q If a above Instⁿ is a 12 byte long execute on a 16 bit µP.

PC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
IMR	5MR	IRR	IRR	IALU	IRR

word size = 16 bit = 2B.

∴ Instⁿ size = 12B = 6W

(2) Mem Based AM

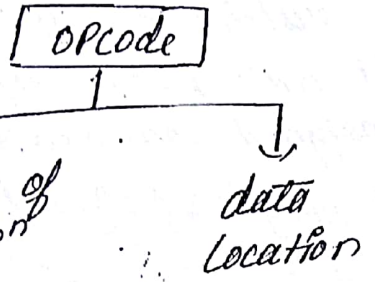
These modes are used to access the data when it is in the m/m. So effective adrs is m/m adrs.

Different AM are employed in this category

(2.1) Implied / Implicit AM

- In this mode data info^n is present in the opcode itself.

Instra^n Design.



Ex STC : set carry

↓

CY=1

Ex CLC : clear carry.

↓

CY=0

Ex ADD : on a stack CPU

↓

Pop
Pop
+
Push.

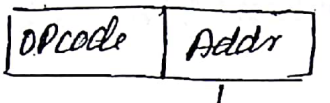
all zero adrs Ins^n is derived from Implied.

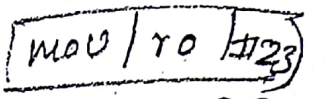
- All the zero add^{rs} instru^n are designed with a implied AM.

(2.2) Immediate AM

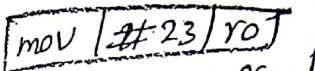
- This mode is used to access the constants.
- In this mode data is present in the add^{rs} field of a Instruc^{n}.

Ins^{n} Design:





R0 ← 23



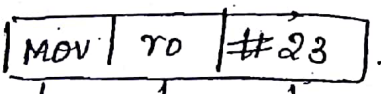
Not Poss

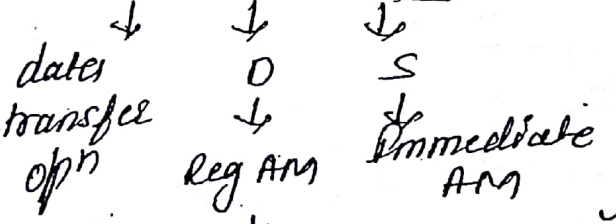
#23 ← R0

Limitation

- Range of constants are restricted by the size of add^{rs} field ie n bit add^{rs} field support.
  {0 to $(2^n - 1)$} unsigned constant or
  {$-(2^{n-1})$ to $+(2^{n-1} - 1)$} range of signed constant.
- Immediate mode is not used as a destinat^{n} mode becoz const doesn't have the storage functionality.

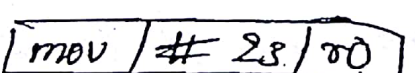
Ex





R0 ← 23

Ex



23 ← R0 : Invalid

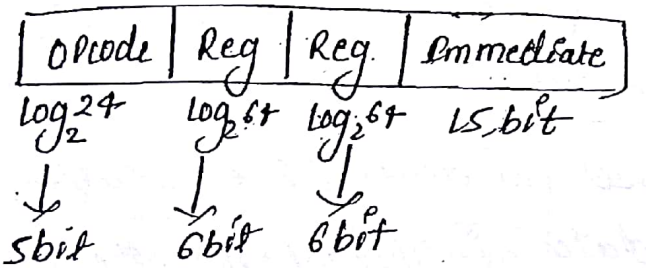
Q Consider a 32 bit hypo. proce^r w/ c support 1 word Ins^n. Ins^n format contain opcode, 2 reg. ad^{r}s field & immediate field. CPU Inst^n set contain 24 Instru^n & supports 64 registers. What is the largest unsigned constant possible in the Instru^n.

Word size = 32 bit

Instru^n size = 1W = 32 bit

64 → $\log_2 64$ → 6 bit

64 → $\log_2 64$ → 6 bit



Ans. $(2^{15}-1)$

(1) Smallest unsigned constant = 0

(2) Largest " " = $(2^{15}-1)$

(3) Smallest signed " = $-(2^{15-1})$

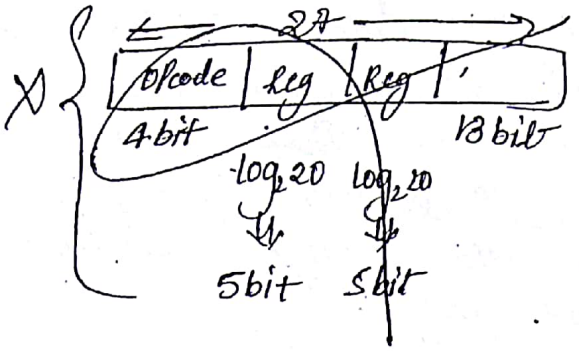
(4) Largest " " = $+(2^{15-1})$

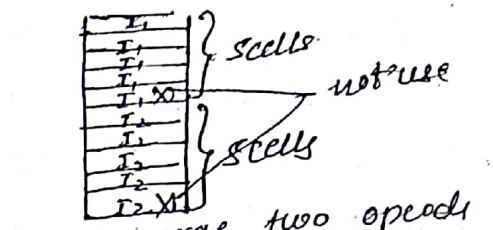
$= +(2^{14}-1)$

(5) # unsigned / signed constants = 2^{15}

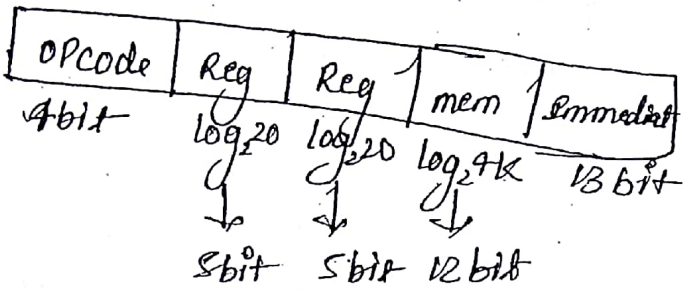
4-2016

Q A hypo. computer supports Instru^n format with 4 bit opcode field, 2 register operands & 1 immediate operand & 13 bit immediate field. Processor





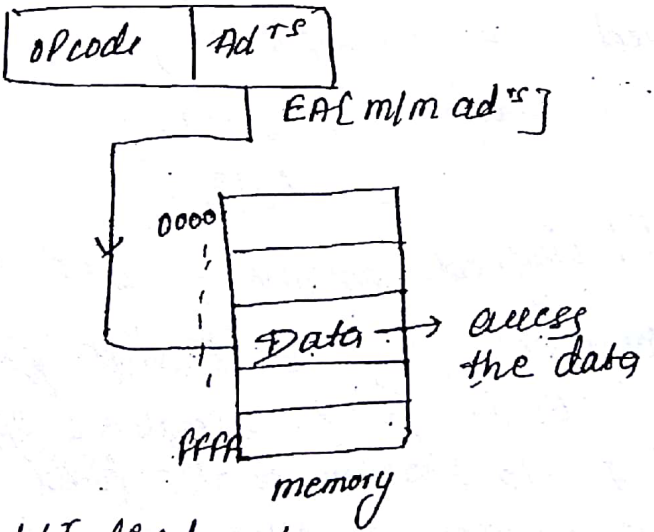
can't merge two opcode in 1 cell.



Ins^{n} size = 39 bits
≅ 5B

2.3. Direct / Absolute Mode

- This mode is used to access the static variable.
- In this mode data is present in the m/m, the corresponding m/m adrs will be maintained in the adrs field of a instruction



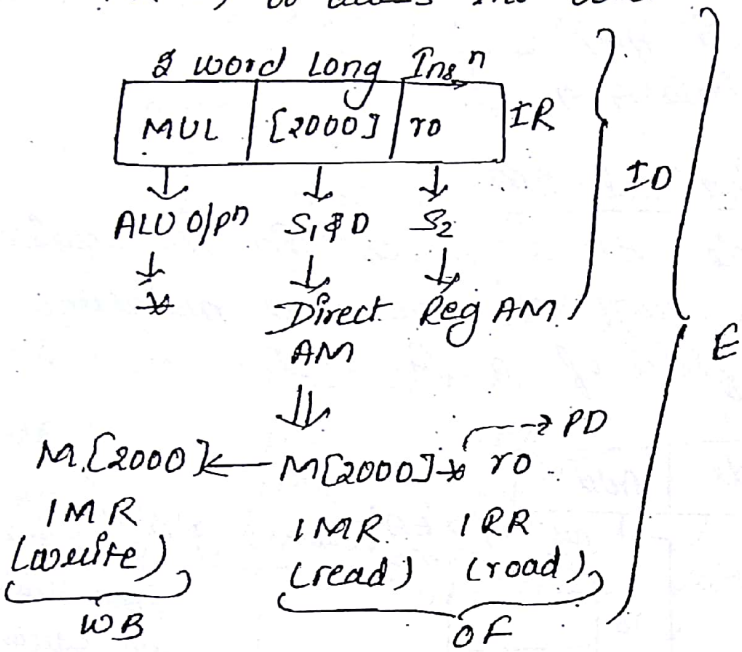
m/m register is countable here.

EA = Addr field value

Data = [EA]
= [m/m adrs]

Actions

IMR → to access the data



FC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
IMR	IMR	IMR	IRR	IALU	IMR

P J
76B2
m/m adrs where EA possess

Q.4. Indirect Addressing modes

- This mode is used to implement the pointers.
- In this mode adrs field of the instruct contain adrs of a Effective adrs i.e.,

EA = [adrs field value]

Data = [EA]

& only M = [[adrs field value]]

1st reference → to get the 'EA'
2nd reference → to access data.

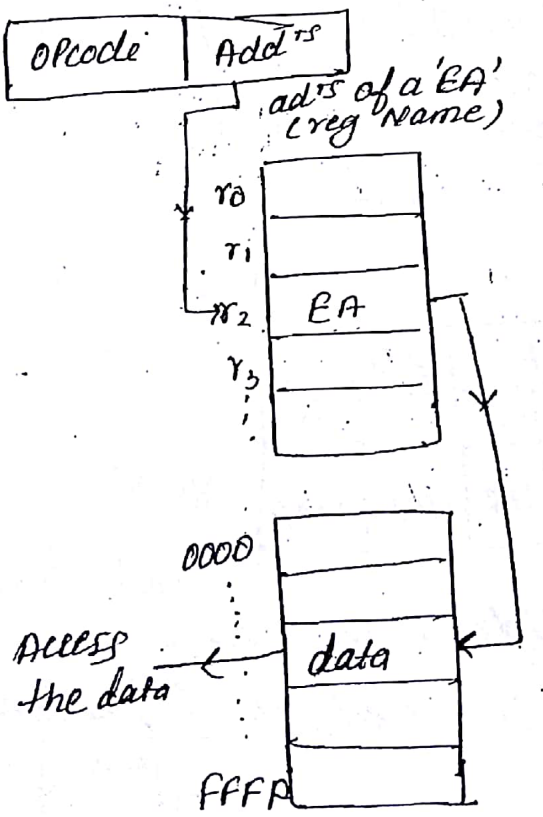
Based on the availability of a EA, Indirect AM mode is divided into two types.

① Register Indirect AM.
② M/m Indirect AM.

2.4.1 Register Indirect AM

In this mode EA is present in the register. The corresponding register name will maintain in the addrs field of a instruction.

Instruction design:



→ When reg is Sq. bracket from that reg. Special value is access. not data.

→ Reg w/o brack Reg is by def used to store the data. So no need to write Reg in Sq. bracket

EA = [Addrs Field value]
↓
Reg. Name

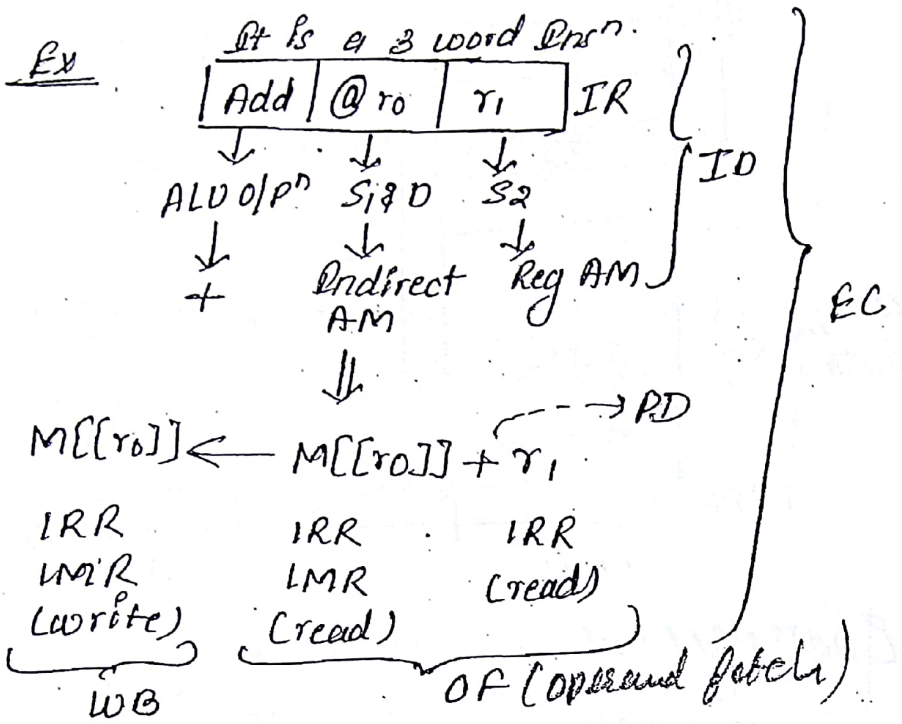
Data = [EA]

= [S. Reg Name]

Actions

IRR → To get the EA
IMR → To access the data.

Ex

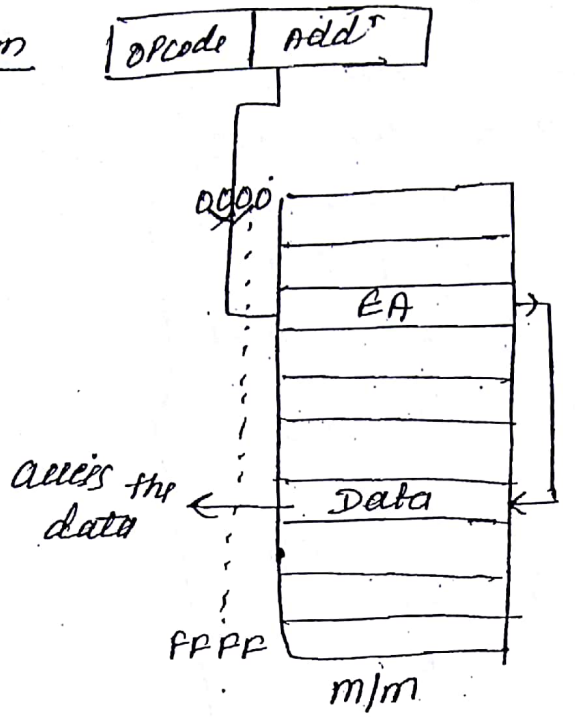


FC	EC				
IF	ID	OF		PD	WB
		S1	S2		D
IMR	2MR	IRR IMR	IRR	ALU	IRR IMR

2.4.2 Memory Indirect AM

- In this adrs mode EA is in the m/m. The corresponding m/m adrs will be maintained in the adrs field of a instr^{n}

Ins^{n} design



EA = [Addrs Field value]
↓
mem addrs

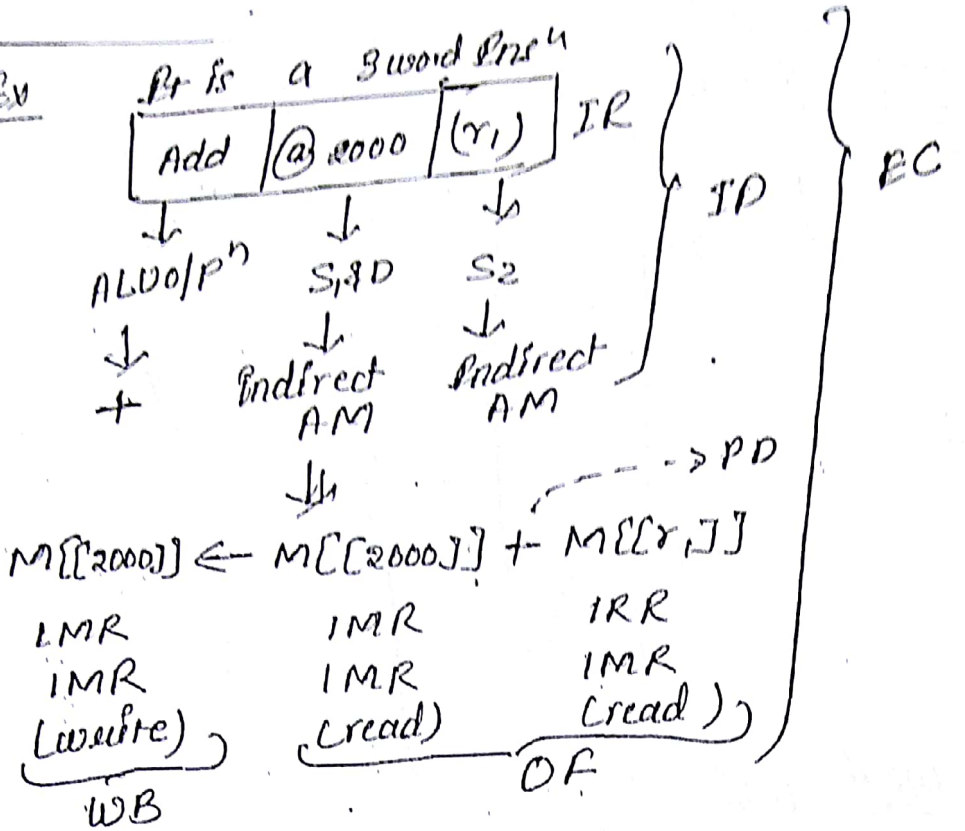
Data = [EA]
= [[mem addrs]]

Actions

IMR → To get the 'EA'

IMR → To access the data.

Ex



FC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
IMR	2MR	2MR	IRR IMR	IALU	2MR

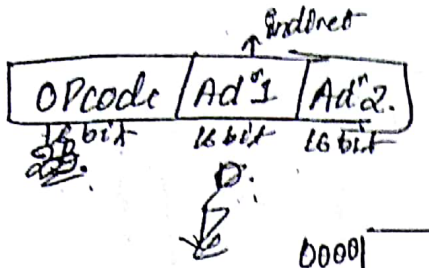
IRR → register reference

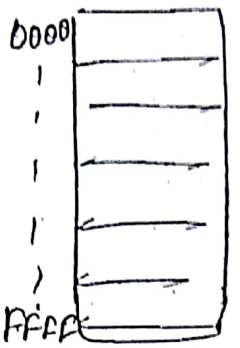
IMR → mlm reference.

Q. A hypothetical processor 1 word opcode w/c contains the ~~ad~~ AM implementaⁿ along with a type of oⁿ, processor supports 2 adʳˢ instruᶜⁿ format where Adʳˢ 1 is also used as a destiⁿ, adʳˢ 1 uses the Indirect adʳˢing & adʳˢ 2 uses the direct adʳˢing. Processor supports 64 KB mlm.

a) How many bits are required to encode the instrᶜⁿ when word length of processor is 16 bit.

HW





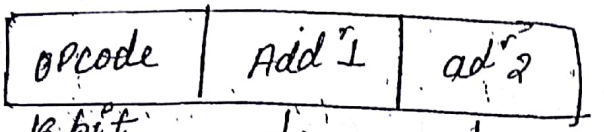
64-KB

$\log_2 2^{16}$

16 bit

$= 2^{16}$

Word size = 16 bit



S_1 & D (Indirect AM)

$\log_2 64K$

16 bit

S_2

direct AM

EA

mm addrs

$\log_2 64K$

16 bit

when in 9th Indirect by default it is mm Indirect

Ans^n size = 48 bits

= 3 words

FC	EC				
IF	ID	OF		PD	WB
		S_1	S_2		D
1MR	2MR	2MR	1MR	1ALU	2MR

{5 mlc cycles are required}

Operand accessing.

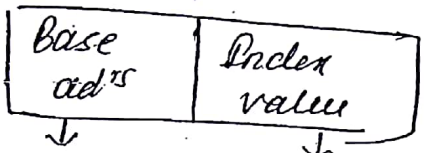
S_1 S_2 D

Total IAB fetch is 3.

2.5. Indexed / Base Indexed Addressing Mode

- This mode is used to access the random array elements from the m/m (to implement the array implementation in the h/w).

- In this style of accessing two parameters are required they are Base addrs, Index value.



Starting addrs of an array → Present in the addrs field of an instn as a constant.

Position of an array element → Present in the Index Reg (Ri)

- In this mode EA is obtain by adding the constant value to a content of the index register, i.e.,

- EA = Addrs field value + [Ri]

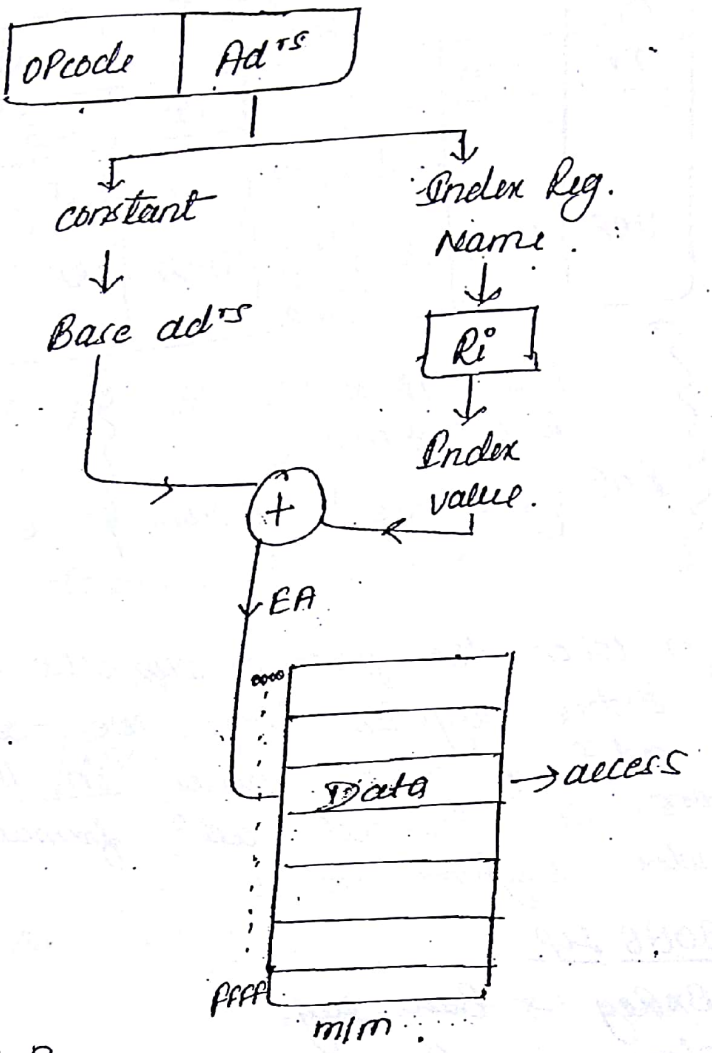
  Addrs field value → Base addrs

  [Ri] → Index value

- Data = [EA]
  = [Addrs field value + [Ri]]

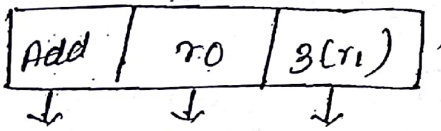
Actions:

Instⁿ design



Ex

4 word Instⁿ



r1 is a index reg.

: IR

IO

ALU O/Pⁿ ↓ +

S1 & D ↓ Reg AM

S2 ↓ Indexed AM

⇓ - - - → PD

r0 ← r0 + M[3+[r1]]

IRR (write) → WB.

IRR (read)

IRR / Arithmetic / IMR (read) → OF

PC	EC				
IP	ID	OP		PD	WB
		S_1	S_2		D
IMR	3MR	IRR	IRR IALU IMR	IALU	IRR

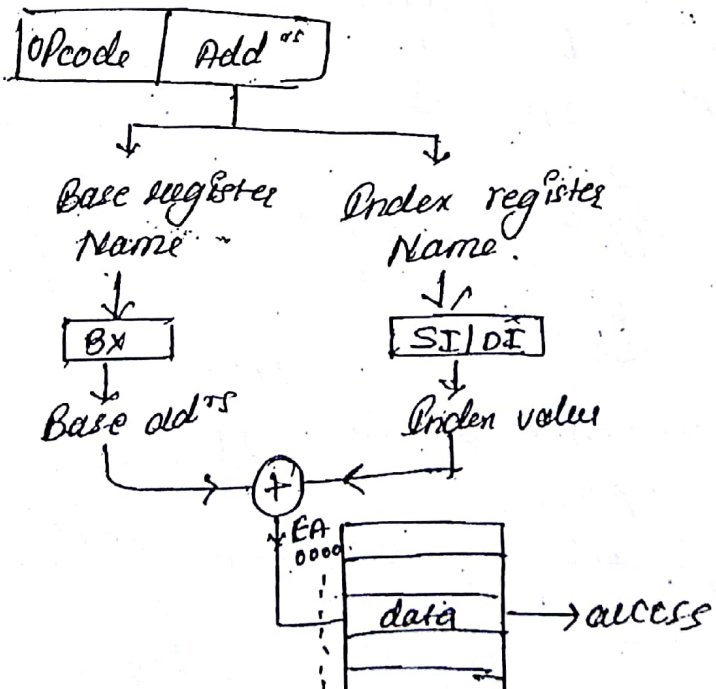
PC : IMR
EC : 4MR
Ins^n cycle (IC) : 5MR

Note → when the processor supports both base and Index registers then we can load the base add^rs & Index values in the respective registers. so Instruct^n add^rs format-contain base & Index register names.

Ex 8086 µP

BX Reg → Base reg.
SI/DI Reg → Index reg.

Ins^n design :



$$EA = [\text{Base reg}] + [\text{Index reg}]$$

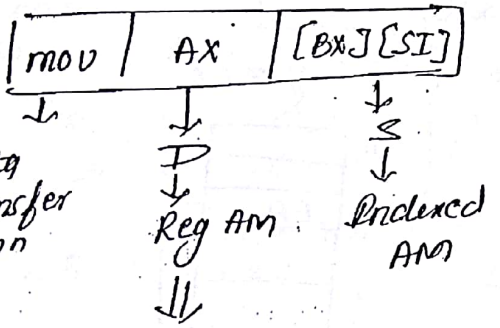
$$Data = [EA]$$

$$= [[\text{Base reg}] + [\text{index reg}]]$$

Actions

IRR → To get the base adrs
IRR → " " " Index "
IArithmetic → To cal. the EA.
IMR → To access the data.

Ex 8086 µP



Note → ① Indirect Indexed AM

In this mode effective adrs is in the m/m, the corresponding m/m adrs is calculated by adding the Index value to a base adr i.e

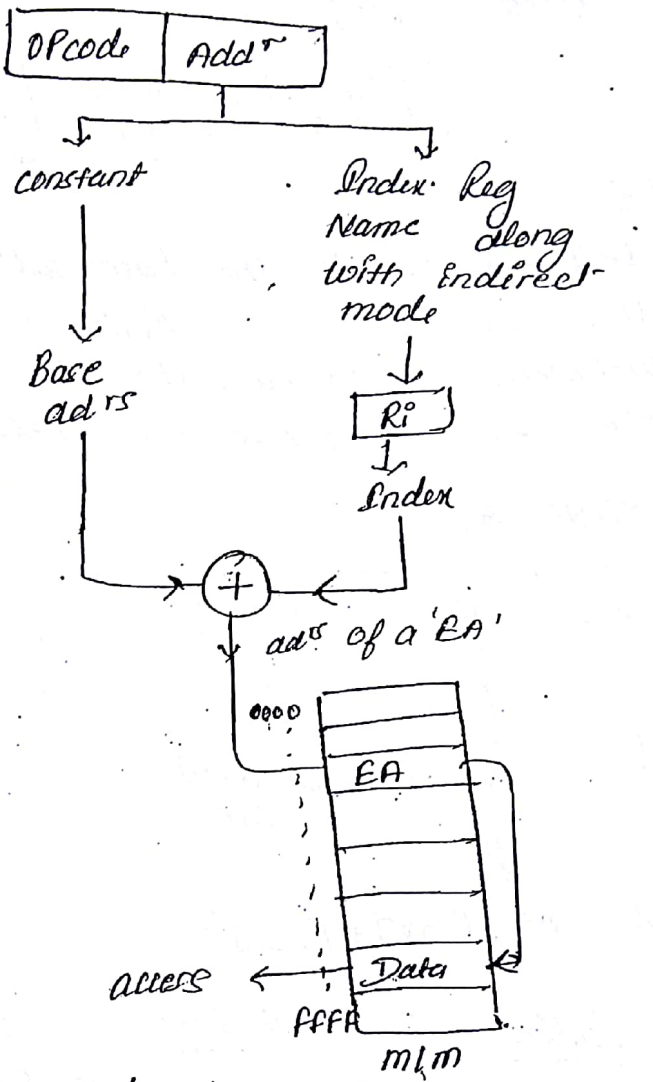
- $EA = [\text{Adrs field} + [Ri]]$
- $Data = [EA]$
  $= [[\text{adrs field value} + [Ri]]]$

Actions

IRR → to get the Index

IMR → To get the EA.
IMR → to access the data.

Instr^{n} Design



This mode take longest time to access the data it is not in the practice...

2.6 Auto Indexed Am.

- This mode is used to access the linear array element from the m/m. Arr
- In this style of accessing only one parameter is required.

Base Adrs

↓

Starting/ending adrs of an array.

↓

Present in the Base register (R_b).

- In this mode EA is calculated either by incrementing or decrementid the content of a base register with a step size.
- Step size is depends on the word length of a CPU. so it is a fixed constant.

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

$$\text{Data} = [EA]$$

$$= [[\text{Base reg}] (+/-) \text{Step size}]$$

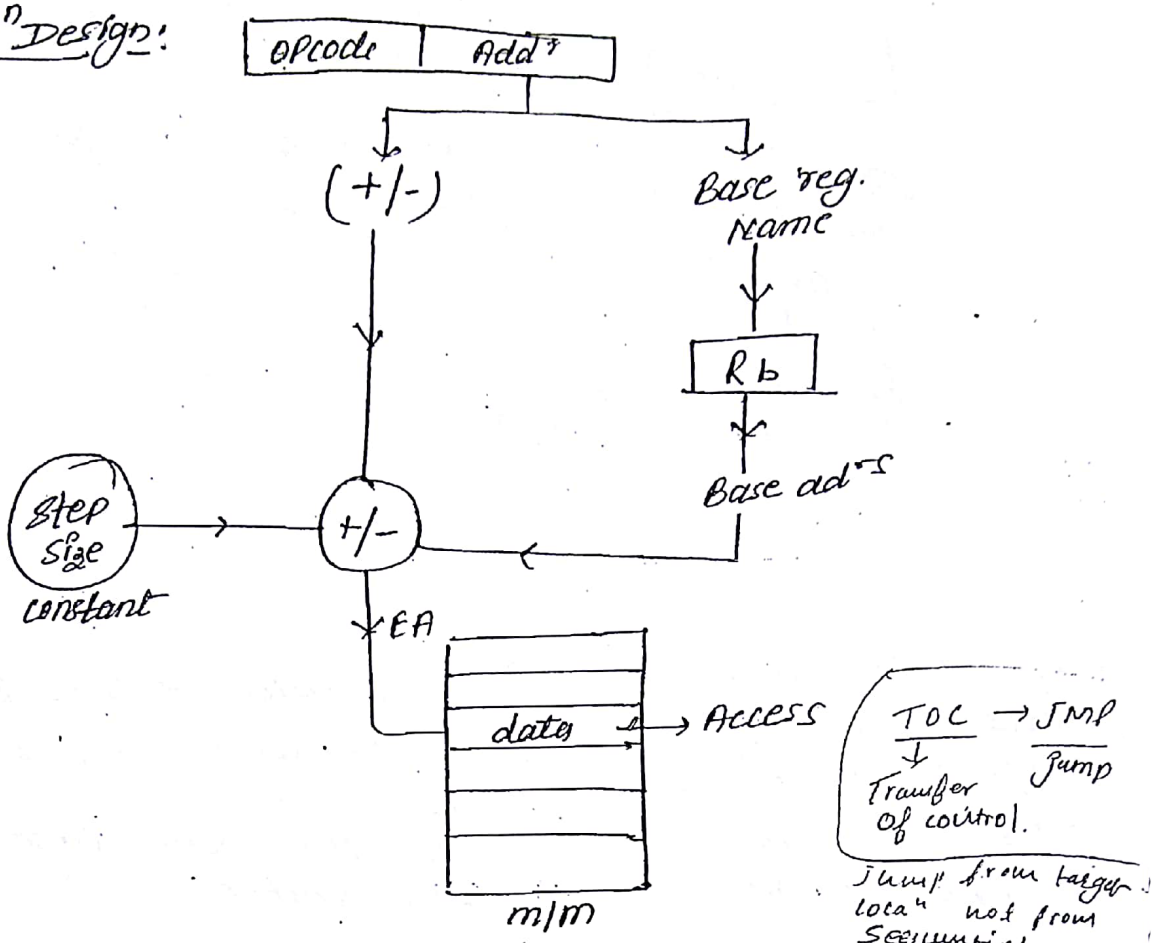
Actions

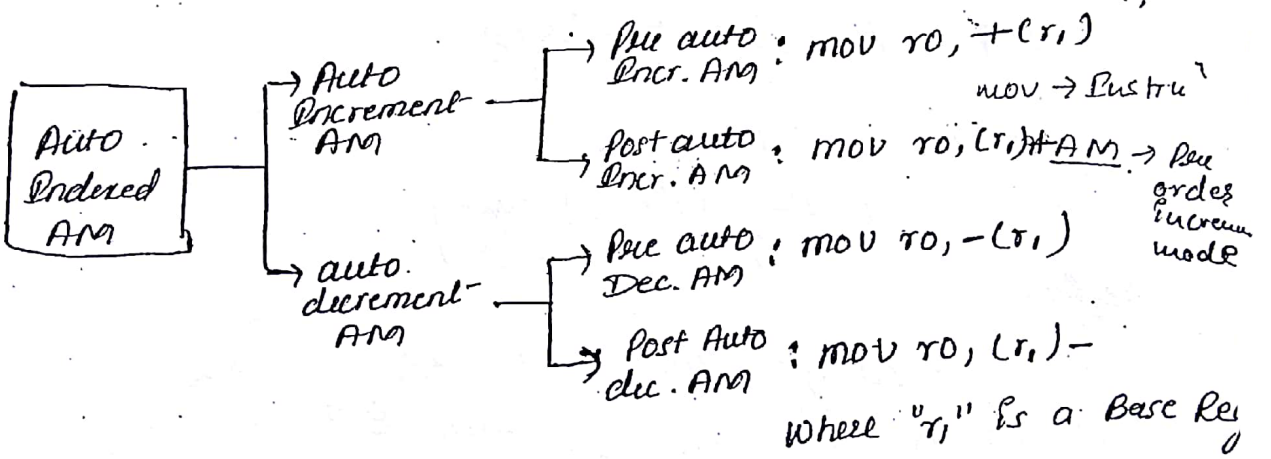
IRR → To get the Base adrs

Arithmetic → To cal. the EA

IMR → to access the data

Ins^{n} Design:





*) Transfer of control flow AM

- These modes are concentrates on Instruc^{n} location so EA indicates the add^{r} of a next Instruc^{n}.
- When the Prog contains control structures then (Jump) program control will

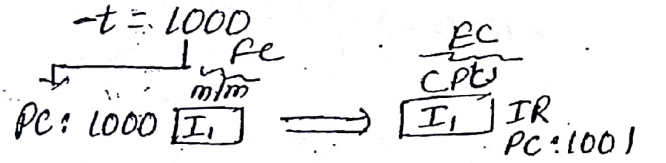
Code

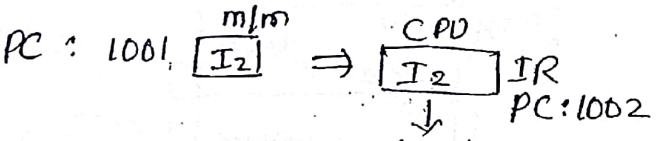
1000 : I_1
1001 : I_2 (goto l_1)
1002 : I_3
1003 : I_4
⋮ ⋮
(l_1) 2000 : Ins^n
2001 : Ins^n
⋮ ⋮

Expected o/p

$I_1 - I_2 - Ins^n - Ins^n$

Execn





↓ decode

↓

goto | l_1

↓

unconditional Jump

↓

AM's used to cal the 'EA'

↓

EA = 2000

↓

Pc : ~~1002~~ 2000

Actual o/p

$I_1 - I_2 - Ins^n - Ins^n$

To implement the control structure in the base h/w special instruct is used name as transfer of control instruct (jump / Branch / skip)

diff memonices jump used ins

transfer of control instruct is design with a AM to calculate the target adrs.

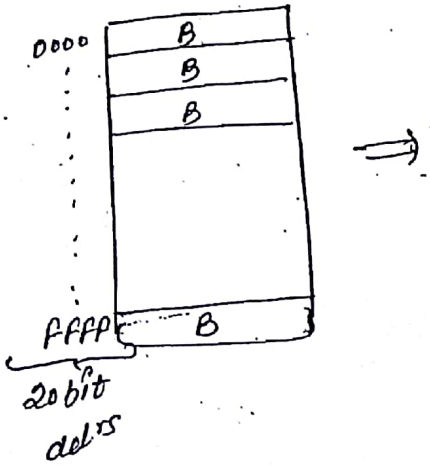
① Relative / PC-relative AM

② Based / Base reg AM.

These addressing modes depends on the m/m organisa let us consider ~~80~~ 8086 m/w design to analys the ~~aw~~ above AM i.e 8086 supports 1MB m/m , Organised into a 16-logical segments with each segment size, of 64KB.

Before org -

1MB → 2^{20} cells

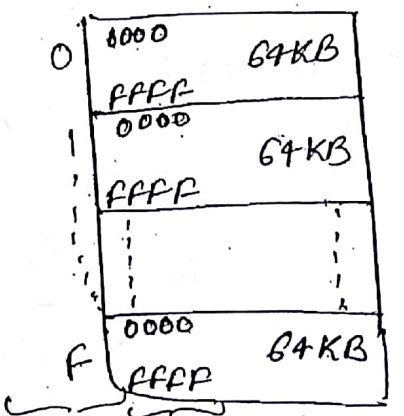


After organ

• 1MB ⇒ 16 X 64KB

2^4 2^6 2^{10}

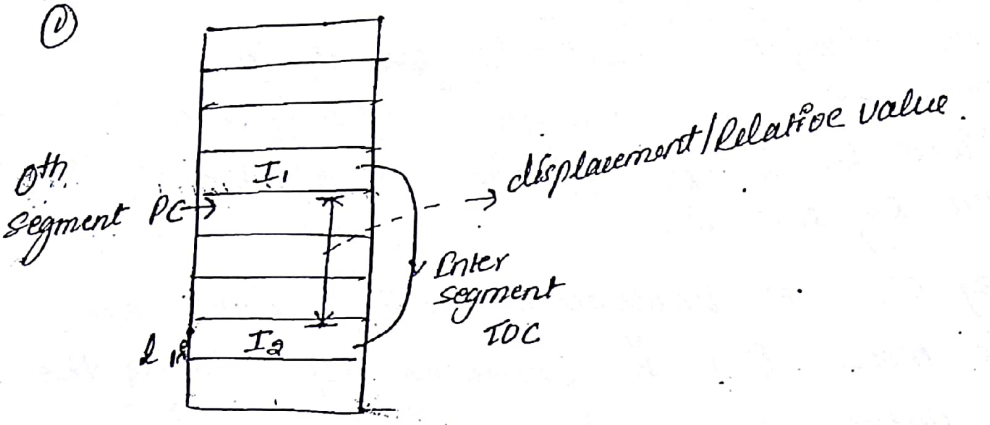
⇒ 2^{20} cells



Base adrs — offset → internal adrs is known as offset

20 bit adrs

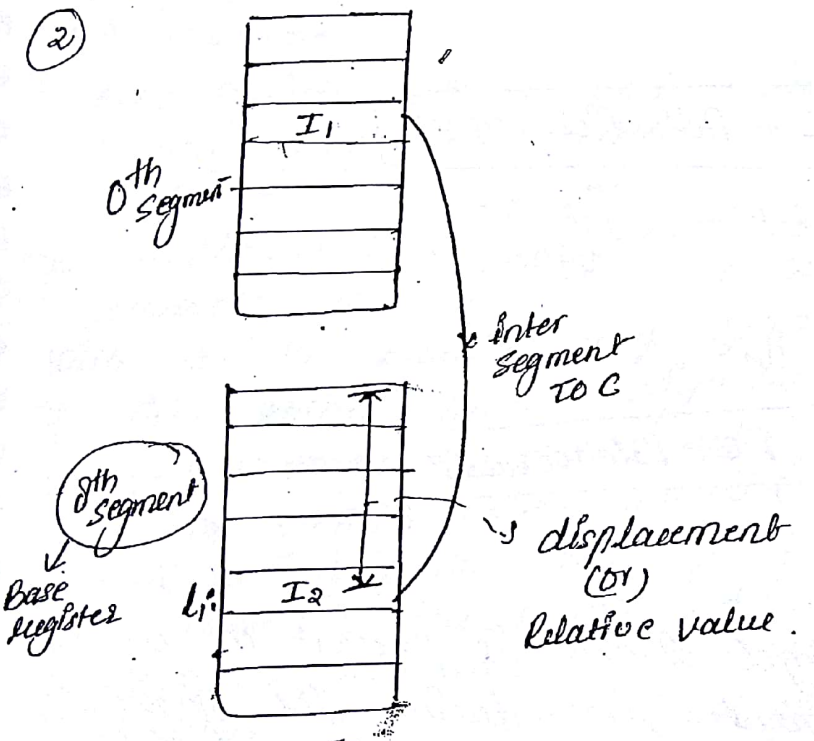
(1)



I_1 : Jmp L_1

{ EA = PC + Relative value. }

(2)



I_1 : call L_1

{ EA = [Base Reg] + Relative value]

7) Relative addressing mode.

- When the target Instruc is present in the same segment then during the prog. execu control will be transfer within the segment called as Intra segment Toc.
- Above o/pn is implemented using Relative Am.
- In this mode EA is obtained by adding the Relative value to a Rc.
- Relative value means distance b/w the current locan to target locan. It is a signed constant present in the adrs field of the Instrucn

$$EA = PC + \text{Adrs field value}$$

- PC → next Instrun adrs
- current locan
- Relative value.

$$PC \leftarrow PC + \text{Relative value}$$

- When the target-Instrun is present in a different segment the during the progam execu control will be transferred b/w the segments called as Inter segment Toc.
- Above o/pn is implemented using Base register Am. ~~In this~~
- In this mode EA is op obtained by adding

$$EA = [\text{Base reg}] + \text{adrs field value}$$

- Next Instn adrs
- target segment adrs
- Relative value.

$$PC \leftarrow [\text{Base reg}] + \text{Relative value.}$$

Note → ① PC relative & Base register both of the modes are used for program reloca^{n} at runtime.

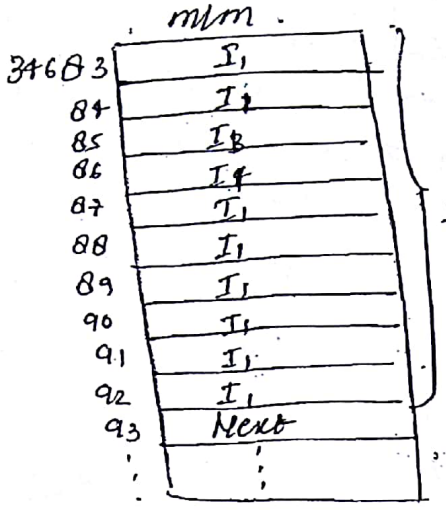
② Base register AM is best suite to write the position independent codes.
(reusable codes)

Q① consider a 16 bit hypothetical processor w/c supports 10 byte Instru^{n}. Instruc^{n} is design with a PC relative jump opcode. Adrs field of the Instru^{n} contain sign constant as -32. Instruc^{n} is stored in the m/m with a starting adrs of 34683 decimal onwards. ⓐ what is the branch adrs ⓑ If the base register contain 43684 then what is the branch adrs, when the instruc^{n} is designed with base register AM.
② If index register contain 28 what is the branch adrs when the instruc^{n} is designed with a base Indexed AM.

Ans: -32
34683
ⓐ Branch adrs

Next Page

Instⁿ Size = 10B



FC → IMR

CPU

Opcode. | IR : PC : 34685 } FC

decode

shift is a 5 words Instⁿ

4 MR Required ; PC ; 34693

JMP | -32

uncond JOC

Relative value.

PC - Relative AM

EA = PC + Relative value
= 34693 + (-3)
= 34661

PC : ~~34693~~ 34661

FC

JMP | -32

unconditional JOC

Relative value.

Base reg AM

EA = 43684 + (-32)
= 43652

PC : ~~34693~~ 43652

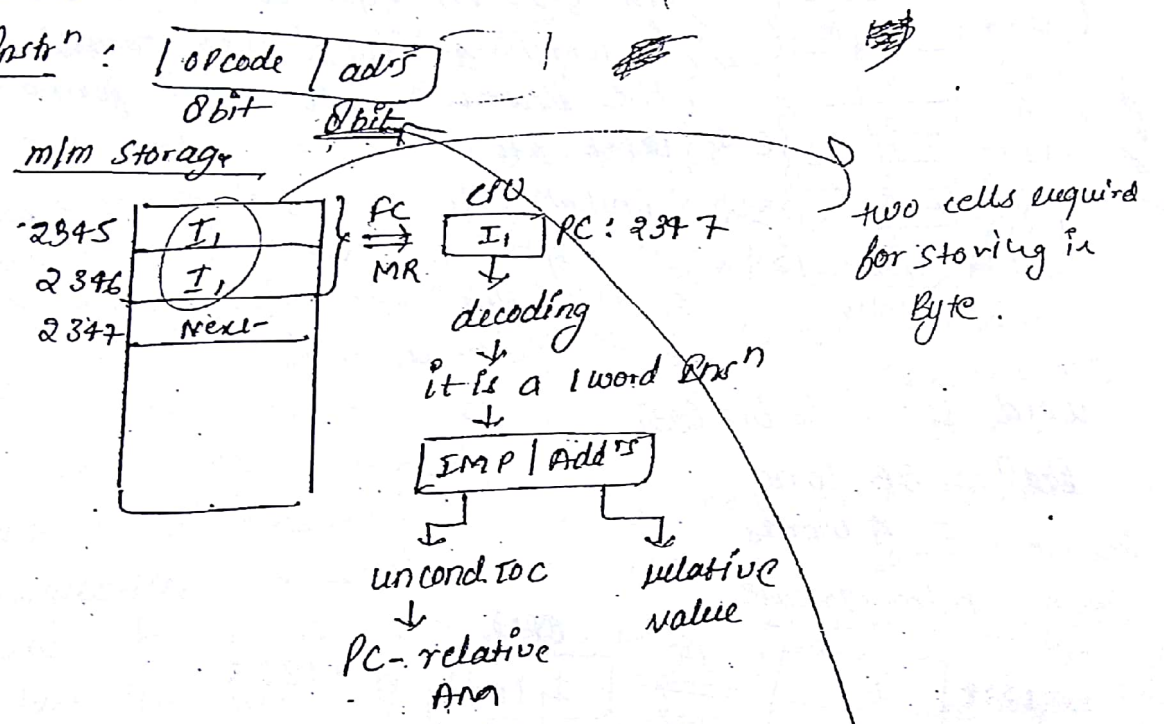
JMP | -32

uncond. JOC

Relative value.

PC : ~~34693~~

Ques 2017
Q. Consider a 16 bit hypothetical processor w/c supports 8 bit opcode & 8 bit adrs in formatted add. Instru is design with a PC relative Jump opcode. Instru is stored in the mem with starting adr of 2345 decimal onwards. During its execution control will be transfer to a 2321 locan. what is the Relative value in Hexadecimal format?



- EA = PC + relative value.
- 2321 = 2347 + relative value

∴ relative value = 2321 - 2347

relative value passed in 8 bit add field here.

= -26

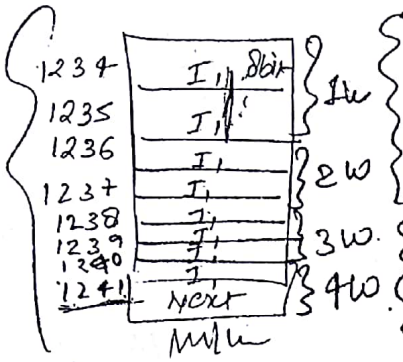
-ve data that's why 2's complement needed.

2's complement format of 8bit

26: 00011010

*complement

Q Consider a 8 Byte long jump Instruc^{n} is stored in the word addressable m/m with a word size of 16 bit. It is stored in the m/m with a starting add^{rs} of 1234 decimal onwards. During its execution control will be transfer to 1246 add^{rs}. What is the relative value in ~~octal~~ decimal format when the add^{rs} field size is ~~12 B~~ 12-bit.

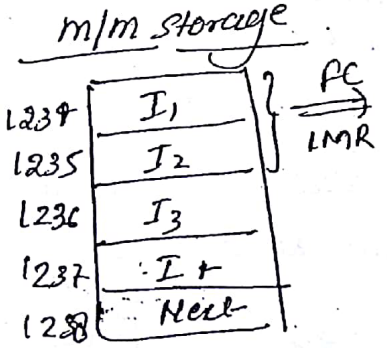


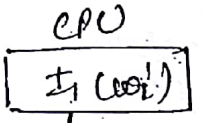
Instruc^{n} size in Byte format & M/m is word add^{rs}able then convert the Instruc^{n} into word format later store it in m/m

Instruc^{n} is in word format & mem is Byte addressable then convert the Instruc^{n} into Byte format later store it in mem.

Word size = 16 bit (2B)

Instruc^{n} = 8B long
= 4 words





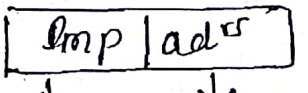
PC : 1235

↓ decoding

↓ it is a 4 word Instruc^{n}

↓ 3MR required : PC = 1238

↓



↓ uncond^{l} to C

↓ relative value (12bit)

↓ relative

EA = PC + relative value

1246 = 1238 + r

r v = 1246 − 1238
= 8

000 000 001 000
0 0 1 0

(0010)$_8$

Q A hypo. processor with diff. operand accessing mode name as Immediate, Register, direct, w/m Indirect & Index addressing mode. In the prog. 20 Instru^n are register based 40 Ins^n are direct based, 60 Ins^n are Indirect based, 30 Ins^n are Indexed, 10 Ins^n are Immediate. Processor is operating with a 500 MHz clock frequency. what is the average operand accessing type in the program code. ~~what is the average~~ when the w/m reference consumes 6 cycle, ALU o/p consumes 3 cycle zero cycle consume to refer the register & access the data directly from the Instru^n

	Ins^n	AM	cycles
I.	20	Reg	0
II.	40	direct	1 MR (6 cycles)
III	60	indirect	2 MR (12 cycles)
IV	30	Indexed	1 RR → 0 1 ALU → 3 1 MR → 6 } 9 cycles
~~V~~	~~10~~	~~Immediate~~	
V	10	Immediate	0.

~~# cycles/Prog~~ → [(20*0) + (40*6) + (60*12) + (30*9) + (10*10)]

= 1230 cycles.

cycle time ∝ $\frac{1}{\text{clock freq}}$.

cycle time = $\frac{1}{500\,MHz}$ sec

= $\frac{1}{500} \times 10^{-6}$ sec

~~= 2ns~~ = 0.002×10^{-6} sec

= 2×10^{-9} sec

= 2 ns

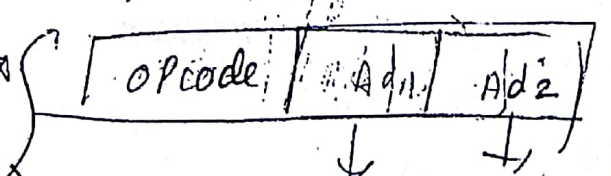
∴ (OF) time/prog = 1230 * 2ns
operand fetch = 2460 ns

Avg. of time. ⇒ $\frac{2460}{160}$

⇒ 15.37 ns.

Q consider a hypothetical Processor w/c is operating with a 2.8 usec clock. Ins^{n} is designed with 4w long. it contain opcode along with 2 operand operand 1 uses Indirect add^{n}ing & operand 2 uses direct add^{n}ing. when the opcode is ALU opcode then result will be placed into a operand 1 locati assume that m/m reference consumes 4 cycles & ALU o/p^{n} consumes 2 cycle. calculate the time required to complete the Instru^{n} when the opcode is ADD.

Ans



4 word Instⁿ

FC	EC				
IF	ID	OF		PO	WB
		S_1	S_2		D
1 MR	3 MR	2MR	1MR	1 ALU	2MR

4 word Long:

OOD	OP1	OP2
	↓ S_1 & D.	↓ S_2
+	↓ Indirect	↓ direct

Time required to complete the Instⁿ $= \{(9MR * 4C) + (1ALU * 2C)\} 2.3ns.$

$= 87.4ns$

Q. How consider a 16 bit hypothetical processor w/c supports 8 bit opcode & 8 bit bit addrs formatted Instruⁿ. Instruⁿ is store in the m/m with a starting addrs of 8888 decimal onwards processor contains 1 Register name as R0 w/c holds 186 value. Addrs field of the Instⁿ contain 200. w/ch content of 200 is 156. cal. the effective addrs. when the instruction is designed with (a) Immediate mode.

(b) Reg mode
(c) direct mode.
(d) reg. indirect mode
(e) m/m. indirect mode
(f) Indexed mode with R0 as a Index Reg.
(g) Pre auto Increment mode with R0 as a Base Reg.
(h) Relative mode.
(i) Base Reg mod. with "R0" as a Base Reg.

Instruction set (Types of Instruction or Type of operation)

Processor contains the 3 category of the Instruⁿ name as: ① data transfer Instruⁿ [mov, load, store, push, pop, In, out etc]

② data manipulation:

(2.1) Arithmetic Instruⁿ [ADD, SUB, MUL, DIV, INC, DEC, etc] ↳INR ↳DCR

(2.2) Logical Instruⁿ [AND, OR, XOR, CMP, etc]

(2.3) Shift & Rotate Instruⁿ

(2.1) Inc, Dec used for counting so it has a limit once the limits are satisfied, counts are automatically roll back

Inc → Invoke → adⁿ
Dec → " → subtraⁿ → carry flag not affected / carry flag affected

Eg: one bottle is full make it empty & again fill can't consider overflow water can't measure.

Verify chapter Note 1 → ① During the execution of increment Instrⁿ value & 1 is added to a register or mem content. when the register satisfies the upper limit then value will be roll back to zero w/o change the carry flag.

Instⁿ | Inc | r0 | → r0 ← r0+1 carry out of MSB.

Ex r0: 11111111 + 1 (Add ⁿ)

r0: 00000000

② During the execution of decrement instruction value 1 is subtracted from the register or m/m content. When it satisfies the minimum limit. value will be roll back to upper limit.

Instⁿ : | Dec | r0 |

↓

r0 ← r0 − 1

Base 23 10 29

Ex

r0 : 00000000

carry 2 2 2 2 2 2 2 2

Subtraction

Remainder 1 1

r0 : 11111111

↳ remainder 1

out of MSB carry flag is there.

Base of the NO. 2|5 is Borrow

25 Base 10

− 19

borrow into MSB

- CMP

This instruⁿ is used to compare the register contents it invokes the subtraction o/pⁿ during the execution after the execution result will be available in the zero & carry flags of a PSW.

Ex | cmp | r0 | r1 | r0 : 80H r1 : 80H

↓

SUB

↓

(r0 − r1)

↓

r0 : 1000 0000

1000 0000

0000 0000

Z = 1 & CY = 0

{ operand 2 = operand 1 }

Ex | cmp | r0 | r1 | r0 = 80 H r1 = 40 H

↓

SUB

↓

(r0 − r1)

↓ 2

r0 : 1000 0000

0100 0000

0100 0000

Z = 0 & CY = 0

{ operand 2 < operand 1 }

Ex: cmp r0, r1 ; r0 = 40H, r1 = 80H

↓ SUB

↓ (r0 - r1)

↓

r0 : 0100 0000

1000 0000

1100 0000

Z = 0 & CY = 1

∴ {operand 2 > operand 1}

(2.3) Shift & Rotate.

Shift o/p^n is used to move the data bit by bit from right to left or left to right direc^n with loss of data.

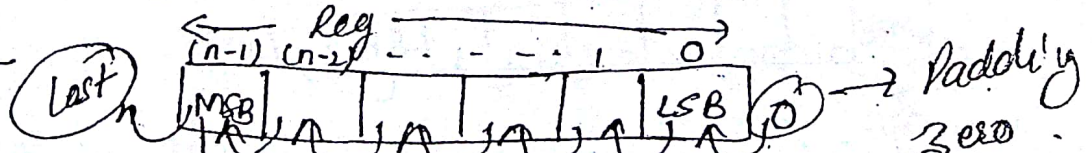
Rotate: circular shift, LSB connected to MSB.

Classifica^n of a shift o/p^n is two types.

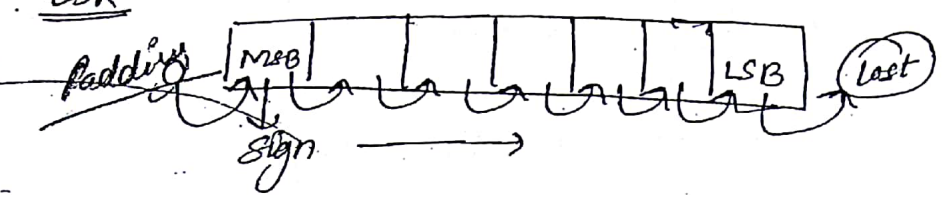
- Shift o/p^n
  - Logical
    - left (LSL)
    - right (LSR)
  - Arithmetic
    - left (ASL)
    - right (ASR)

• LSL: In the logical shift o/p^n data bits are moving to left or right direc^n including the sign bit.

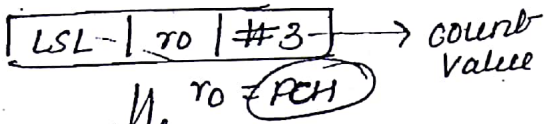
LSL

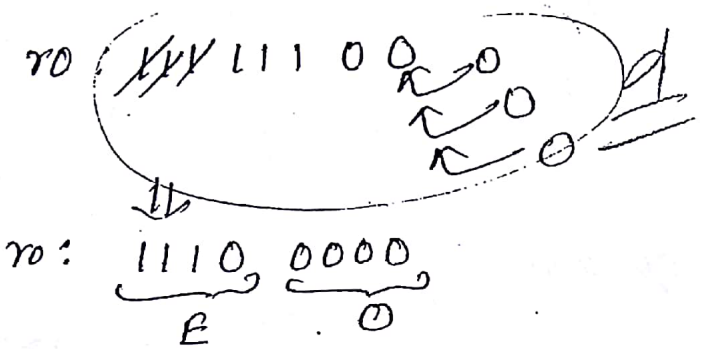


LSR



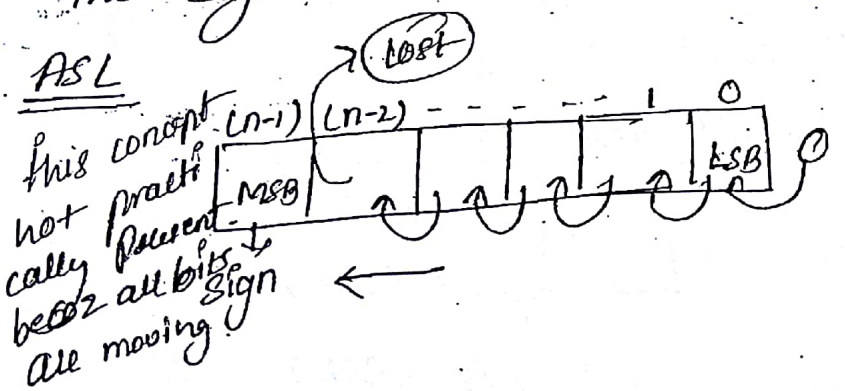
Ex





In the arithmetic shift o/p^n data bits are moving to left or right direc^n except the sign bit.

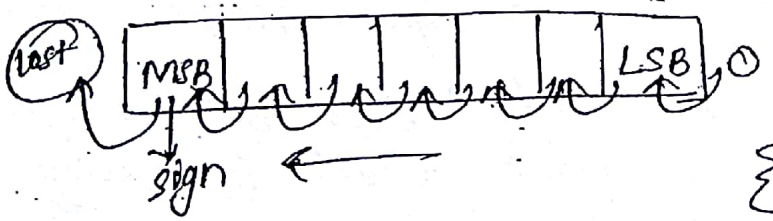
ASL



this concept not practically possible because all bits are moving

concept wise.

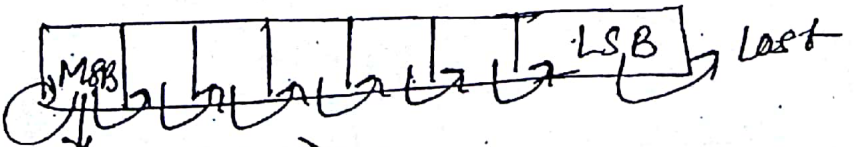
if u-2 change to u-1 sign will change so (u-2) will be lost



Implementation wise.

{ ASL = LSL }

ASR



ASR	r0	#3

⇓ r0 = (FCH) data in Hexadecimal format.

r0 : 1 1 1 1 1 1 0 0

⟶ (ASR)

⇓

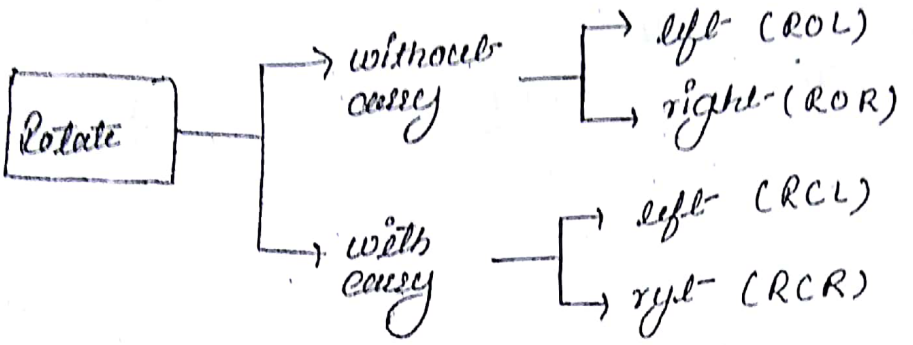
r0 : 1111 1111

F F

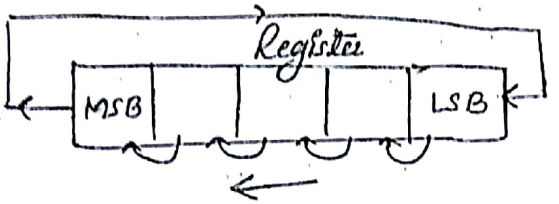
19/8/2017
Saturday
Class - 6

★ Rotate Instruction

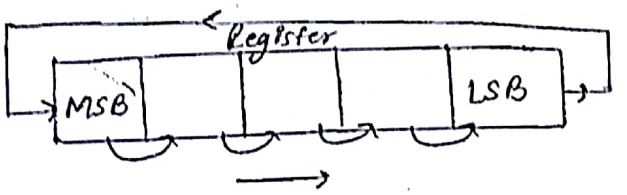
while by execution of this instruction data bits are moving to left or right direction w/o loss.



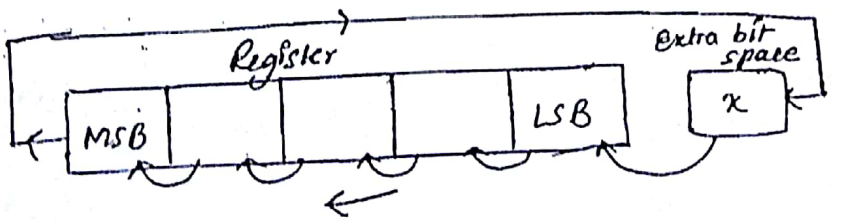
ROL :



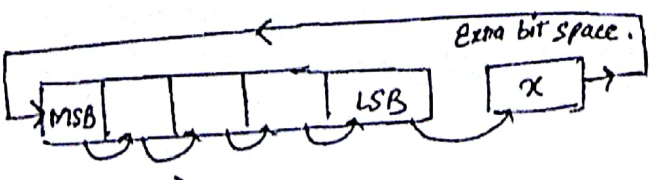
ROR :



RCL:



RCR:

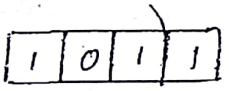


- RCL Instn : 4 times

{Reg size = 4 bits}

1st time : carry set-
2nd time : Reset-
3rd " : set
4th " : set-

data in the reg.



Reg size = 4 bit

Reg-A				Extra
A_3	A_2	A_1	A_0	x

: initial

RCR Instn

1st:	x	A_3	A_2	A_1	A_0	: carry set/Reset-
2nd:	A_0	x	A_3	A_2	A_1	: "
3rd:	A_1	A_0	x	A_3	A_2	: "
4th:	A_2	A_1	A_0	x	A_3	: "

To get the initial status in the one more register: one more RCR required to rotate ry + along with carry.

A_3	A_2	A_1	A_0	x

① data → Reg.

② Reg/2

③ If remainder = 0 then data in the Reg is even else odd.

1 - 001
2 - 010
3 - 011
4 - 100
5 - 101
6 - 110

we can apply RCR & Read carry.

(3) Transfer of control Instructions

(3.1) unconditional TOC

(3.2) conditional TOC

(1) unconditional TOC

While execution of this instruction, Program control will be ~~lacy~~ transfer to a target local w/o checking any condition.

Ex JMP 2000

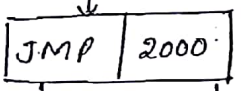
Insⁿ JMP 2000 : mem

↓

JMP 2000 : CPU

PC : seg. add's } FC

↓



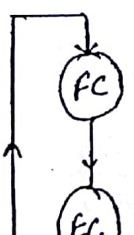
: Insⁿ format

↓ ↓

unconditional TOC → direct AM

EA → Target addᴿˢ → sequential addᴿˢ 2000

- This Insⁿ is used to implement the unconditional selection statement (go to) & the m/c control Instrⁿ (halt).
- Halt → (stop the execuⁿ of prog.)



- Halt is a m/c control Insⁿ, invokes the unconditional jump Instruction with starting addᴿˢ of a halt as a target addᴿˢ, during its execution. i.e

same location again & again w/o change the data. ∴ user point of view program execution is completed when the halt is executed, but CPU point of view halt is executed ∞ no. of times.

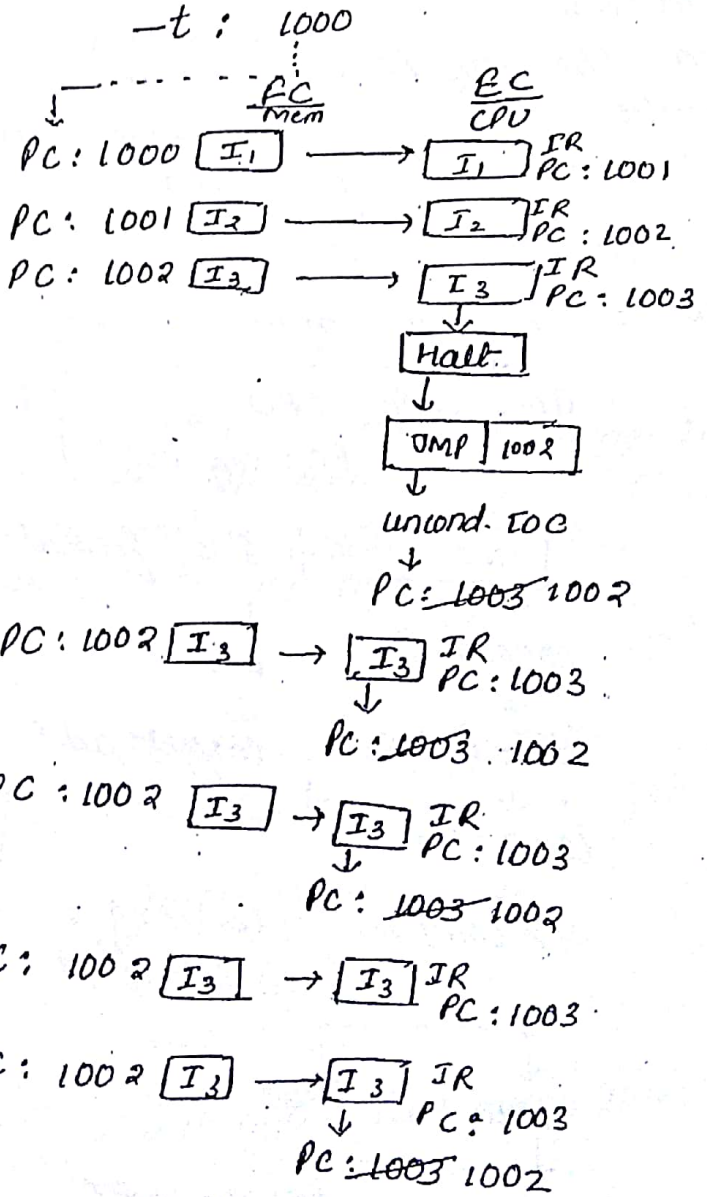
- when the CPU enter into a halt state then reset operation is required to clear the PC value, to execute the new program.

Code:

I inform to CPU plz stop at I_3

1000 : I_1
1001 : I_2
1002 : I_3 (Halt)
1003 : I_4
1004 : I_5
⋮ ⋮

execution stmt



Old sequence :

→ Petrol pump use the halt of

I_1 : MOV r_0, # user i/p r_0 : 100 by using this Instru

I_2 : mov r_1, # 00 r_1 : 00

(4) I_3 : start the gun : cenit of petrol flow unit

I_4 : INC r_1 $r_1 = 1$ $r_1 = 2$

I_5 : CMP r_0, r_1 100 cmp 1 (NZ), 100 comp 100 Z.

I_6 : JNZ l_1 NZ(F) NZ(T) PC : l_1

I_7 : Halt.

(3.2) conditional TOC

while execution of this instruct, associated condition is evaluated based on the status of a previous Instruction.

When the condition is true then the control will be transfer to a target location otherwise seq. Instn in the mm will be executed.

Ex JNZ 2000

Instn : JNZ 2000 : mem

↓

JNZ 2000 : CPU

PC : seq. addrs } PC

↓

| JNZ | 2000 | : Instn formal-

↓ ↓

cond. TOC EA

↓ ↓

direct AM target addrs

↓

COND : NZ

↓

NZ compare status of a zero flag.

↓ ↓

true false

↓ ↓

PC : seq. addr PC : seq. addrs

PC

code

1000 : I_1
1001 : I_2
1002 : I_3 (Dec r0); r0 = 2
1003 : I_4 (JNZ $1001)
1004 : I_5
1005 : I_6
⋮

Execution

t: 1000

FC (mem)		EC (CPU)
PC:1000 [I_1]	→	[I_1] IR, PC:1001
PC:1001 [I_2]	→	[I_2] IR, PC:1002
PC:1002 [I_3]	→	[I_3] IR, PC:1003 ↓ r0 = 1 (NZ)
PC:1003 [I_4]	→	[I_4] IR, PC:1004 ↓ NZ CMP NZ ↓ True (PC: ~~1004~~ 1001)
PC:1001 [I_2]	→	[I_2] IR, PC:1002
PC:1002 [I_3]	→	[I_3] IR, PC:1003 ↓ r0 = 0 (Z)
PC:1003 [I_4]	→	[I_4] IR, PC:1004 ↓ NZ CMP Z ↓ false (No change in PC)
PC:1004 [I_5]		
PC:1005 [I_6]		

- This instruction is used to implement the conditional selection stmts i.e., (if, ~~w~~ switch) & various iterative stmts (for, while, do while) in the base h/w. (final)

Control Structures →
- selection } already done.
- Iteration }
- Function ✓
  ↓
  re usable code.
  ↓
  write once, read many time.

Call - return
↓ Entry ↓ exit

Q Why return does not contain address.

- It is reuseable code, u can call it anywhere in the applica
- Subprogram can be call many times.
- Return Add^rs depends on calling fu^n. u cant fix it No fixed ret add^rs connected to program.

*) Subprograms

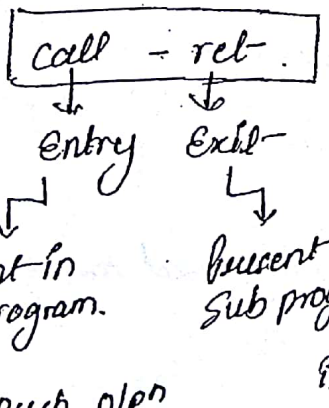
defin → Subprogram is a reuseable program, means that repeatidely occured functionality is dwelop as a subprogram only once, later refer it in the applica many times.

Characteristics

- Single entry - single exit
- main program is suspended during the execution of a subprogram.
- control will be Transfer back to main program after completion of a subprogram.

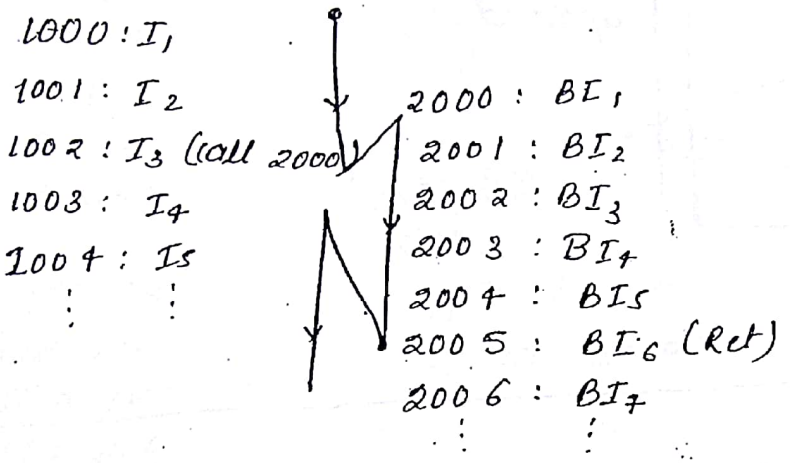
- Implementation

Pair Instruc^n is used to implement the subprogram in the base h/w.

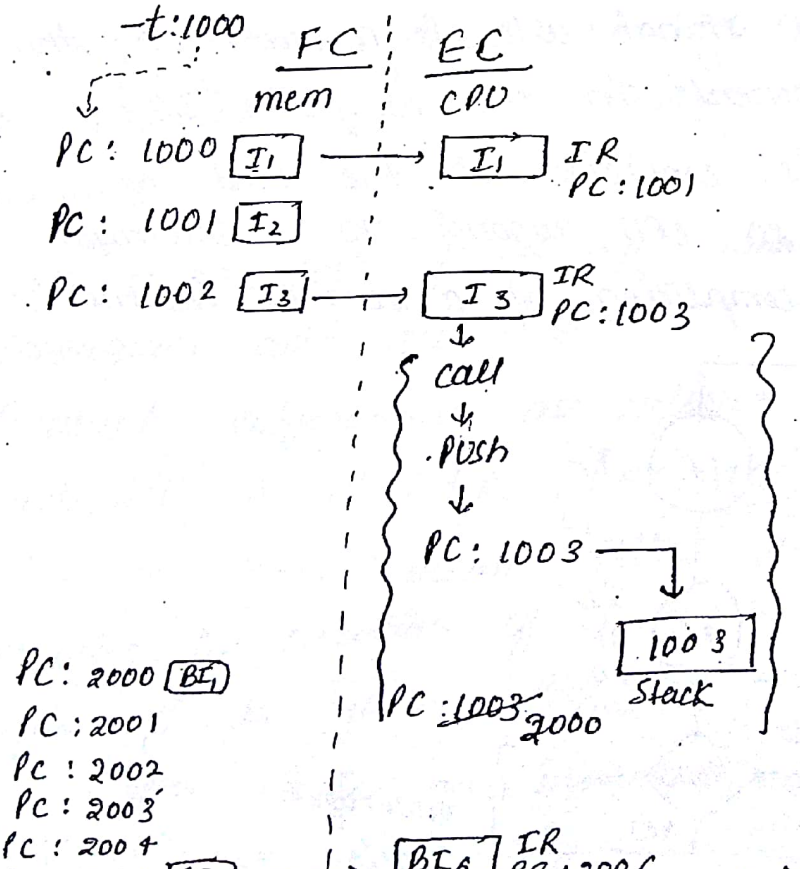


- In this implementation stack is used to store the return adrs.
- Return adrs is a next Ins^n adrs after the call Ins^n it will be varies, depend on the calling position of a subprogram.

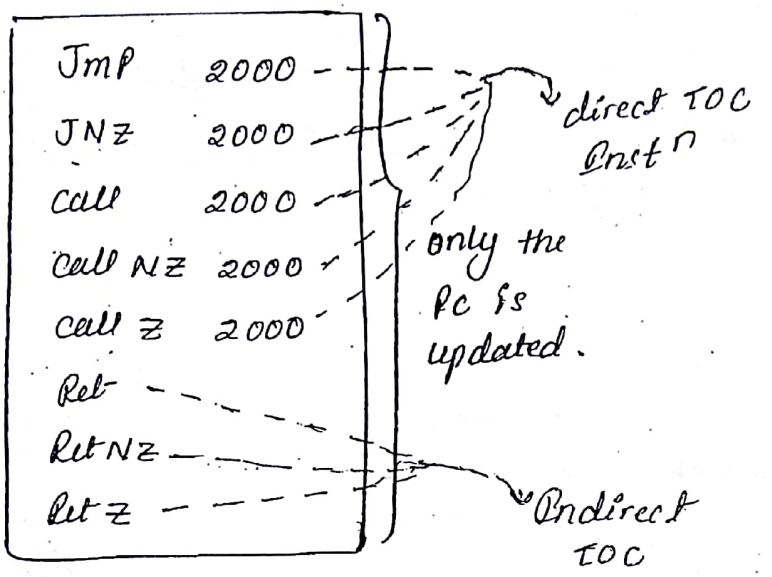
Code



B stands for nothing it is used to differ-late blw main prog. & subprog

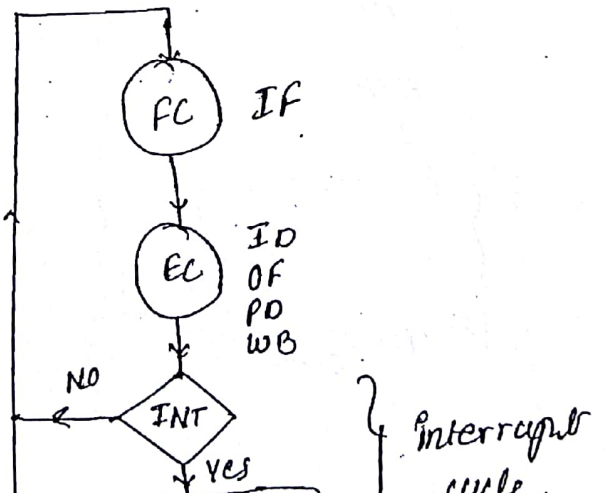


- These all are different type Instruc^n



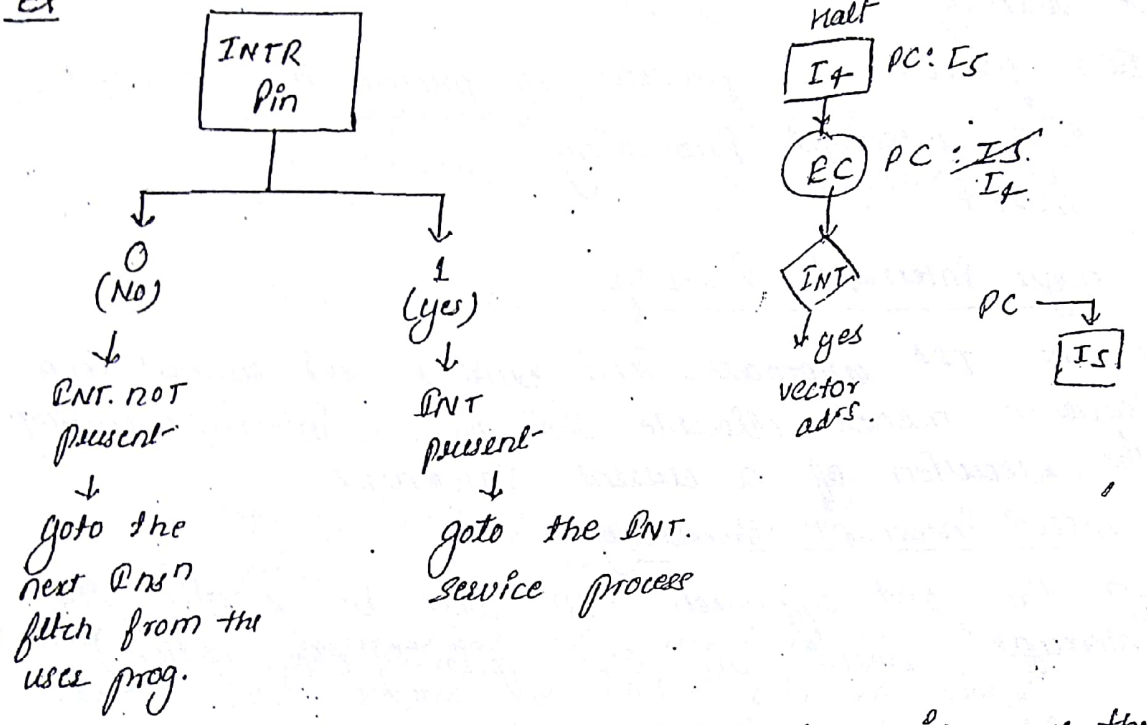
Interrupt cycle

- Interrupt is a signal w/c is generated by the low speed component in the computer (IO).
- Interrupt cycle is connected at the end of execution cycle so CPU respond to a interrupt only after the completion of a current Instruct^n execution.



- CPU reads the status of a interrupt pin after the execution cycle to detect the Interrupt.

Ex



- When the CPU detect the Interrupt then it saves the PC value into a stack later transfer the control to Interrupt vector table. (IVT)
- IVT is a part of the m/m where the Interrupt subprogram adrs are present.
- Interrupt subprograms are suffixed with a vector adrs end with a IRET / RETI / RFE Instn..
  - IRET → Interrupt return
  - RETI → return from Interrupt.
  - RFE → return from exceptn
- During the execution of IRET instrun invokes the pop opn to restore the PC with a return adrs ∴ after servic the Interrupt, control will be transfer to a user program.

Note → objective of a interrupt cycle is, interrupt subprogram initialization, in this process PC value saved into a stack, later vector addrs will be loading into a PC.

⇒ Two process are present to process the interrupt

1) single interrupt Processing
2) nested " "

1) single interrupt Processing.

In the 1st approach, CPU services the request in a sequence means disable the further interrupt during the execution of a current interrupt.

2) nested interrupt Processing:-

In the 2nd approach CPU will be services the interrupt based on the priority (Preemption).

Q) Consider a hypothetical CPU w/c supports 4 interrupts with a respective service time of

(I_1, I_2, I_3, I_4)

20ns, 40ns, 30ns, & 10ns.

response time of a interrupt is '2ns'.

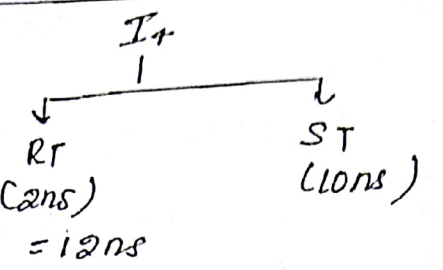
I_1 has the highest priority & I_4 has the least priority what is the range of time required to service the I_4 interrupt when the interrupt may or may not occur simultaneously.

Range → minimum to maximum

- Total time = response time + service time

delay b/w request line & ack. line is 'R.T.'

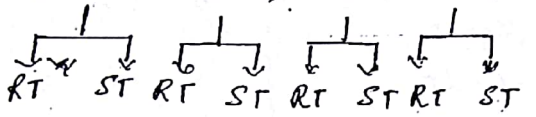
① Min time required for "I_4" : only one INT. at a time



$= 12ns$

(2) Max. time required for "I_4" : All 4 INT's present at a time {with simultaneous occurrence}

$\Rightarrow$ (I_1, I_2, I_3, I_4)



$\Rightarrow 108ns$

$\therefore$ Range of time : {12ns to 108ns}
Req. for I_4

Q2) consider a hypothetical CPU used to execute the program w.r.c contain 20 Instruⁿ (I_1 to I_{20}) Avg CPU Cycle per Instⁿ is 8. If interrupt occurs during the execuⁿ of I_8 Instruⁿ then after how many cycle CPU will be respond to a interrupt in the program execution.

Ans $8 \times 8 = 64$ cycle

Interrupt occurs during the I_8 Instⁿ
So, CPU respond to the interrupt after "I_8" Instruⁿ execution.

$\therefore$ cycles required upto I_8 Instⁿ Execuⁿ
$= 8 * 8$ cycles
$= 64$ cycles.

Q3) consider a hypothetical processor used to execute the following code, word length of the CPU is 16 bit, m/m reference consumes

In the m/m with the starting adr of $(AAA)_H$

Instn	meaning	size (in words)
move r0, @ 3000	$r_0 \leftarrow M[[3000]]$	4 8
move r1, [2000]	$r_1 \leftarrow M[2000]$	2 4
Add r0, @ r1	$r_0 \leftarrow r_0 + M[[r_1]]$	3 6
mul r0, r1	$r_0 \leftarrow r_0 * r_1$	2 4
mov 3(r0), r1	$M[3+[r_0]] \leftarrow r_1$	3 6
mov r1, 4(r0)	$r_1 \leftarrow M[4+[r_0]]$	3 6
Sub r0, r1	$r_0 \leftarrow r_0 - r_1$	2 4
Halt	mlc Halts	1 2
		40

a) If an interrupt occurs during the execution of a halt instruction what will be the return adrs pushed into a stack.

b) How many cycles are required to access the operands (Read & write) prog. exec^n)

c) How many cycles are required to complete the program.

A 10 11 12 13

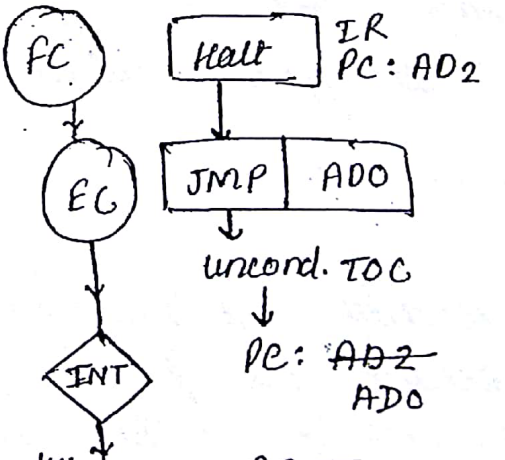
1 2

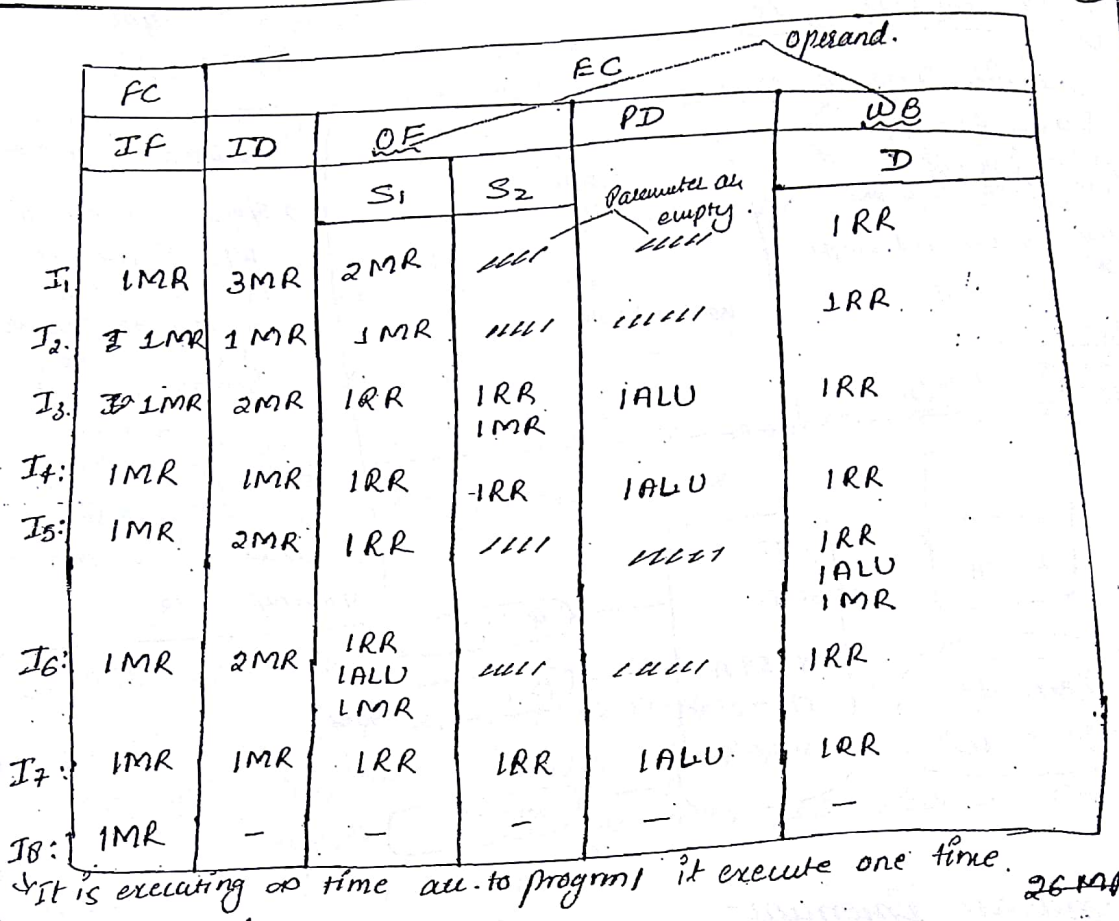
(a) • word size = 16 bit
= 2B

• m/m ⇒ Byte addressable By default

Storage

I_1 :	AAA	— AB1
I_2 :	AB2	— AB5
I_3 :	AB6	— ABB
I_4 :	ABC	— ABF
I_5 :	AC0	→ AC5
I_6 :	AC6	→ ACB
I_7 :	AC C	→ ACF





↳ It is executing ∞ time acc. to program it execute one time.

~~26 MR~~

Operand access ⇒ 6MR + 2ALU + 15RR

⇒ (6 * 6C) + (2 * 2C) + (15 * 0)

⇒ 40 cycles.

Program Execution time = [26MR + 5ALU + 15RR]

⇒ (26 * 6C) + (5 * 2C) + (15 * 0)

⇒ 166

Types of Interrupts

whenever CPU enters upt here → RST (4.5)

vector Interrupt not require ack — vector.

vector : 4.5 × word length adrs
= 4.5 × 8

16/36
2 – 4 (24)$_H$

H/W devies

High prior → nonmaskable I

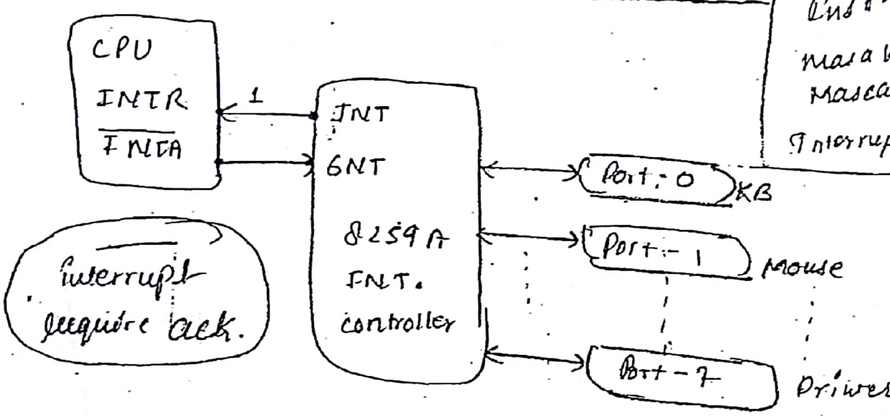
Low pr → masks

Basically 2 types.
S/w & H/w
generated signal is called H/W interrupts

→ S/w I → is a instrn which is defined
→ in the comp. design to execute I O which are present in the prog area.

In 8th pt of vector maskable & non maskable not present.

Interrup vector



1) Hardware Interrupt:-

It is a signal generated by the h/w components in the computer design h/w is interface using internal interface or external interface so h/w interrupts may be a internal interrupt or external interrupt.

External Interrupt is a signal generated by the external h/w.

Eg → power supply.
Basic IO divices
[key board, Printer, etc)

Internal Interrupt → is a ~~of~~ signal ~~gene~~ generated by the internal h/w.

Ex → Temp. sensor, timer, clock, critical sensors in the motherboard, Invalid opcode, divide-by-zero, stack overflow, etc.

On the CPU design, hlw pins are reserved to hold the hlw interrupts.

2) Software Interrupts.

It is an inst, used to execute the predefined IO fun in the processor area.

Ex. system calls.

3) Maskable Interrupts

It is a low priority interrupt, so CPU may or may not respond to this interrupt after the execution cycle. ∴ low priority IO devices are connected to this pin.

4) Nonmaskable Interrupts

It is a high priority interrupt, so CPU compulsor y respond to this interrupt after the execution cycle, ∴ high priority IO devices are connected to this ~~pin~~ pin.

5) Vectored Interrupts.

This Interrupts contain the vector so vector add^n is calculated based on the interrupt vector.

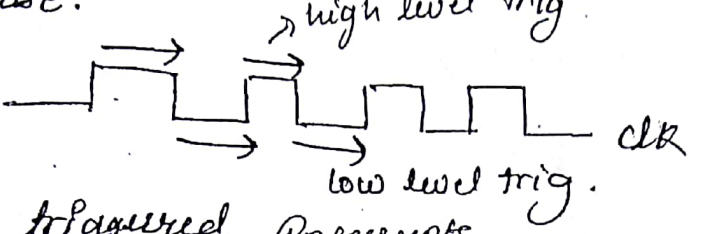
6) Non vectored Interrupts

This Interrupts doesn't contain the vector so

Ex INTR pair with $\overline{INTA}$

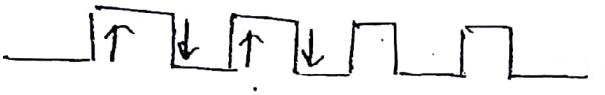
4) Level triggerred Interrupts.

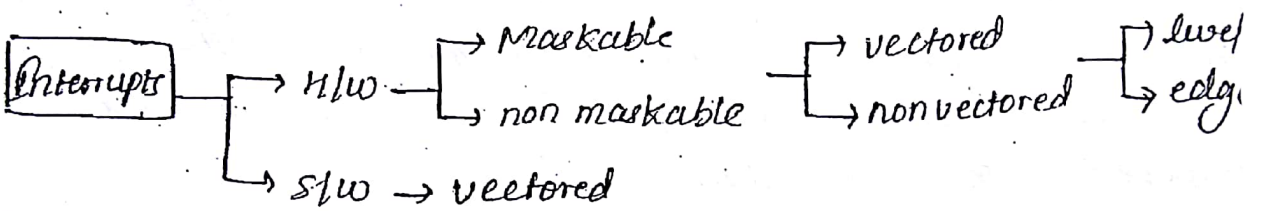
This Interrupt is based on the levels of a clock pulse.



5) Edge triggerred Interrupts.

It is enable based on the rising edge or falling edge transition of a clock pulse.





20/8/2017 (68)
sunday
class-7

RISC & (VS) CISC

★ Characteristics of RISC

① reduce Instruction set computer. It supports more register.
② it supports less addrsng mode
③ " " fixed length Ins^n
④ " " successful pipeline.
⑤ CPI = 1
⑥ It is a super computer.
⑦ It is used in the real time application.
⑧ It is a high expensive processor
⑨ It contain optimized smaller Ins^ set
⑩ It uses hard ~~wired~~ wired control unit.
⑪ Ex. motorola processor, power pc processor, ARM processor.

→ Supercomputer is RISC comp.
→ RISC comp have smaller optimize Ins^n
→ practical comp. are motorola all Personal comp. are CISC.

★ Characteristics of SE CISC (complex Instruc^n set computer)

⑥ It supports less register
② " " more addressing modes
③ " " variable length Instruc^n
④ " " unsuccessful pipeline
⑤ CPI ≠ 1
⑥ It is a general purpose computer
⑦ It is used in the personal application.
⑧ It is a less expensive processor.
⑨ It contain complex Instruc^n set.
⑩ It uses micro program control unit.
Ex. Pentium Processor.

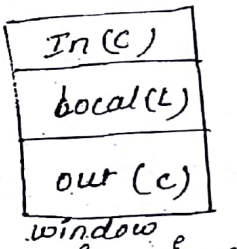
★ Register organization in Risc CPU.

- Risc supports more no. of register, categorized into 4 groups.

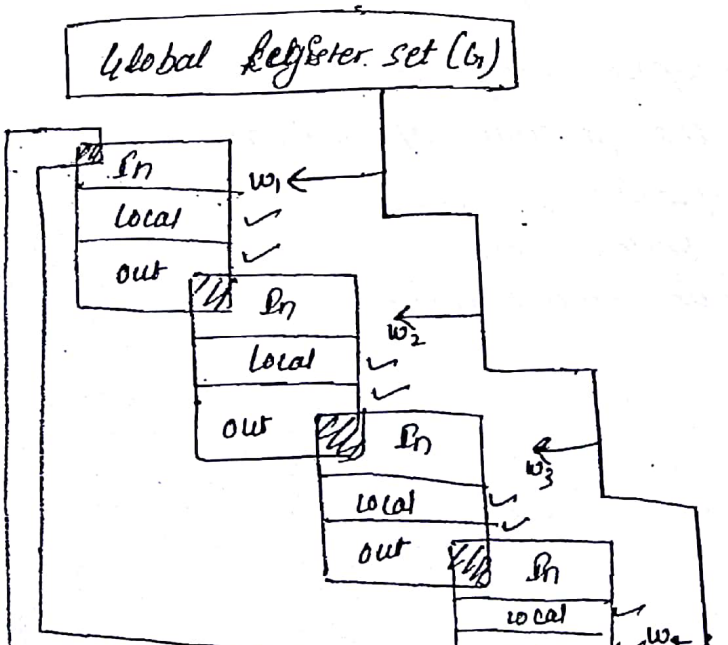
① Global register set (G)
② local Register set (L)
③ In Register set (C)
④ out Register set (C)
↳ common

- In a Risc CPU register window is formed by grouping the local, In & out register set.
- Global register are also accessible by the register window ∴ no. of register in the window is calculated as:

$$\text{Window Size} = L + 2C + G$$



- In the RISC CPU reg. windows are organize in a overlapping order i.e one window out register are used as a In register in another window.



Registers in the CPU (Reg file)

42(30)+32

$\Rightarrow \boxed{w(L+C)+G}$

w : # windows

L : # local registers

G : # Global "

C : # In/out "

42(20+10)+32

30
10+2
60
120

1262
32
1294

1262

Q. Consider a hypo. super computer w/c contain 42 windows CPU support 32 Global reg, 20 local reg, 10 In-reg & 10 out reg. What is the window size of reg. file size in the computer.

window size = L+2C+G

= 20+(2*10)+32

= 72

Reg. file size = w(L+C)+G

= 42(20+10)+32

= 1292

**** End of module 1 ****

★ Computer components

computer s/s contain 3 fundamental components.

① CPU
② m/m
③ I/O

CPU organization

CPU contains 3 internal components used to process the instructions

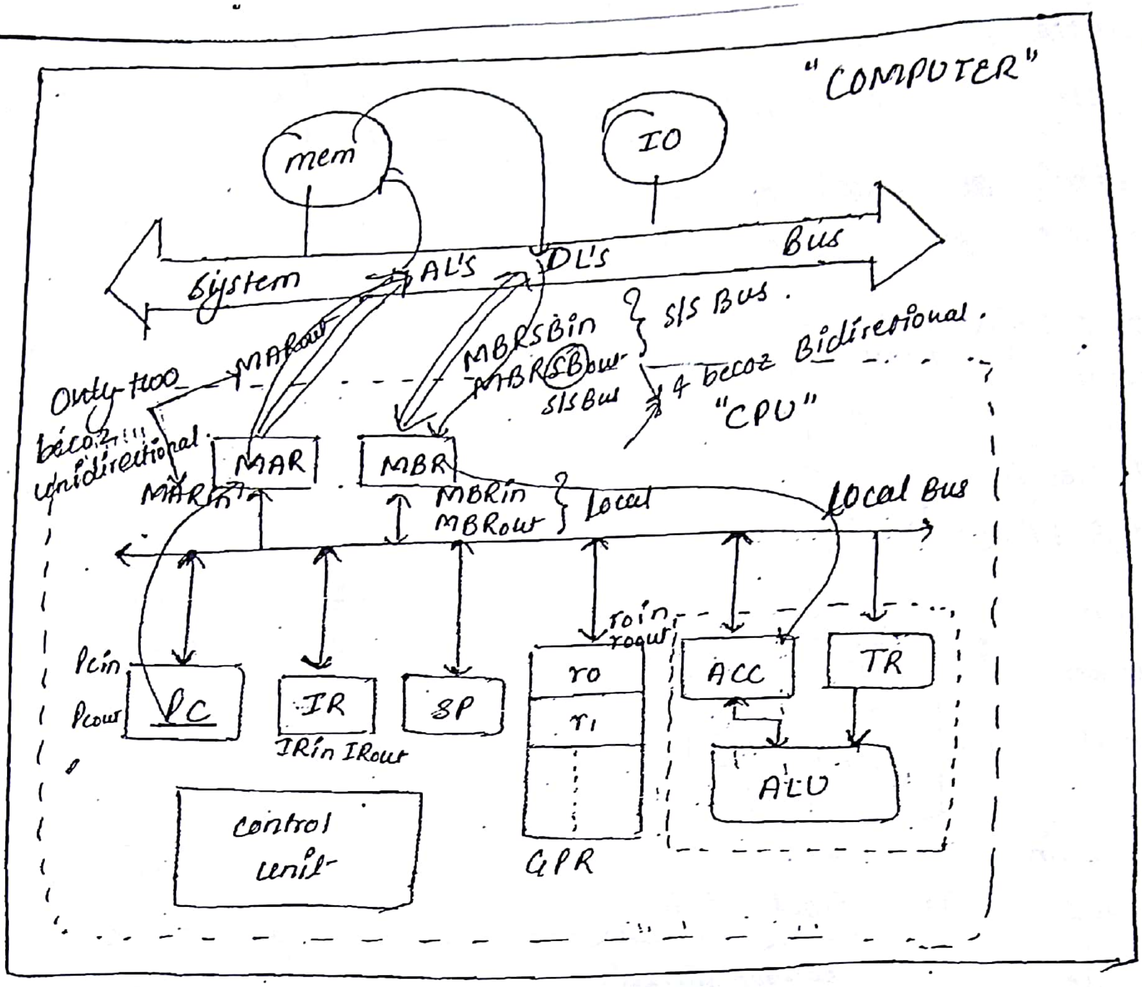
① Registers
② ~~ALU~~ already complete in module 1.
③ control unit

★ Registers

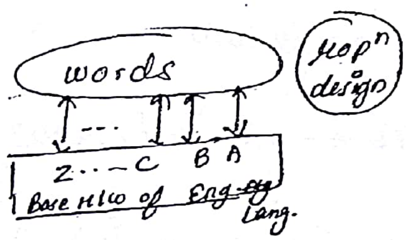
Reg. is mandatory. Every CPU must contain the CPU. to hold the status of CPU, Reg

Register is a internal storage element in the CPU, so CPU contain a set of mandatory registers.

① PC (Prog. counter): Holds the Instⁿ adᵗˢ
② IR: Holds the currently fetched opcode.
③ ACC: Holds the ALU i/p & ALU o/p.
④ TR (temporary reg): Holds the ALU 2nd operand.
⑤ MAR: Holds the m/m adᵗˢ connected to a adᵗˢ lines. ALUS of the s/s bus.
⑥ MBR/MDR: Holds the m/m content, connected to a data lines of the s/s bus.
⑦ SP: Holds the top of stack adᵗˢ
⑧ Flag reg: Holds the status of an Instruⁿ.
⑨ GPR: Holds the data.



* Micro-operation is a primitive o/p^n design in base chip.



→ designer attach μ program to every actions

① μo/p^n is a primitive o/p^n in the base h/w.

② Register to Reg. transfer o/p^n is a one kind of the μo/p^n.

③ μo/p^n takes one cycle to complete.

④ When the μo/p^n contain m/m reference then it takes more than one cycle to complete.

⑤ μ Instru^n contain one (or) more μo/p^n. All the μo/p^n in the μIns^n is running in parallel, ∴ μIns^n also takes one cycle to complete. Hence μIns^n equal to μo/p^n.

{ μ-Ins^n — 1 μop^n → exe^n — 1C
  μIns^n — 1000 μop^n → Exe^n in parallel — 1C }

(6) μIns^n or μop^n contain set of control signals, to execute the action directly on a base h/w.

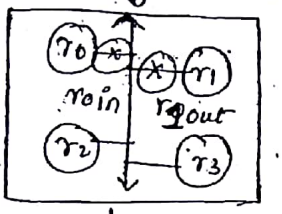
m/c Instru^n : mov r0, r1

↓

Reg. Transfer Lang. (RTL) : r0 ← r1

↓

H/w design :



↓

μop^n list : T1 ; r1out , r0in

control signal.

↓

μprogram

Add r0, [2000]

↓

r0 ← r0 + M[2000]

↓

H/w design

↓

μo/p^n list

↓

μprogram

↓

Control signals.

Mul r0, @ 2000

↓

r0 ← r0 + M[[2000]]

↓

H/w design — for instru^n fetch.

↓

μo/p^n list

↓

μprog.

↓

Control signals.

* **Micro Program** for o/p^n fetch using direct accessing mode.

- Sequence of a μo/p^n executed in the base h/w to satisfy the meaningful action is called as μprogram.

✓ μ prog. for Ins^n fetch is H/W sequence / H/W design.

H/w design : PC → MAR → AL's] [memory] → DL's → MBR → IR

(PCout → MARin; MARout … MBRSBin; MBROut → IRin)

{Instr^n add's} $\overline{RD}$ → CL's] {Ins^n}

PC + I → PC (PCout → PCin)

T₁: PCout , MARin

T₂ : MBRout , IRin

μprog

T₁ : PC → MAR ; PCout , MARin

T₂ : M[MAR] (memory) → MBR ; MARout , MBRSBin } system Bus.

PC + I → PC ; PCout , Inc , PCin } local Bus.

T₃ : MBR → IR ; MBRout , IRin

✓ of using Direct AM is :

Instr^n ; with direct mode. [opcode | Addr] IR
↓
EA

Ex. load [2000] : ACC ← M[2000]

H/w design : IR[Addr] → MAR → AL's] [Mem | Binary] → DL's → MBR → ACC

Uprog

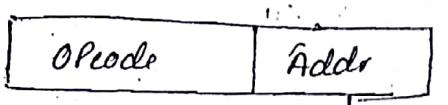
T_1 : IR [Addr] → MAR ; IR Addr$_{out}$, MARin

T_2 : M[MAR] → MBR ; MAR$_{out}$, MBRSBin

T_3 : MBR → ACC ; MBR$_{out}$, ACCin.

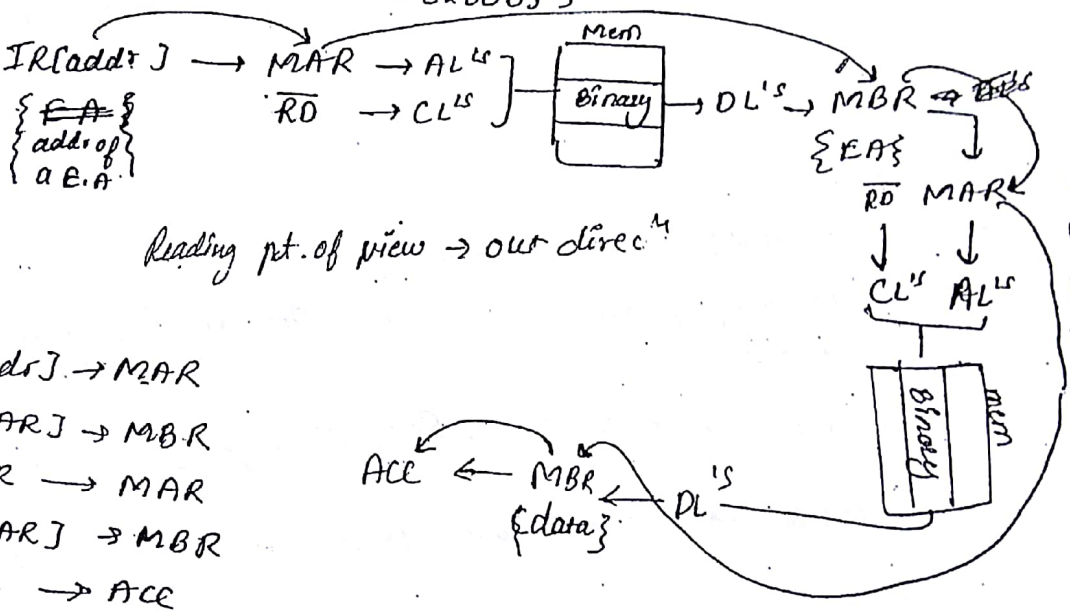
✓ Uprog. for OF using Indirect AM is :-

Instru^n Design with Indirect AM :



Ex. load @2000 ; ACC ← M[[2000]]

H/w design :



Reading pt. of view → our direc^M

Uprog

T_1 : IR[Addr] → MAR

T_2 : M[MAR] → MBR

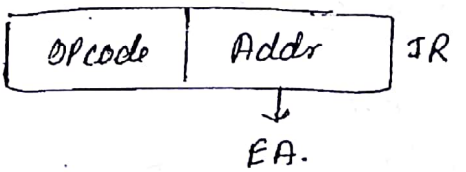
T_3 : MBR → MAR

T_4 : M[MAR] → MBR

T_5 : MBR → ACC

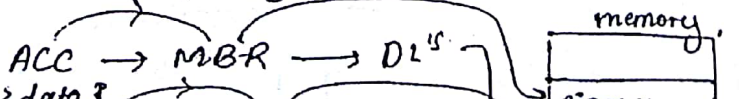
✓ Uprog. for oprand store (WB) using direct mode is,

Instruc^n Design with direct mode :



Ex. Store [2000] ; M[2000] ← ACC.

H/w design :



Uprog

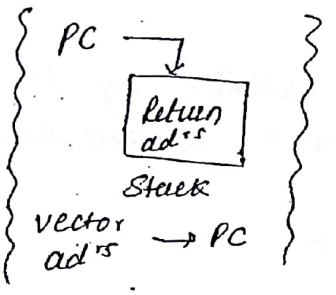
T_1 : ACC → MBR ; ACCout , MBRin

T_2 : IR [Addr] → MAR ; IR addrout , MARin

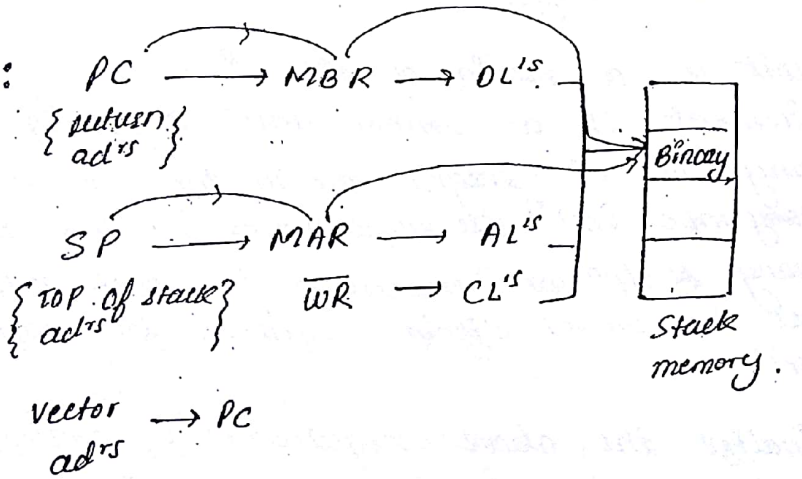
T_3 : MBR → M[MAR] ; MBRSBout , MARout

(M[MAR] : mem.)

✓ Uprog. for interrupt subprogram initialization is



H/w design :



Hprog

T_1 : PC → MBR

T_2 : SP → MAR

T_3 : MBR → M[MAR] } S/S Bus.

(M[MAR] : mem.)

Vector addrs → PC { local BUS.

IF	ID	OF	PO	WB	INT
μprog.	Internal	μprog	Internal	μprog.	μprog.

GATE 2013 (2M)

Q consider the foll. μprog.

T_1 : PC → MBR

T_2 : (X) → MAR (SP)

T_3 : MBR → m/m

(Y) vector → PC

When qn having abstracn then apply op bottom up approach from o/pn. option.

w/e one of the foll. o/pn is satisfied by the above μprog.

a) IF (b) OF (c) cond. JUMP (d) Int. subprog initializa

if X is SP & — call / ret. sub prog

*) control unit Design.

- control unit is a brain of the CPU.
- Pre requirement of a control unit designing is :-
  - → How many control signal are in the base h/w.
  - → How many m/c instn are implemented in the h/w.
  - → How many μ.o/pn are required for each m/c insn.
  - → what are the control signal required for each μo/pn for each instrn
- After finalize the above requirements, control unit is implemented by using the following approaches
  1. Hard wired approach
  2. μprogramed "

(1) Hard wired control unit Design

1. In this design control signal is expressed in a a sum of product expression format.
2. control expression is directly realized by the independent h/w called as hard wired control unit.

④ It is used in the real-time application
⑤ Even a minor modification require redesign & reconnection of a control signal.
⑥ It is not suitable in the design & testing places.
⑦ RISC control unit is a hard wired control unit.

<u>Sample CU</u>

$S_0 : T_1(I_1+I_2)$ 1st µop^n belong to I_1 generate S_0.

① # cs's in the H/W : { S_0, S_1, S_2, S_3 }

② # Ins^n in the H/W : { I_1, I_2, I_3 }
Add (I_1), Mul (I_2), DIV (I_3)

µ op^n nothing but a clk.

③ # µop^n/Ins^n : { T_1, T_2, T_3, T_4 }

④

µo/p^n / Ins^n	I_1 (01)	I_2 (10)	I_3 (11)
T_1	S_0, S_1, S_3	S_0, S_1	S_2, S_1
T_2	S_3, S_0	S_1, S_2	S_2, S_3
T_3	S_0, S_2	S_1, S_2	S_1, S_2, S_0
T_4	S_3, S_2, S_0	S_3, S_1	S_2, S_3, S_0

<u>Hard-wired CU</u>

{ cs : SOP format }

<u>control Expressions</u>

$S_0 : T_1(I_1+I_2) + T_2 I_1 + T_3(I_1+I_3) + T_4(I_1+I_3)$

$S_1 : \underbrace{T_1(I_1+I_2+I_3)}_{T_1} + T_2 I_2 + T_3(I_2+I_3) + T_4 I_2$

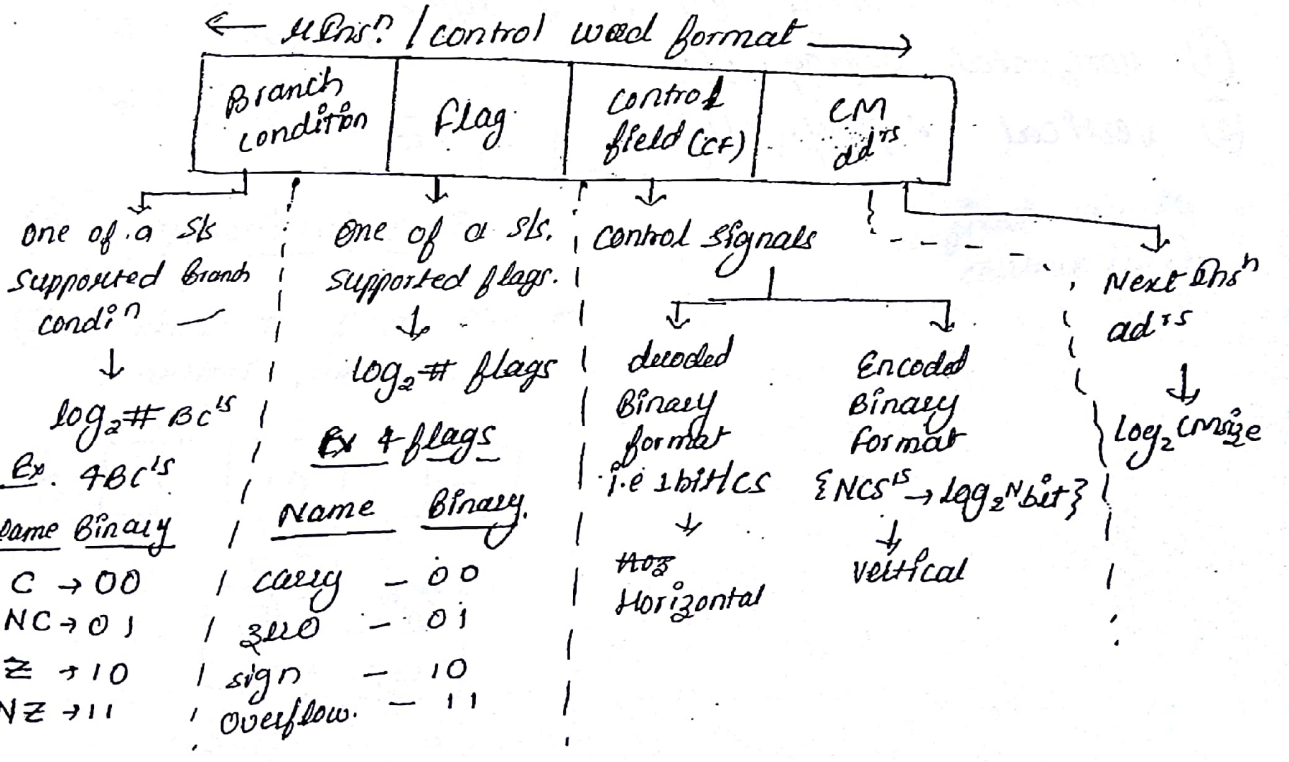
$S_2 : T_1 I_3 + T_2(I_2+I_3) + T_3 + T_4(I_1+I_3)$

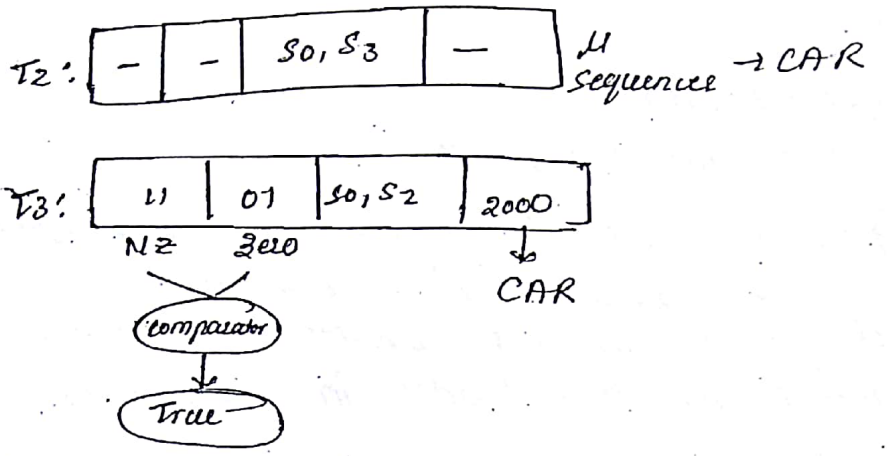
~~Microprogram is~~

★ (2) Microprogram control unit-

- control unit prog. with a microprogram.

- ✓ In this design, control m/m is program with a Microprograms.
- ✓ control m/m is a permanent m/m i.e ROM.
- ✓ In this design μ-sequencer unit is present, used to generate the μInstruction adrs in a sequence.
- ✓ control m/m associated with a CAR & CDR register used to hold ~~the~~ control m/m adrs & control m/m content respectively.
- ✓ In the control m/m μ-Instruction is stored in a following format.





Note: Based on the style of representing "CS" μ Instⁿ is 2 kinds.

① Horizontal μ Instruⁿ

② vertical "

Based on the kind of μ Instruⁿ programmed in the control memory. CU is 2 types.

① Horizontal μprog. CU

② vertical μprog. CU

21/8/2017 (45)
Monday
class-8

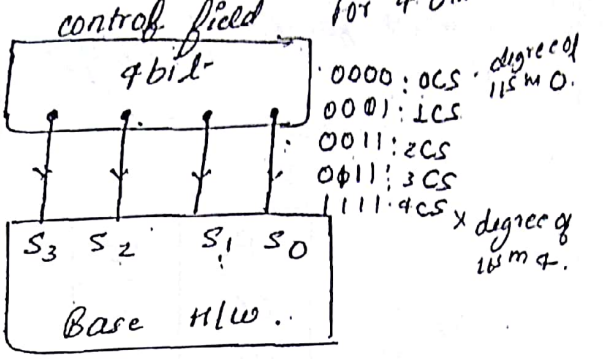
Horizontal :

① # cs's in the Hlw : {S_0, S_1, S_2, S_3}

② decoded form of a "cs" : for 4 bit

i.e 1bit/cs

{1bit → 0 : Disable
1 : Enable}



③ ✓ μIns^n design.

(I) T_1 : {S_0, S_1, S_3} | - | - | 1011 | - |

T_2 : {S_0, S_3} | - | - | 1001 | - |

T_3 : {S_0, S_2} | - | - | 0101 | - |

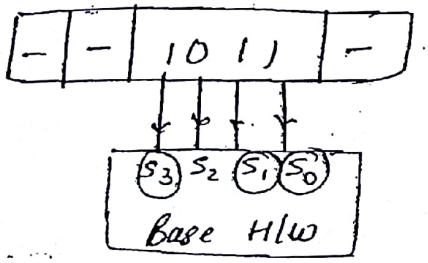
T_4 : {S_0, S_2, S_3} | - | - | 1101 | - |

} control memory.

④ Operational state

fixed length control unit.

(I) : Read T_1 from the control memory



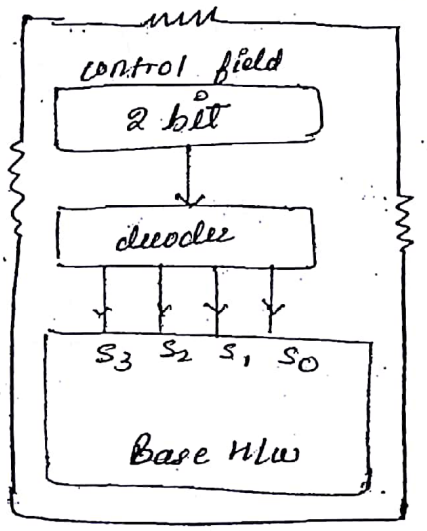
• Vertical

① # cs's in the : $\{S_0, S_1, S_2, S_3\}$
H/w

② Ended form of : $\log_2 4$ = 2bit / cs
a "cs"
i.e $\log_2 N$ bit
{fun^n code fc}

variable length
M.Ins^n design

fc	cs
00	S_0
01	S_1
10	S_2
11	S_3



③ M.Ins^n design

(I$_1$) T_1: $\{S_0, S_1, S_3\}$

–	–	00	01	11	–

T_2 : $\{S_0, S_3\}$

–	–	00	11	–

T_3 : { S_0, S_2 }

–	–	00	10	–

T_4 : { S_0, S_2, S_3 }

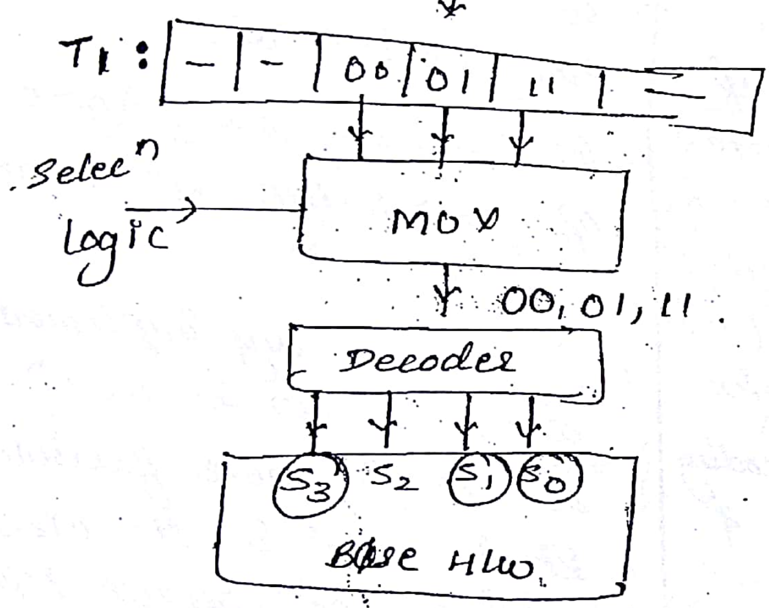
–	–	00	10	11	–

⇓

Programmed into a control m/m.

④ operational state.

(I_1) Read T_1 from the control m/m.



Every tym degree of 11ˢᵐ is one.

High impedance state is dont care or none.

H	V
• For 100 CS's 100 bits require	• For 100 CS's 8 bits require
• Every tym degree of 11ˢᵐ is change	• Every tym degree of 11ˢᵐ is '1'.

degree of 11ˢᵐ 1 → means 1 program execute at a time.

Nothing mention µprog. CU is a vertical.

★ Horizontal vis Vertical.

Horizontal.

- In this design control signal is expressed in a decoded binary format.
- It support longer control word.
- No need of a external decoder to generate the control signal.
  so it is faster than vertical.
- It allows high degree of parallelism none or more than one. "Allows in cf"
- It is a bit flexible compa-red hard wired control unit
- It is not in the practice becoz managing the decoding signal is very complex & expensive

Vertical

- In this design control signal is expressed in a encoded binary format i.e $\log_2^N$.
- It support shorter control word.
- Need of a external decoder to generate the control signal.
  so it is slower than ~~Hor~~ Horizontal.
- It allows low degree of parallelism i.e none or one.
- It allow easy implement-ation of new Instruc^n so it is more flexible.
- It is used in the CISC computer so default Uprog. control unit is vertical.

Note → Ascending order of a control unit design

a) In terms of speed is:
   Vertical < Horizontal < Hardwired.

b) In terms of flexibility is,
   Hardwired < Horizontal < Vertical.

Q) Consider a hypothetical control unit w/c supports 16k control word m/m, H/w supports 64 control signals, & 16 flags. What is the size of a control word in bits & control m/m in bytes using (a) horizonal programming (b) vertical programming.

(i) Horizontal

(1) # CS's in the H/w : 64

(2) decoded form of "CS" i.e 1bit/cs : 64 bits

(3) Hbrs^n design { default with 1 "CS" }

control field

BC	Flag	(CF)	CM adrs

BC → $\log_2 16$ → 4 bit

(CF) → ICS → 64 bits { In this bit pattern one bit status is "1" }

CM adrs → $\log_2 16k$ → $\log_2 2^{14}$ → 14 bit

- HCW size = 4 + 64 + 14
  = 82 bits
- HCM size = 16k CW
  ↓
  16K X CW
  ↓
  16K X 82 bits
  ↓
  $\left(\frac{16 * K \times 82}{8}\right)$ Bytes.

(b) Vertical

(1) # CS's in the HLW : 64

(2) end encoded form : $\log_2 64 = 6$ bit (CS {fc})
of "CS"

(3) μRns design {default with "1" CS}

BC	flag	CF	CM addrs
—	$\log_2 16$ → 4 bit	↓ ICS → 1 fe → 6 bit	$\log_2 16K$ → 14 bit

- VCW size : 4 + 6 + 14
  = 24 bits

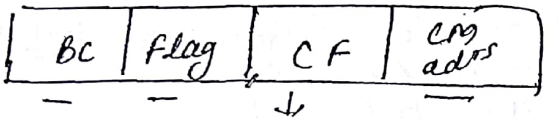
- VCM size = 16K CW
  ↓
  16K × CW
  ↓
  16K × 24 bits
  ↓
  $\left(\frac{16K \times 24}{8}\right)$ Bytes.

(2) A μprog. control unit support 400 control signal. μIns^n is designed in such a way that 4 signals are active.
What is the size of a control field in the μIns^n.

- Nothing is given by default μprog. CU is vertical CU.
- Becoz of vertical CU encoded form of "cs":

$\log_2 400 = 9$ bit/cs
{fe}

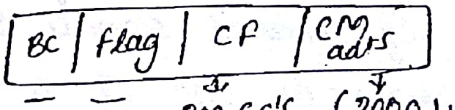
- μIns^n design with "4 CS's"



4 CS's
↓
4 FC's
↓
4 * 9 bits
↓
36 bits.

(Q3) A horizontal μ prog CU supports 600 control signals. μIns^n is designed with 200 signals. what is the size of a μInstrun when the branch adrs is $(2000)_H$ during the execution of a Instruct with unconditional jump.

- Horizontal
- decoded form of "CS"
  i.e 1 bit/cs
- 600 CS's → 600 bits
- μIns^n design with 200 CS's



200 CS's (2000)H
↓ ↓
600 bits 16 bit

{in this pattern any 200 bits status is '1'}

μIns^n size = 1600 + 16
= 616 bits.

Q4 A hypo. control unit support 5 groups of mutually exclusive control signal discribe below.

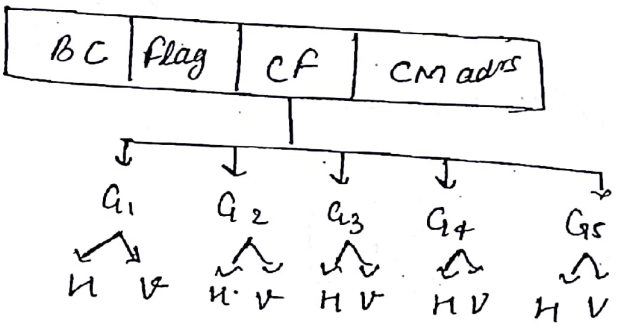
Group	G_1	G_2	G_3	G_4	G_5
CS	1	10	20	30	40

How many bits are saved using vertical over horizontal programming.

Mutual Ex → Every group support vertical & horizontal but not at the same time.

- μrc is shared b/w diff. processes w/o conflict.

μ Instⁿ

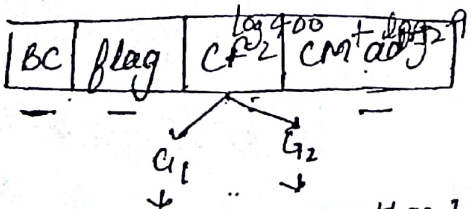


$\log_2 1$
$\log_2 10$
$\log_2 20$

- Horizontal groups : 1+10+20+30+40
  Bit required ⇒ 101 bits
- Vertical groups : $\log_2 1 + \log_2 10 + \log_2 20 + \log_2 30 + \log_2 40$
  Bit required : ⇒ 1 + 4 + 5 + 5 + 6
  ⇒ 21 bits

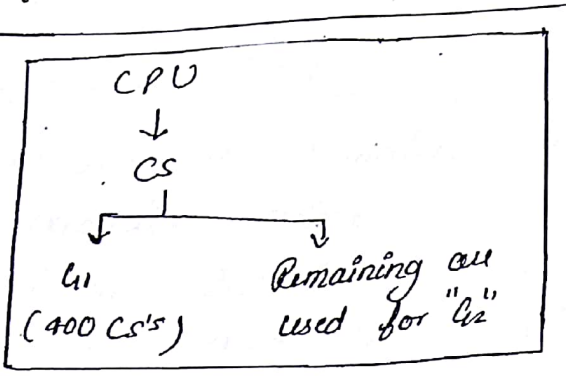
∴ Bits save = (101 − 21)
= 80

Q5. control field of a μ Instⁿ supports 2 groups of control signals in w/c in w/c G_1 indicates none or one of a 400 signals, & group 2 indicate almost 9 signals from the remaining.
What is the size of a control field in a μ Instⁿ



$\log_2 400$

↓ Vertical → $\log_2 400$ → 9 bits
↓ Horizontal → 9 bits space required in the group 2

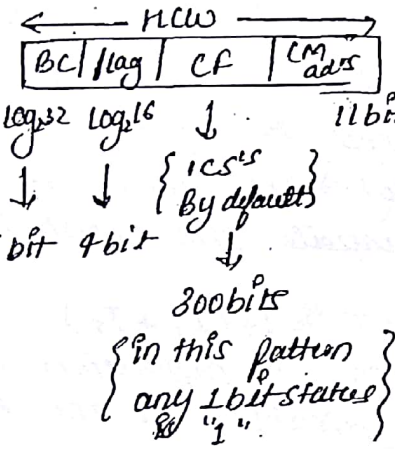


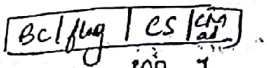
Q. In hypothetical computer Instruc^n set size is 200. Each Instruc^n takes 10 cycles to complete, hardware contains 300 control signals, 16 flags & 32 branch condition. CPU uses horizontal uprog. control unit. What is the size of CAR & CDR registers when one add^{rs} format is used to control the branch logic.

means control m/m add^{rs} is 1.

CAR → control m/m add^{rs} register

CDR → control m/m data register





5 4 300 1

HCW → Horizontal control word.

for every Ins^n 1 CM adr

- # Ins^n in the CPU = 200
- # cycles / Ins^n = 10

{# usn^n (uop^n) / Ins^n}

- Total # usIns^n in the CPU = 200 × 10

= 2000

↓

Programmed into a control memory.

∴ CM size = 2000 LW

↓

$\log_2 2000$

↓

$\log_2 2^{11}$

↓

11 bit adrs

{CAR}

MCW size = 5 + 4 + 300 + 11

{COR} = 320 bits

Q① A hypothetical CPU μw contain 316 CS. manage using decoded binary format, μ instn is deigned with a degree of parallelism of 16. How many bits in the control field is connected to a logic one.

Ans degree of μ instn is 16.

16 is connected to logic 1.

Q② Consides a hypothetical control unit wlc uses the foll. control expressions to generate the control signal.

$$Xin = T_1 + T_2(I_2 + I_3) + T_3 + T_4(I_1 + I_4)$$

during the execution of type 4 expression in wlc ~~μ instn~~ μ op^n Xin signal disable in the hlw.

Ans ~~T2~~. $Xin = T_1 + T_2(I_2 + I_3) + T_3 + T_4(I_1 + I_4)$

T_1 → any → I_4 → Active

T_2 → only I_2 & I_3 → I_4 → disable

T_3 → any → I_4 → active

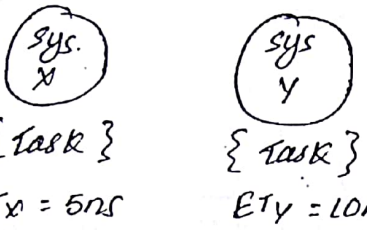
T_4 → only I_1 & I_4 → I_4 → active.

Ans T_2.

Performance Analysis

- Performance is a indirect measurement depends on the execution time.
- Amount of the time required to complete the program is called as execution time.

Eg:



When performance depend. on ex^n time

$$P \propto \frac{1}{ET}$$

- Execution time means CPU time, calculated based on the processor clock i.e

CPU time = ## seconds / Prog.

↓

Instr^n / Prog * ## cycles / Instr^n * ## seconds / cycle

↓ Instr^n count (IC) ↓ CPI ↓ cycle time

$$\text{CPU time} = IC * CPI * \text{cycle time}$$

- In the program, different Instruc^n takes different cycles to complete ∴ CPU

$$\text{CPU time} = \Sigma (IC_i * CPI_i) \text{ cycle time}$$

i: types of Instr^n

- Speedup(s) factor is used to measure the performance gain

Perf^n inversely proportional to exec^n time

- $S = \dfrac{\frac{1}{ET_x}}{\frac{1}{ET_y}}$
- $S = \dfrac{ET_y}{ET_x}$
- $\boxed{S = n}$

↓

{some constant}

[sys. "x" will runs "n" time faster than sys. "y"]

- $S = \frac{y}{x}$
- $2 = \frac{y}{x}$
- $\boxed{y = 2n}$

In apti.

$\boxed{x = 2y}$

Q Consider a hypo. processor w/c is operating with a 500 MHz clk w/c consumes 8 cycles for load & store Ins^n & 6 cycles for ALU Ins^n & 4 cycle for branch Ins^n. Relative frequencies of these Instru^n are 30%, 40% & 30% respectively, what is the performance of CPU in terms of MIPS (million Instru^u per sec)

S
- In Q^n data give 30%, 40% & 30% i.e relative data not absolute. 500MHz
- $\boxed{Avg = \frac{sum}{count}}$

count = 30% + 40% + 30%

= 100%

= 1

when relative data given count = 1, Avg = sum.

$\boxed{\therefore Avg = sum}$

Avg. Ins^n ET = Σ (ICi * CPIi) cycle time

= [(0.3*8) + (0.4*6) + (0.3*4)] * $\frac{1}{500*10^6}$ sec

= 6*2ns

= 12ns

~~1Ins~~ 1 ins^n takes 12ns

? # Ins^n 1sec

$= \frac{1 Ins^n}{12ns}$

$= \frac{1}{12 \times 10^{-9}}$ Ins^n/sec

$= \frac{1}{12} * 10^9$ Ins^n/sec

$\Rightarrow 0.0833 \times 10^9$

$\Rightarrow$ 83.3 MIPS

Q consider 2.3 ns clock cycle processor w/c consumes 6 cycle per load & store Ins^n, 4 cycle per ALU Ins^n, 2 cycle per branch Ins^n, Program contain 20 load Ins^n 10 store Ins^n, 10 ALU Ins^n, 5 Branch Ins^n. What is the avg ~~exec~~ Ins^n & Execn time in ~~kilogram~~. the processor:

In this q^n absolute data give.

(a) cal. execn time.

Program Exen time (ET) = Σ (ICi * CPIi) cycle time * 2.3ns

= (20*6) + (10*6) + (10*4) + [5*2]

~~52~~ = 529 ns

- Avg. Ins^n ET = $\frac{529}{45}$ = 11.75ns. { $\frac{\text{total time required/program}}{\text{total \# Ins}^n \text{ in the prog}}$ }

Amdhal's law

Different Quantitative principles are present in the processor design used to improve the performance, one of the Principal states that improve the performance within a cost & price limit, ^Whole making the common case fast. called as Amdhal's law.

→ Amdhal's law concentrate on the performance gain after enhance the part of the s/s i.e

$$S_{overall} = \frac{\text{Performance of a s/s with enhance (new)}}{\text{Performance of a s/s w/o enhance (old)}}$$

$$S_{overall} = \frac{1}{\frac{ET_{New}}{ET_{old}}}$$

$$\boxed{S_{overall} = \frac{ET_{old}}{(ET_{new})}} \quad \text{--- (1)}$$

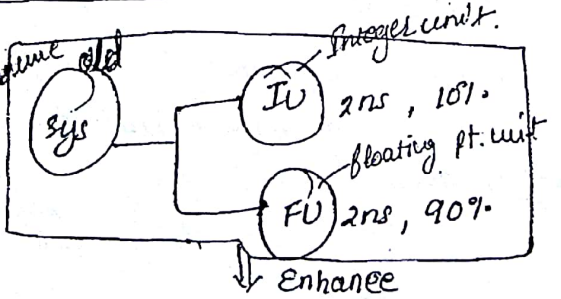
exec^{n} tym of enhanced s/s

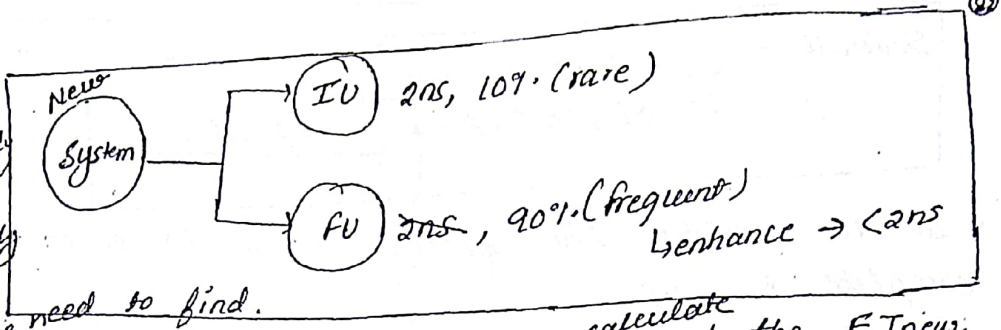
In the enhancement process, only the part of the s/s is modified so new s/s contains two portion

(1) unenhance portion
(2) enhanced "

$\therefore$ $\boxed{ET_{new} = \text{Exec}^{n} \text{ tym of unenhanced portion} + \text{Exec}^{n} \text{ tym of enhanced portion.}}$ --- (2)

Eg





This is not completely new → it is partially new.
In this we need to find.

- Two parameters are required to calculate the ETnew.

(1) fraction enhance (F)
(2) speedup enhance (S)

(1) fraction enhance indicate, how much part of the sys is modified i.e F: Enhanced position
(1-F): unenhanced "

(2) Speedup enhance indicates, performance gain of a enhance part i.e.

$$S = \frac{\text{Performance of new 'f'}}{\text{Performance of old 'F'}}$$

$$S = \frac{1}{\frac{\text{ET of new 'F'}}{1}}$$

$$\frac{\text{ET of old 'F'}}{}$$

$$S = \frac{\text{ET of old 'F'}}{\text{ET of new 'F'}}$$

$$\therefore \text{ET of new 'F'} = \frac{\text{ET of old 'F'}}{S}$$

Substitute the above parameter in equation (2).

$$\text{ET}_{new} = \text{ET of Old } (1-F) + \frac{\text{ET of old 'F'}}{S}$$

Substitute the above value in eq^n (1)

$$S_{overall} = \frac{ET_{old}}{ET \text{ of old } (1-f) + \frac{ET \text{ of old } 'f'}{S}}$$

- Let us consider the relative data to cover the complete s/s i.e.
  - System = 100%
  - identify frequently use parameter
  - $$S_{overall} = \frac{100\%}{(100\% - f) + \frac{f}{S}}$$

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

when data is relative data.

When $S = 1$ no enhancement.

When $ET_{old} = 1$

then ET_{new} always < 1

$\therefore S_{overall} > 1$

- When the system contain multiple enhancement then performance gain is calculated as:

$$S_{overall} = \left[(1 - \Sigma f_i) + \Sigma \frac{f_i}{S_i}\right]^{-1}$$

i: # enhancements.

functionaly will be enhance Amdhal's law is best

Q) consider a computer system used in the scientific applica^n area. Applica^n prog. uses the integer & floating point units during the execution. Floating pt unit is enhance, then it runs two time faster, but in a application program, only the 10% of Instru^n are floating point.

a) what is the performance gain?

~~Ans~~ $\left(\frac{90}{100} + \frac{10}{100 \times 2}\right)^{-1}$ ∴

$$\left[(1-f) + \frac{f}{s}\right]^{-1}$$

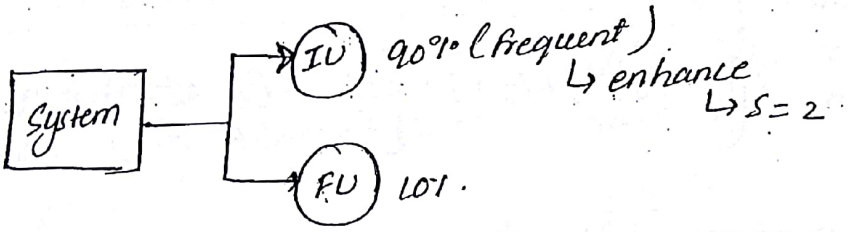
$$= \left[\frac{90}{100} + \frac{10}{100 \times 2}\right]^{-1}$$

$$= \left[(1-0.1) + \frac{0.2}{2}\right]^{-1}$$

$$= 1.05$$

Note ① In the above s/s ~~ease~~ component is modified during the enhancement so performance gain is negligible.

② Amdhal's law stating that improving the performance by modifying the frequently used component in the s/s.



$$\text{Soverall} = \left((1-f) + \frac{f}{s}\right)^{-1}$$

$$= \left[(1-0.9) + \frac{0.9}{2}\right]^{-1}$$

$$= 1.81$$

Q2) In the m/m design cache m/m 12 time faster than main m/m & it is influence by the CPU 70% of the time. What is the performance gain when the cache m/m used in the s/s.

Soverall = ?

$$\text{Soverall} = \left[(1-f) + \frac{f}{s}\right]^{-1}$$

$$= \left[(1-0.7) + \frac{0.7}{12}\right]^{-1}$$

$$= 2.79$$

Q3) 3 enhancements are proposed for a new architecture with the foll. speed ups. $S_1 = 30$, $S_2 = 20$, $S_3 = 15$. If enhancement ① & ② each usable of 25% & enhancement three will be usable of 45% then what is the performance gain.

Ans: $S_1 = 30$, $f_1 = 25\% = \frac{25}{100} = 0.25$

$S_2 = 20$, $f_2 = 25\% = 0.25$

$S_3 = 15$, $f_3 = 45\% = 0.45$

Soverall = ?

$$= \left[\left(1 - (f_1+f_2+f_3)\right) + \left\{\frac{f_1}{s_1} + \frac{f_2}{s_2} + \frac{f_3}{s_3}\right\}\right]^{-1}$$

Q4) consider 4.2 ns clk cycle procedure w/c consumes 10 cycles per load Instn, 7 cycles for ALU Instn & 5 cycles for branch Instn. Relative frequencies of these Instruct are 40%, 30% & 10% respecti-vely.

Processor is enhanced to make the average CPI as one. By this process clock sym is increased upto

$$S = \frac{ET_{old}}{ET_{new}}$$

$ET_{old} = \Sigma(IC_i * CPI_i)$ cycle time

$= [(0.4) * 10) + (0.5 * 7) + (0.1) * 5)] \times 4.2ns$

$= 33.6ns$

$ET_{new} = \Sigma(IC_i * CPI_i)$ cycle time

{CPI = 1}

$= [(0.4 * 1) + (0.5 * 1) + (0.1 * 1)][4.2ns + (40\% * 4.2ns)]$

$= 5.88ns$

$$S = \frac{ET_{old}}{ET_{new}}$$

$$S = \frac{33.6}{5.88} = \underline{5.71}$$

★ High Performance CPU Design

- High performance architecture exhibits the concurrency. concurrency means two or more Instruⁿ execuⁿ at a time.
- Acc. to a ~~flin~~ Flynn's classification, computer architecture is divided into 4 types.

(1) SISD [single Insⁿ stream & single data stream]
(2) SIMD [single Instrⁿ stream & multiple " "]
(3) MISD [Multiple " " & single " "]
(4) MIMD [" " " & multiple " "]

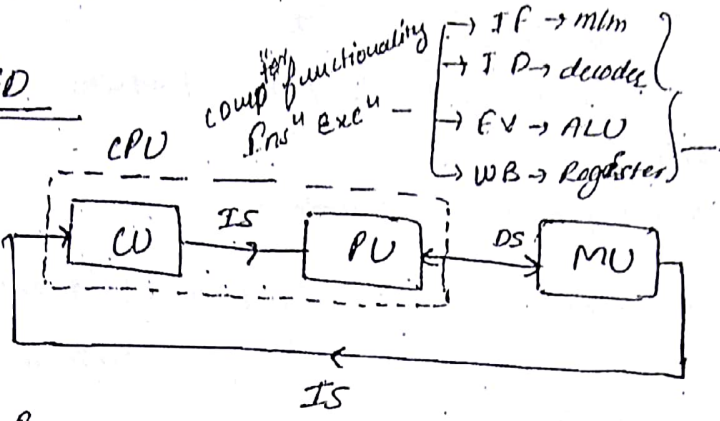
Short cuts

CU : control unit
PU : Processing unit (ALU)] → CPU

MU : mem unit
IS : Ins^n stream
DS : Data stream.

→ MISD is not in practice.
→ two concept MIMD & SIMD allow concurrency

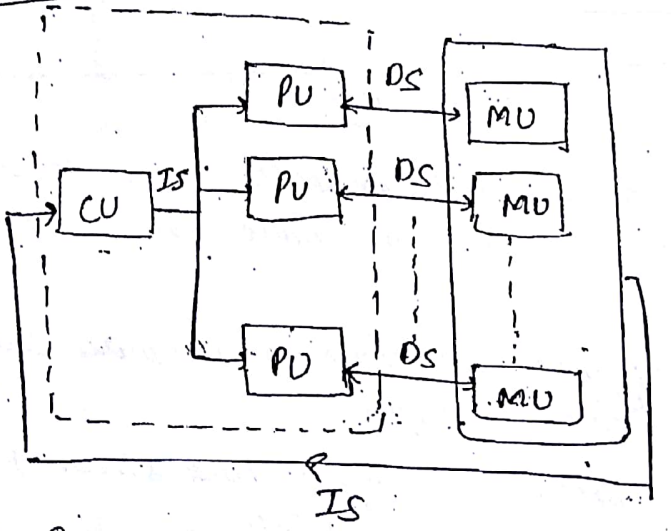
① SISD



Implemented as a uni-processor S/S.

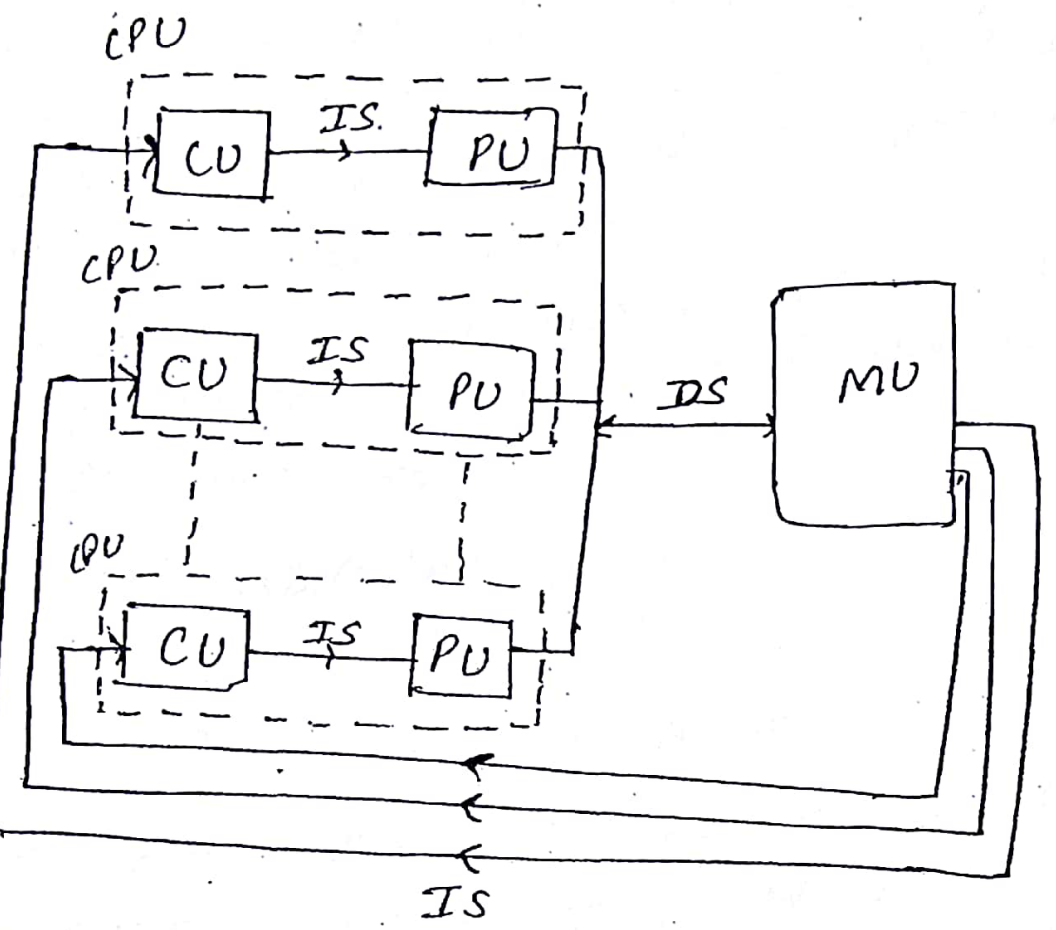
Ex. 8085 µp
8086 µp

② SIMD

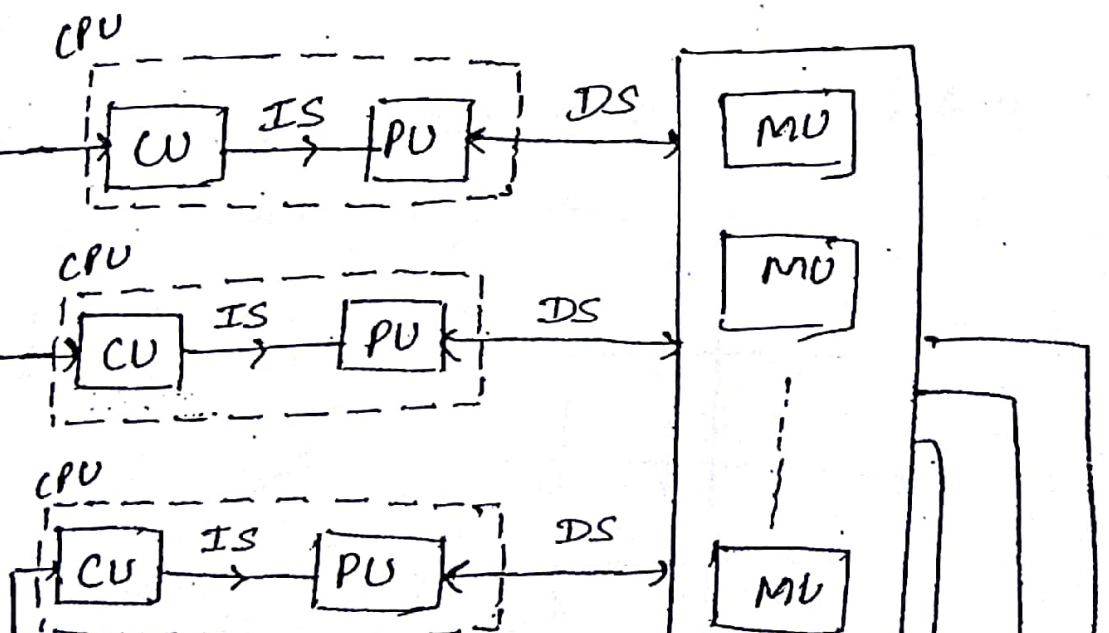


Implemented as a memory processor or S/S.

Ex. Staran Processor PEPE Processor.



It contain multiprocess, But one processor in the use at a time.
It is not yet implemented



- Implemented as a multiprocessor s/s.

Ex. Cray Processor
Cyber Processor

PIPELINING

Func^n Pipelining uses the decomposition technique i.e problem stmt decompose into a independent subproblem assign them to a independent h/w, later connect them in a frugunt sequence.

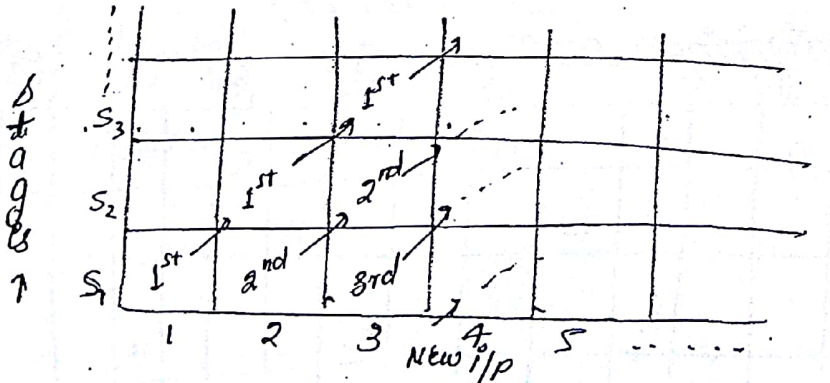
Definition → Accepting the new i/p at one end before previously accepted i/p is appears as a o/p at the other end.

Based on two label "A" & "B"

"A" → After → non-pipeline.

"B" → Before → pipeline

Definition states that insert the new i/p into a pipeline before completion of a old i/p.
New i/p is overlapping with old i/p called as overlapping execu^n.
In a non pipeline s/s, new i/p is inserted after the completion of old i/p called as non overlapping execution.
overlapping execu^n sequence in a pipeline is describe using space time diagram.

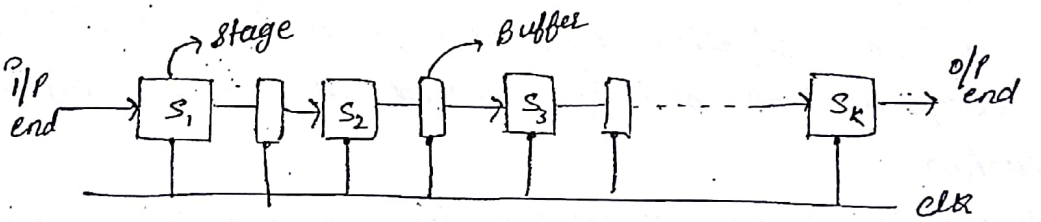


that successful characteristics of pipeline is in every new cycle new i/p must be inserted into the pipeline.

CPI = 1

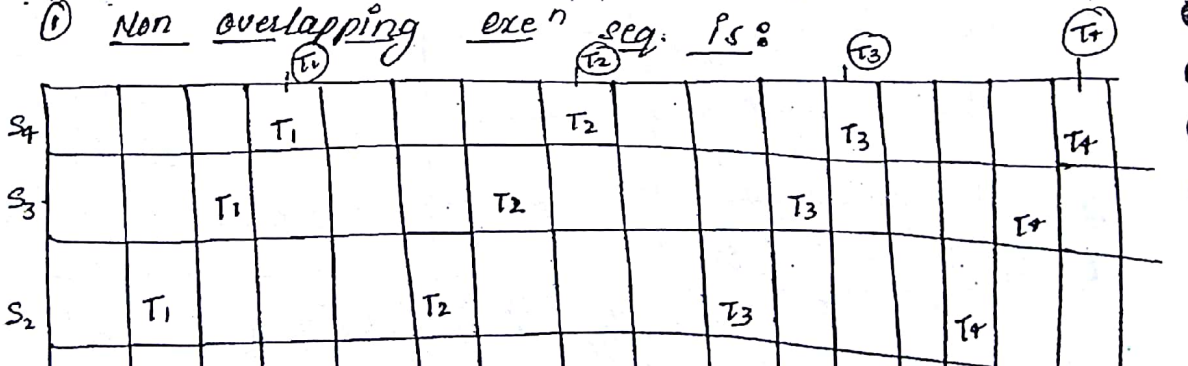
Design:

- Pipeline has two ends - i/p end & o/p end; b/w these ends multiple pipes are interconnected to satisfy the functionality of pipeline.
- Each pipe in the pipeline are called as stage/segment.
- interface register is used b/w the stages to hold the intermediate o/p also name as buffer or latch or pipeline register.
- All the stages in the pipeline along with a interface register is connected to a common clock. So clock adjustment is very important in the pipeline design.

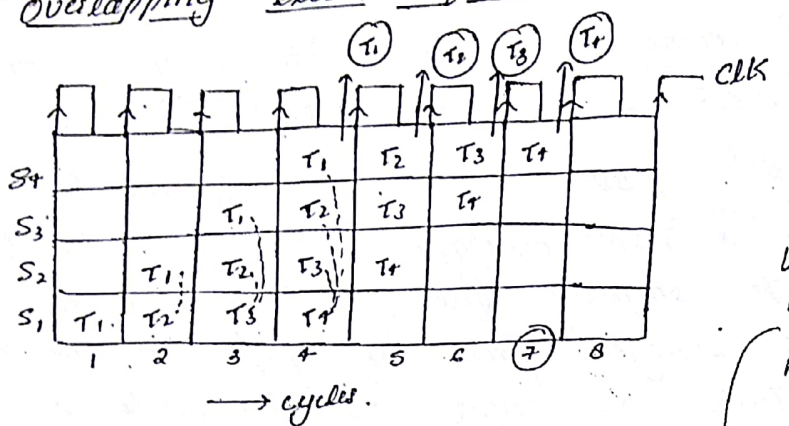


consider 4-stage pipeline (S_1, S_2, S_3, S_4) used to execute 4 tasks (T_1, T_2, T_3, T_4)

① Non overlapping exec seq is:



(2) Overlapping execn seq: fs :-



i/c dont have more i/p in our prog. CPI ≠ 1 but pipeline CPI = 1.
→ so we cant give more i/p.

1st → 4 cycles
3 I/P's → 3 cycles/i/p
7 cycles

K - stages
n - tasks (i/p's)
1st → K cycles
(n-1) → (n-1) cycles
(k+n-1)

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

Programs CPI ≠ 1
Pipeline CPI = 1
→ n ← limited.

★ Performance Analysis

- consider k-segment pipeline with a cycle time of t_p used to execute n-tasks.
- The very 1st task in the pipe pipeline is executed in a non overlapping order so it takes k-cycles to complete. The remaining n-1 tasks are emerged from the pipe at the rate of one task per cycle, so n-1 cycle are required to complete (n-1) tasks.
- Total time required to complete the n-tasks in a pipeline is :

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

- Consider non-pipeline processor used to execute n-task in w/c each task take t_n time to complete ∴ total time required to complete the n-tasks in a non-pipeline is :

$$ET_{nonpipe} = n.t_n$$

- Performance gain of a pipeline is performence of pipe by performance of nonpipe.

$$S = \frac{\text{Performance}_{pipe}}{\text{Performance}_{nonpipe}}$$

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

- When the no. of tasks are increases then n become much larger then K-1, so K+n-1 value will be approach to 'n', under this condition →

- $S = \frac{n \cdot t_n}{n \cdot t_p}$

- $\boxed{S = \frac{t_n}{t_p}}$; $n \leftarrow \infty$ when n unlimited. ← one task exec^{n} timing w.r.t pipeline.

- When the k pipeline stages are perfectly balanced, then one task exec^{n} time in the non task is also equal to a no. of stages in the pipeline i.e,

$$\boxed{t_n = k \text{ cycles} = k \cdot t_p}$$

In this condition S :→

$$S = \frac{k \cdot t_p}{t_p}$$

$\boxed{S = k}$: $\eta = 100\%$ ← efficiency.

stages in pipeline (pipeline depth)

- When the s/s is operating with 100% η then maximum speedup is possible i.e equal to a no. of stages in the pipeline.

100% η — S_{max}

? η — S

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

Throughput pipe = # Task processed / Total time taken to process the tasks

$$\text{Throughput}_{pipe} = n / (k+n-1)\,tp$$

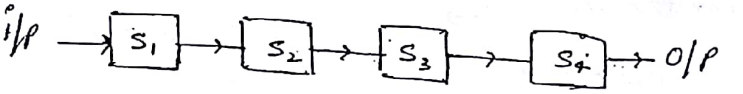
★ Types of Pipeline

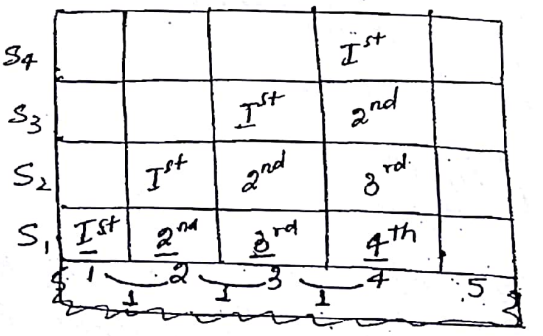
① Linear Pipeline → (In Syllabus)

- This pipeline always contain the forward connection so i/p are always moving to forward direction in a sequence.

∴ latency = 1.

- All the linear pipeline are synchronous pipeline becoz of initiation depends on the clock.

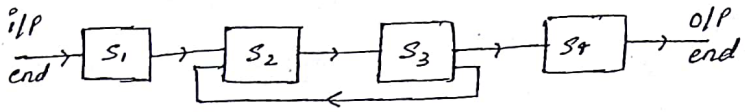




- Linear pipeline in Syllabus.
- non-linear not in Syllabus.

② Non linear pipeline (Not in Syllabus)

- This pipeline contain both forward & backward connection so reservation table is used to process the i/p in the pipeline.
- Latency depends on the reservation table.

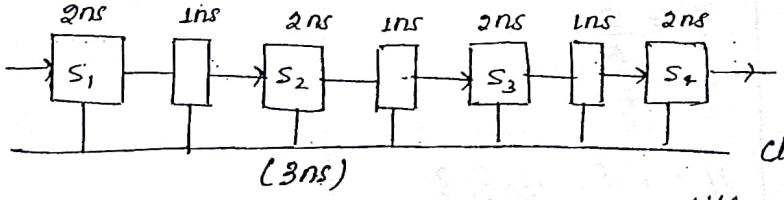


- Reservation table associated with a non-linear pipeline.

	1	2	3	4	5	6
S_1	X					
S_2		X		X		
S_3			X		X	
S_4						X

*) Uniform delay Pipeline design.

- In this pipeline all the stages are taking same amount of time to complete the assigned o/p".



diff. stage, diff delay. Take max delay.

a) **cycle time (tp) = stage delay**

b) If Buffer delay included then,

tp = stage delay + Buffer delay.

© If skew time / setup time overhead is included. then

tp = tp + Overhead.

Note When the pipeline stages are perfectly balanced (uniform delay) then one task execution time in the pipeline is also equal to one task execution

(a) One task ET in pipeline

$k = 4$
$n = 1$
$tp = 2ns$

$ET_{pipe} = (k+n-1)\,tp$

$= (4+1-1)\,2ns$

$= 8ns$

(b) one task ET in nonPipe (tn)

$tn = S_1 + S_2 + S_3 + S_4$

$= 2+2+2+2$

$= 8ns \{= k\,tp\}$

$\Rightarrow$

$$S = \frac{tn}{tp} \; ; \; n \leftarrow \infty$$

$$S = \frac{8}{2}$$

$$S = 4 \quad \{= k\}$$

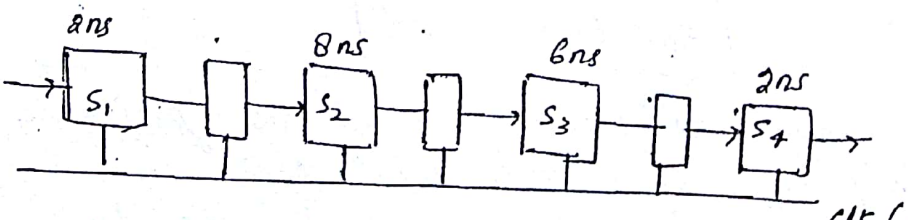
$$\eta = \frac{S}{k}$$

$$\eta = \frac{4}{4}$$

$$\eta = 1 \; \{= 100\%\}$$

★ Non uniform delay Pipeline design.

In this pipeline different stages takes different sizes time to complete the assigned operation.

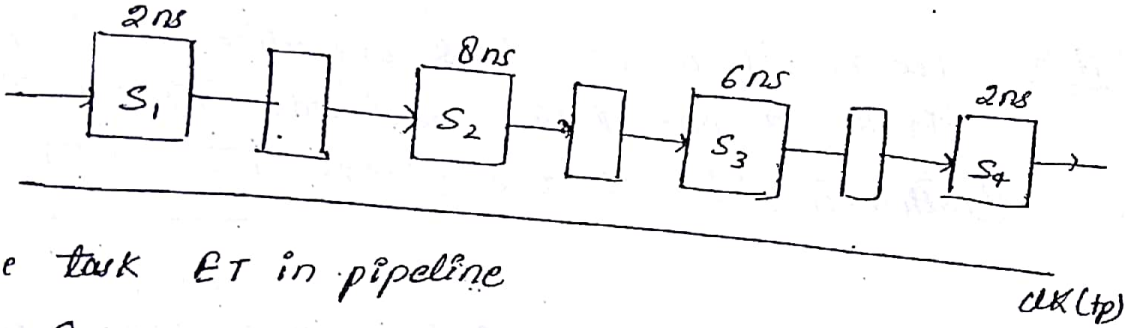


a) $\boxed{tp = \max(\text{stage delay})}$

b) If Buffer delay included then

$$\boxed{tp = \max\begin{pmatrix}\text{stage} \\ \text{Delay}\end{pmatrix} + \begin{matrix}\text{Buffer} \\ \text{delay}\end{matrix}}$$

- When the pipeline stages are not perfectly balanced (non-uniform delay.) then one task execu time in the pipeline is always greater than one task execun time in the non pipeline.



a) one task ET in pipeline

$$\left\{\begin{matrix} k=4 \\ n=1 \\ tp = 8ns \end{matrix}\right\}$$

$$ETpipe = (k+n-1)\,tp$$
$$= (4+1-1)\,8ns$$
$$= 32ns$$

this is only for one task we execute set of Ins

b) one task ET in nonpipe (tn)

$$tn = (S_1 + S_2 + S_3 + S_4)$$
$$= 2+8+6+2$$
$$= 18ns \; (\neq k\,tp)$$

$$S = \frac{t_n}{t_p} \; ; \; n \to \infty$$

$$S = \frac{18}{8}$$

$$S = 2.25 \quad \{\neq k\}$$

$$\eta = \frac{S}{k}$$

$$\eta = \frac{2.25}{9}$$

$$\eta = 56.25\% \quad \{\neq 100\%.\}$$

Note: Let T_1 is a one task execution timing pipeline.
T_2 is a one task execution timing non pipeline.
Relationship b/w the T_1 & T_2 is $\boxed{T_1 \geq T_2}$

Q. consider an Instruc pipeline w/c has a speedup factor 13 while operating with 75%. η. what could be the no. of stages in the pipeline.

$S = 13$

$\eta = 75\%$

$k = ?$

$$\eta = \frac{S}{k}$$

$$75\% = \frac{13}{k}$$

$$k = \frac{13}{0.75}$$

$$k = 17.33$$

Q Consider two pipelines A & B. where pipeline A is having 8 stages of uniform delay of 500 MHz clock. Pipeline B is having 5 stages with a respective delays of 2ns, 6ns, 8ns, 1ns. How much time is saved when the 100 tasks are pipelined using A instead of B.

Pipe-A → { K=8, n=100, tp = 1/500M sec = 2ns }

$ET_A = (k+n-1)\,tp$

$= (8+100-1)\,2ns$

$= 214ns$

Pipe-B : { K=5, n=100, tp = Max(2, 6, 8, 1, 3) = 8ns }

$ET_B = (k+n-1)\,tp$

$= (5+100-1)\,8ns$

$= 832ns$

Time save $= ET_B - ET_A$

$= 832 - 214$

$= 618ns$

Q Consider a 4 stage pipeline with a stage delays of 20ns, 30ns, 40ns 10ns what is the performance gain of a pipeline

if in Q^n n is not given so consider it is ∞.

{ K = 4, tp = 40ns }

$$S = \frac{tn}{tp} \quad ; \quad n \leftarrow \infty$$

$tn = S_1 + S_2 + S_3 + S_4$

$= 20 + 30 + 40 + 60$

$= 100ns$

$S = \frac{100}{40}$

$\boxed{S = 2.5}$

Q. consider 4 stage pipeline with a respective delays of 20ns, 40ns, 30ns & 60ns. Interface registers are used b/w the stages have a delay of 2ns. What is the performance gain of a pipeline when the very large no. of tasks are executed.

$\boxed{S = \frac{tn}{tp}} \quad n \leftarrow \infty$

$tp = max(\text{stage delay} + \text{Buffer delay})$

$= max(22, 42, 32, 60)$

$= 60ns.$

↳ no buffer at last

~~$tn = S_1 + S_2 + S_3 + S_4$~~

~~$= 20 + 2 + 30 + 2 + 40 + 2 + 60$~~

~~$= 158$~~

~~$= 158$~~

~~60~~

~~$= 2.5$~~

$tn = S_1 + S_2 + S_3 + S_4$

$= 20 + 40 + 30 + 60$

$= 150ns$

$S = \frac{150}{60}$

$\boxed{S = 2.5}$

Note: In a non-pipeline design buffer is not used Becoz data dependency problem doesn't occured in the non pipeline.

Q consider 4 stage Pipeline with a respective

into 2 substages of a equal delays. what is the clk frequency in the enhanced pipelining. ncb

$k = 4$

$t_p = 20ns$

Pipe old : S_1, S_2, S_3, S_4

8ns, 20ns, 5ns, 6ns

Pipe new : S_1 8ns, S_2 10ns (1) → S_{2_1} 10ns, S_{2_2} 10ns; S_3 5ns; S_4 6ns

$t_{p_{newpipe}} = \max(8, 10, 10, 5, 6)$

$= 10ns$

Clock Frequency $= \frac{1}{10ns}$

$= \frac{1}{10 * 10^{-9}} Hz$

$= \frac{1}{10} * 10^9 Hz$

$\Rightarrow 100 MHz$

Q consider a non pipeline processor with a average of CPI of 8 wcd to execute 200 task. The corresponding task are executed on a pipeline processor wle is operating with a ~~non~~ ~~non~~ 2ns clk. when the non pipeline s/s clk cycle tym is 4ns. what is the performance gain.

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

$= 6400 ns$

$ET_{pipe} = \Sigma (IC_i * CPI_i)$ cycle time

$\{CPI = 1\}$ $= (200 * 1) ns$

$= 400 ns$

$S = \frac{6400}{400}$

$\boxed{S = 16}$.

Q consider 4.6 ns clk cycle Processor w/c consumes 8 cycles per load Instruⁿ, 4 cycles per ALU Instruⁿ & 2 cycles for branch Instruⁿ.

a) Frequency of a respective Instⁿ are 30%, 40%, & 30%.

b) Processor is enhanced with a pipeline.

c) In this process set up time overhead is 0.8 ns. What is the performance gain.

→ It is not given directly do generalization

i.e $ET = (IC_i * CPI_i)$ cycle time

~~$= 8 * 4.6 ns$~~

a) $ET_{nonpipe} = \Sigma (IC_i * CPI_i)$ cycle time

$= [(0.3 * 8) + (0.4 * 4) + (0.3 * 2)] 4.6 ns$

$= 21.16 ns$

$ET_{pipe} = \Sigma (IC_i * CPI_i)$ cycle time

$\{CPI = 1\}$ $= [(0.3 * 1) + (0.4 * 1) + (0.3 * 1)] [4.6 + 0.8)$ ns

$= 5.4 ns$

$S = \frac{ET_{nonpipe}}{ET_{pipe}} = \frac{21.16}{5.4} \Rightarrow 3.91$

Q Consider 4 stage pipeline where different instruc

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

(a) How many cycles require to complete the Instn

(b) performance gain.

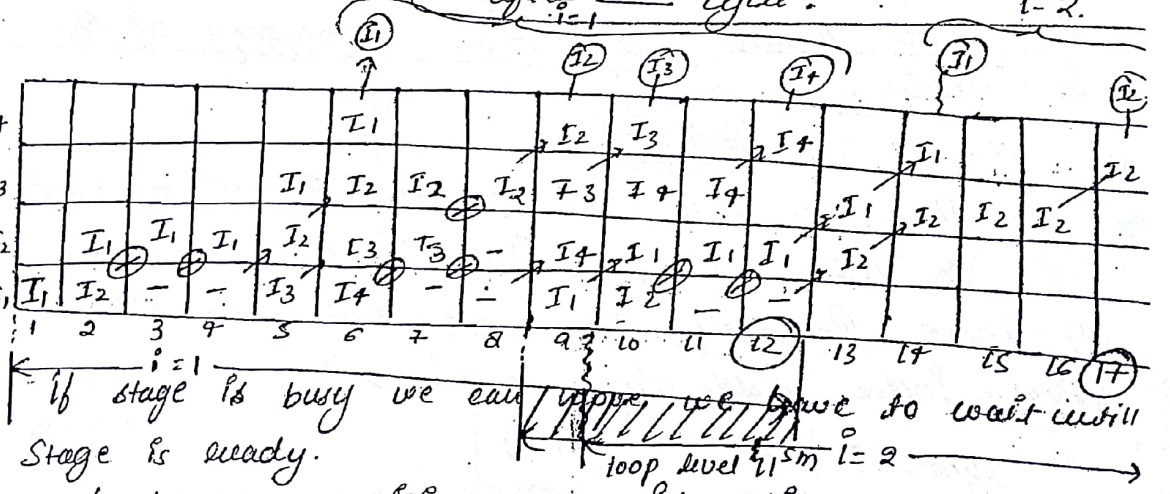
(c) foll. code is executed on above pipeline

```
for (i=1 ; i<=100 ; i++)
{
  I1;
  I2;
  I3;
  I4;
}
```

Output of the "I_1" for ($i=2$) will be available after ___ cycle.

(d) prog. ET w/o loop level usm.

'–' wait on same stage.



If stage is busy we can't proceed we have to wait until Stage is ready.

'–' mean waiting means i/p still waiting they can't insert.

'—' indicating waiting not available.

(a) 12

(b) $S = \frac{ET_{nonpipe}}{ET_{pipe}}$

$ET_{nonpipe} = I_1 + I_2 + I_3 + I_4$

$(S_1 + S_2 + S_3 + S_4)$

If opⁿ not match d b) c) d)

→ Pⁿ structure w/o loop level usm

→ for each PE will

(c) 17

(d) Prog Exec = 100 * 12 cycles

= 1200 cycles.

RISC Pipeline

→ To analysis the implementation issues of pipeline design, let us consider RISC pipeline as a reference mode.

→ RISC CPU supports 5 stage Instruction pipeline used to execute the Instructions.

→ RISC CPU Instruction set contain 3 category of the Instructions.

① data transfer Instruction

② In the RISC CPU load & store Instruction are used as a data transfer Instructions.

In these Instruction, dest mem will be referred using Index addressing mode, &

Syntax:

load	r0	3(r1)

↓

r0 ← M[3 + [r1]]

store	3(r1)	r0

↓

M[3 + [r1]] ← r0

② data manipulation Instruction

In the RISC CPU ALU O/P are always performed on a register data.

Syntax:

Add	r0	r1	r2

③ Transfer of control Instruc^n

ⓐ unconditional TOC

syntax: | JMP | 2000 |

↓

PC: ~~seq. addr~~ 2000

ⓑ conditional TOC

Syntax: | DJNZ | r0 | 2000 |

↓

Dec r0

↓

NZ compare 'r0' Status

↓ True → PC: ~~seq. addrs~~ 2000

↓ False → PC: seq. ad^s

DJNZ → decrement Jump if non zero

| JNZ | 2000 | ⟹ DJNZ r0 2000

↓

(NZ) compare status of a prev. Instn

To execute the above Instruc^n set 5 stage pipeline is used in the RISC CPU.

Different stages present in the pipeline is as follows -

① Stage 1 [Instn fetch] : (IF)

- In this stage CPU reads the Instr^n from mem based on the PC. Simultaneously PC will be incremented to a next Instruc^n ad^{rs} in a sequence.

② Stage 2 [Instr^n decode] : (ID)

- In this stage 2 o/p^n are perform :-

① decode the Instr^n

② fetch the operand from the Register file.

- This stage also contain the comparator logic to evaluate the branch condition.

③ stage 3 (execute) (Ex)

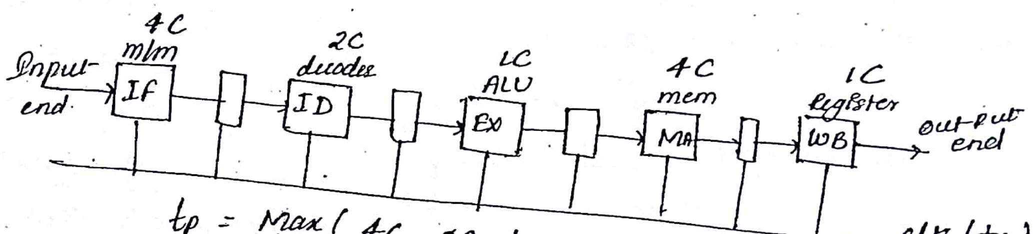
In this stage ALU o/p^n are performed.

④ stage 4 [m/m access] (MA)

In this stage m/m read & m/m write o/p^n are performed to access the data.

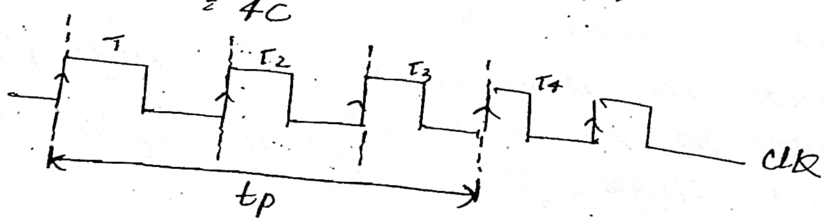
⑤ stage 5 [write Back] (WB)

In this stage register write o/p^n is perform to store the result into a register file.



$t_p = Max(4C, 2C, 1C, 4C, 1C)$

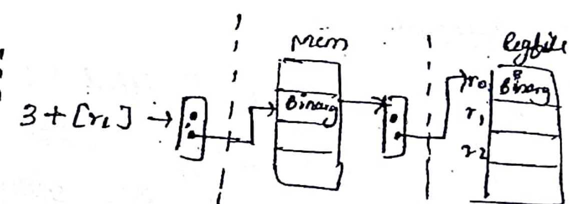
$= 4C$



Load r0, 3(r1)

PC ← PC + SS

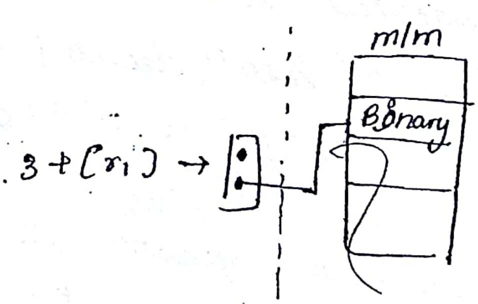
mem RD
S: mem (3, r1)
D: reg (r0)
r1 = value



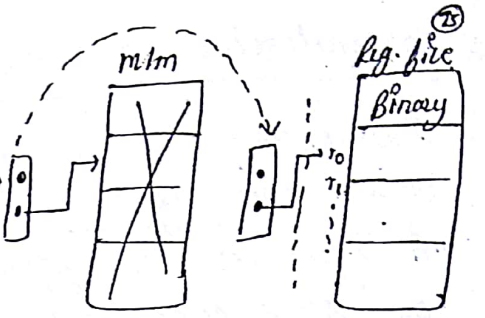
Store 3(r1), r0

PC ← PC + SS

mem WR
S: reg (r0)
D: mem (3, r1)
r0 = value
r1 = value



Add r_0, r_1, r_2	+
$PC \leftarrow PC+SS$	S : reg(r_1, r_2)
	D : reg(r_0)
	r_1 = value
	r_2 = value



Note : This pipeline is suitable for the RISC Instru^{n} set only.

unconditional transfer of control.

JMP 2000	Uncon^{l} TOC
$PC \leftarrow PC+SS$	TA : 2000
	PC : Seg address 2000

conditional transfer of control.

DJNZ r0,2000	cond. TOC
$PC \leftarrow PC+SS$	COND : NZ
	r0 = value
	↓
	Dec r0
	↓
	NZ compare r0
	1
	T : (PC : seq 2000)
	F : (PC : seq)

DJNZ
↓
decrement Jump if non zero.

Dependencies in the pipeline

- Dependency is a major problem in the pipeline causes extra cycles.
- A cycle in the pipeline w/o new i/p is called as stall cycle (extra cycle).

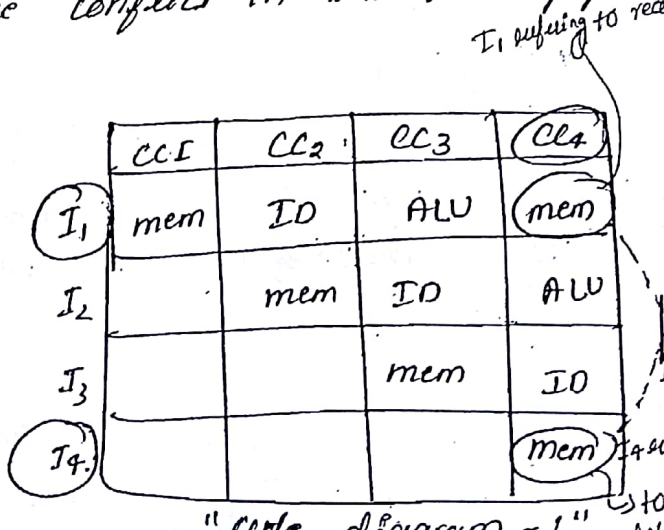
when stall is present in the pipeline then CPI ≠ 1.
→ cycle is time w/o new i/p.

- Three kinds of dependencies are possible in the pipelining

(1) structural dependency
(2) data "
(3) control "

(1) structural dependency

- structural dependency will be occured in the pipeline due to a resource conflict.
- Resource may be a register or m/m or function unit (ALU).
- Resource conflict in the wise pipeline is describe below.



	CC1	CC2	CC3	CC4
I1	mem	ID	ALU	mem
I2		mem	ID	ALU
I3			mem	ID
I4				mem

"cycle diagram - 1"

1st & 4th stage both are ovelap in the pipeline resrc depicted. There may be conflict in term of w/r

- In the above excerpt seq. I_1 & I_4 both of the instruⁿ all tries to access the same resource

- Conflict is a unsuccessful o/p", so to make it successful we need to keep the I_4 into a waiting, until resource become available. This waiting creates a stalls in the pipeline i.e ↳cycle in the pipeline w/o new i/p. describe below:

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1	mem	ID	ALU	mem	WB		
I_2		mem	ID	ALU	mem	WB	
I_3			mem	ID	ALU	mem	WB
I_4				IF	ID	IF	mem
					stalls	no i/p stall	i/p is given no stall.

structural hazards.

- To minimize the structural stalls h/w, techniques is used i.e renaming ("Re-naming").

7 cycles − 4 i/p's = 3 stalls

- In the cycle diagram ⑤ I_1 refers the m/m in a 4th stage to read or write the data, simultaneously, I_4 refereers the m/m in a Ist stage to access the Instruction in the same cycle of CC_4.
- when the Instruction & data both are present in the same m/m then the above situation create conflict.
- Renaming mechanism states that divide the

as code m/m (CM) & Data m/m (DM).

- Refer the code m/m in 1st stage & refer the data m/m in the 4th stage of the pipeline, so accessing of these two modules, so the same cycle doesn't create the conflict.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1:	CM	ID	ALU	DM	WB		
I_2:		CM	ID	ALU	DM	WB	
I_3:			CM	ID	ALU	DM	WB
I_4:				CM	ID	ALU	DM
I_5:					CM	ID	ALU
I_6:						CM	ID
I_7:							CM

When the pipeline is fully occupied.

7 cycles - 7 i/p = 0 stalls

---

② data dependency -

- consider the program where instruc 'J' follows instruc 'I' in a program order.
- Prog. In the prog. 'J' coming after I.

```
I : Insn
J : Insn
: :
```

- data dependency condn will be occur when the Instruc J tries to read the data before Insn 'I' writes it.

non pline.

Gone after another

I & o/p of I_1 is same.

ecuted in a non
lency condition doesnot
after I_1. so I_2
after I_1 ryt shift

it is possible
in non
pipeline
SIS
executed in a pipe
ncy condition will be
long with a I_1.
req
to value before I_1
incorrectly read the
ta loss] described below.

CC_4

23/8/2017
Wednesday
class - 10

- To detect the data dependency condition, some logic will be maintain in the ID stage of a pipeline.
- Structure of the logic is :-

ID Stage

TOMASULO Algorithm

S.No	Func^n unit	Destin^n	Independent src^1 (I)	Independent source2 (I)	dependent source1 (D)	dependent source2 (D)	status
I_1	ADD	r_0	r_1	r_2	—	—	0
I_2	MUL	r_3	—	r_2	r_0 [I_1]	-	0

Reg-file Access
r_1 = value
r_2 = value
⇒ Allocated

Ex Stage
r_2 = value ⇒ NOT allocated.

Status
- 0 : Execu^n not yet completed.
- 1 : Execu^n completed

- By using the above logic we need to stop the accessing of a dependent data untill data becomes available, so I_2 will be waiting on ID stage untill the I_1 execu^n is completed. This waiting creates stalls in the pipeline described below:

WB allow write Read sequence 11 y.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1	IF	ID	EX	MB	WB status = 1 {r_0 : updated}		
I_2		IF	ID Src1 (r_0) (D), Src2 (r_2) (I) — independent. I_1 dependent r_2 : value	ID	ID r_0 : value [Read]	EX — — →	Read-after-write
I_3							

{ 7 cycles - 4 i/p's = 3 stalls }.

- To minimize the data stalls h/w technique is used i.e operand forwarding also name as "short circuiting" / "By passing".
- This technique states that we the buffer blw the stages to hold the intermediate o/p. so the dependent Ins^n will be access the data from the buffer before update the register file.

data dependency could never possible in non pipeline.

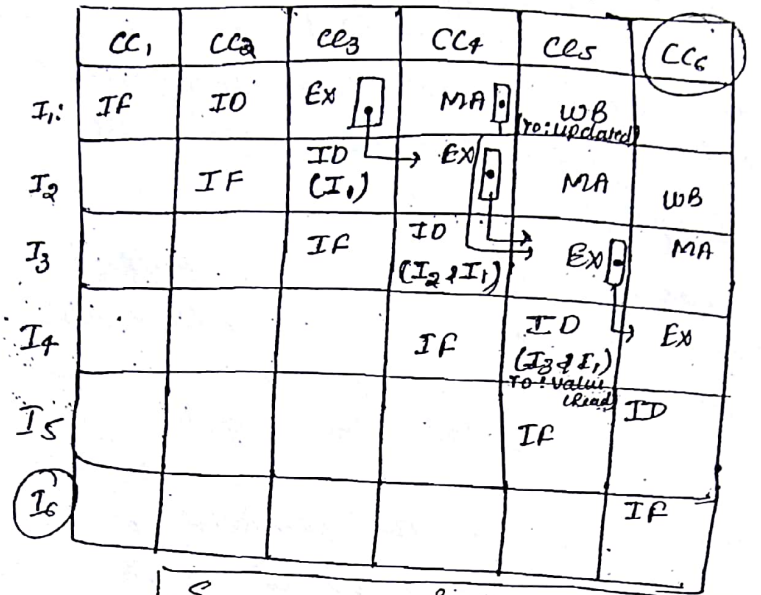
- In this technique buffer is not connected at the end of last stage becoz of in the WB stage write - read o/p^n are performedon a same register data.
- In the RISC pipeline it is possible so no need of the buffer at the end of WB stage, if h/w is not flexible to do this o/p^n then buffer is introduce at the end of last stage to handle the data dependency.
  Becoz WB allow write read sequence..

Q consider the foll. prog. segment execute on a RISC pipeline with operand forwarding technique.

I_1 : Add $(r_0), r_1, r_2$
I_2 : Sub $(r_3), (r_0), r_2$
I_3 : Mul $(r_4), (r_3), (r_0)$
I_4 : DIV $r_5, (r_4), (r_0)$

I_1 to I_2 adjacent
I_2 to I_3 "

- Adjacent data dependency is a true data dependency
- Non adjacent data dependency is a data dependency.
  I_3 to I_1 not adjacent
  I_4 to I_1 " "



{ 6 cycles − 6 i/p's = 0 stalls }

Q3. consider the foll. code executed on a RISC pipeline How many cycles are required to complete the instrn where all the instrn are spending one cycle on the all stages but load instrn takes 3 cycles on MA [4 stages] of a pipeline.

I_1 : Load (r0), 3(r1)

I_2 : Add r1, (r0), r1

I_3 : Load (r2), 3(r3)

I_4 : Sub r3, (r2), r4

Ans RISC Pipeline { when it is RISC pipeline by default K = 5 operand forwarding default }

	IF	ID	EX	MA	WB
I_1	1	1	1	3	1
I_2	1	1	1	1	1
I_3	1	1	1	3	1
I_4	1	1	1	1	1

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7	CC_8	CC_9	CC_{10}	CC_{11}	CC_{12}	CC_{13}	CC_{14}	CC_{15}
I_1	IF	ID	Ex	MA	MA	MA	WB								
I_2		IF	ID (I_1)	ID	ID	ID →	Ex	MA	WB						
I_3			IF	IF	IF	IF	ID	Ex	MA	MA	MA →	WB			
I_4				////	////	////	IF	ID (I_3)	ID	ID	ID	Ex	MA	WB	

3 RAW Dn load Dns^n data is present in the mlm Hazards.

cycle = 4

. . . . . .

b) consider 4 stage pipelining (IF, ID, Ex, WB) used to ~~exed~~ execute the following code.

I_1 : Div r_0, r_1, r_2

I_2 : MUL r_3, r_4, r_5

I_3 : Add r_4, r_3, r_2

I_4 : Sub r_3, r_4, r_1

where all the Dns^n are spending 1 cycle on all the stages but DIV takes 4 cycles, ~~or~~ & MUL takes 3 cycles on Ex stage.

Operand forwarding used in the pipelining.

a) How many cycles are required to compute the code. (with operand forwarding optimization)

b) How many cycles are saved by using the optimizn over w/o optimizn in the pipeline to handle the data dependency. → w/o operand forwarding.

→ 1st decide hypo: or RISC

→ then decide operand forwarding is there or not

→ then draw table. for I_1 I_2 I_3 I_4

	~~IF~~	~~ID~~	~~Ex~~	~~WB~~ WB
I_1	1	1	4 4	1
I_2	1	1	3 3	1
I_3	1	1	1	1
I_4	1	1	1	1

	cc_1	cc_2	cc_3	cc_4	cc_5	cc_6	cc_7	cc_8	cc_9	cc_{10}	cc_{11}	cc_{12}
I_1	If	ID	Ex	Ex	Ex	Ex	WB					
I_2		If	ID	ID Ex	ID Ex	ID Ex	EX	Ex	Ex	WB		
I_3			If	If	IF	If	ID (I_2)	ID	ID	Ex	WB	
I_4							If	If	If	ID (I_3)	Ex	WB

ID is waiting because Execution not available

	cc_1	cc_2	cc_3	cc_4	cc_5	cc_6	cc_7	cc_8	cc_9	cc_{10}	cc_{11}	cc_{12}	cc_{13}	cc_{14}	cc_{15}	cc_{16}
I_1	If	ID	Ex	Ex	Ex	Ex	WB									
I_2		If	ID	ID	ID	ID	Ex	Ex	Ex	WB Ex updated	~~WB updated~~					
I_3			If	If	If	If	ID (I_2)	ID	ID	ID	ID Read	~~Ex ID Read~~	WB Ex updated	~~WB updated~~			
I_4							If	If	If	If	If	ID (I_3)	ID	ID (Read)	Ex	WB

Control Dependency

- control dependency will be occur in the pipeline when the transfer of control instn are executed.
- consider the foll. code.

```
1000 : I1      ↓ falling / fall through path   execn from starting adrs is falling.
1001 : I2      ↓
1002 : I3 (JMP 2000)
1003 : I4
  :     :
2000   BI1     ↓ taken
2001   BI2     ↓ path
  :     :
```

Expected o/p sequence:

$I_1 - I_2 - I_3 - BI_1 - BI_2$

PC:1000	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6
I_1:	IF PC:1001	ID	EX	MA	WB	
I_2:		IF PC:1002	ID	EX	MA	WB
I_3:			IF PC:1003	ID unwanted Instr PC: ~~1004~~ 2000	EX	MA
I_4:				IF PC: ~~1004~~	ID	EX
BI_1:					IF PC:2001	
BI_2:						IF PC:2002

IF :

T_1 : PCout , MARin

T_2 : MARout , MBRSBI, PCout , Inc, PCin

T_3 : MBRout , IRin

- the O/P we are expecting is missing when unwanted Instr^n is executed.

Actual O/P : $I_1 - I_2 - I_3 - (I_4) - BI_1 - BI_2$

I_4 ↓ unwanted

- I_4 executed extra. when the unwanted prog. executed our PC's is unsafe & that unsafe Instru^n is control dependency.

- In the above execution sequence unwanted Instruc^n is overlapped with a jump Instru^n when it is executed in the program, then the program functionality is missing. This kind of problem in the pipeline is called as control dependency.
- To handle the above problem we need to stop the unwanted Instru^n execution, for this purpose (NOP) Instruc^n is ~~execut~~ inserted after the jump Instru^n to suspend the unwanted Instruction fetch. This process is called as flush / freeze O/P^n.

Instruc^n → no O/P^n → delay Ins^u

- flush O/P^n creates stall in the pipeline describe below.

IF PC:1001	ID	Ex
	IF PC:1002	ID
		IF PC:1003

At what Stage target adr available in Pipeline — 1

Note: In the RISC pipeline branch penality is always 1 becoz target adrs is available in the 2nd stage from the branch.

- No. of stalls created during the program execution is calculated as branch frequency into branch penality

Branch frequency * Branch Penality

↓ ↓

Branch Instn in the program. # stalls / Branch

Note: • To minimize the control stalls H/w technique is used i.e branch prediction buffer. (loop buffer or branch target buffer). It is a high speed buffer present in the IF stage used to hold the predicted target adrs.

- When the target adrs is available in the IF stage then stalls are not present, it is not practically possible to avoid all the control stalls.
- Partially suitable for loops.

Q consider pipeline w/o branch Penality. If target adrs not present in the IF stage.

Note → when the 2^{n} stating that Pipeline designed with branch Prediction, then assum that target adrs is present in the I^{st} stage of a pipeline otherwise (w/o branch prediction) assume that target adrs is not present in the I^{st} stage.

re-arrangement

I_1	I_1	I_1
I_2	I_3 (JMP 2000)	I_2
I_3 (JMP 2000) ⇒	I_2	I_3 (JMP 2000)
I_4	I_4	NOP → Becoz of NOP inserted after JMP, NOP overlap with I_3.
⋮	⋮	I_4
BI_1	BI_1	⋮
BI_2	BI_2	

compiler re-arrange code before execution

Note → to minimize the control dependency stalls & also preserve the correct execution path in the pipeline, one of a sw technique is used i.e. delayed branch.

★ Delayed Branch

- It is a compiler technique so, compiler rearranges the program to avoid the stall if possible or substitute the NOP Instn after the branch Instn to preserve the execution path if not possible to rearrange.
- In the compiler design this logic is programmed based on the pipeline structure to handle the control dependency.

User prog

1000 : I_1

```
1003 : I4
  :    :
2000 : BI1
2001 : BI2
  :    :
```

- Expected o/p:

$$I_1 - I_2 - I_3 - BI_1 - BI_2$$

- Actual o/p:

$$I_1 - I_2 - I_3 - \boxed{I_4} - BI_1 - BI_2$$

Unwanted.

→ with the help of rearrangement unwanted instr. is eliminated before entering

- "compiler rearrangement"

code:

Machine generated addrs space

```
; I1
  I3 (JMP BI1)
  I2
  I4
  :
  BI1
  BI2
  :
```

→ I_3 is jump it is either execute after I_2 or before I_2 & no change in o/p sequence. o/p of both applications is same.

PC: I_1	CC_1	CC_2	CC_3	CC_4	CC_5
I_1	IF PC: I_3	ID	EX	MA	WB
I_3		IF PC: I_2	ID uncod. toC PC: ~~IF~~ BI_1	EX	MA
I_2			IF PC: ~~I_4~~	ID (I_1)	EX
BI_1				IF PC: BI_2	ID
BI_2					IF PC: BI_3

Nop nothing but stall.

I_1

I_2

I_3 (Jmp BI_1)

NOP

I_4

!

!

BI_1

BI_2

!

PC: I_1	CC1	CC2	CC3	CC4	CC5	CC6
I_1	IF PC: I_2	ID	EX	MA	WB	
I_2		IF PC: I_3	ID	EX	MA	WB
I_3			IF PC: NOP	ID uncond Tdc PC: ~~IF~~	EX BI_1	MA
I_4				IF PC: ~~I4~~	ID	EX
I_5					IF PC: BI_2	ID
I_6						IF PC: BI_3

Actual o/p : $I_1 - I_2 - I_3 - (NOP) - BI_1 - BI_2$

↳ stall

INSTRUCTION SCHEDULING

- CPU always execute the program in a sequence called as inorder execution.
- In this execn sequence if any instru is dependent then the remaining Ins^n are also sharing the stall cycles, even they are independent.

Ex. I_1 : Add r0, r_1, r_2
I_2 : mul r_3, r0, r_4
I_3 : DIV r_4, r_5, r_6
I_4 : sub r_3, r_7, r8

In order execn sequence

$I_1 - I_2 - I_3 - I_4$
stalls

If in inorder sequence if I_1 is dependon I_2 then remaining Ins^n are also waiting.

- In the above code I_2 is data dependent on I_1 so I_2 will be waiting untill the I_1 execution is completed.
- These stalls are also shared by the I_3 & I_4 even they are independent.
- To handle the above impact Ins^n scheduling concept is closed. It states that insert the independent Ins^n IST. It causes out of order execution (reorder execution i.e $I_1 - I_3 - I_4 - I_2$).
- Re-order execution creates & two more dependencies in the pipeline. → out of order (Re-order)

(1) ANTI data dependency
(2) output " "

$I_1 - I_3 - I_4 - I_2$

problem.

- Anti data dependency will be occured when the instruction J tries to write the data before ins^n I reads it.

  Ex I_3 executing before I_2

  So I_3 updates the reg. r_4 before I_2 reads it

  ∴ I_2 incorrectly read the new value from the r_4 (data loss).

- Output data dependency will be occured in the pipeline when the ins^n J tries to writes the data before ins^n I writes it.

  Ex I_4 executed before I_2

  So I_4 updates the reg. r_8 before I_2 writes it. ∴ destina^n is incorrectly updated with a old value (data loss).

- To handle the above dependency problem h/w technique is used i.e Register Re-naming.

- This technique states that use the reorder buffers to store the out of order ins^n o/p, later update the register file with a reorder buffer ~~content~~ contain after the completion of a

dependent- · Ins^n execution.

Execu^n

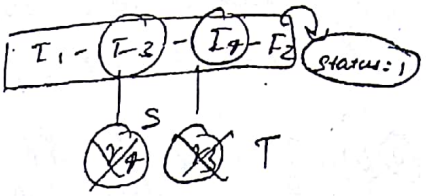
$I_1 \longrightarrow r_0$

I_3, I_4] → Re-order Buffer

$I_2 \longrightarrow r_3$ { status = 1 }

⇓

re-order Buffer → Reg. file { R_4, R_3 }



Q. Anti data dependency always creates the Hazards in the pipeline

Ans False

Hazards

- It is a delay, delay created in the pipeline due to a dependency problem.
- three kinds of hazards possible in the pipeline.

(1) structural Hazard
(2) Data "
(3) control "

- Data Hazard is further divided into three types based on the read & write o/p^n names as.

(1) RAW [Read-after-write] Hazard
(2) WAR [write-after-Read] Hazard
(3) WAW [write-after-write] Hazard.

① RAW Hazard is created when the instruction J tries to read the ~~data~~ data before Ins^n I writes it, (true data dependency).

② WAR Hazard is created when the Instr^n J tries to write the data before Ins^n I reads it (anti data dependency).

③ WAW Hazard is created when the Instr^n J tries to write the data before Ins^n I writes (out data dependency).

④ RAR is not a Hazard becoz RBR is not a problem.

adjacent verification — Ins^n 'J' — Ins^n 'I'

- In-Reg — out-Reg ; True data dependency
- out-Reg — In-Reg ; Anti " "
- out-Reg — out-Reg. ; o/p " "

Non adjacent verification.

★ Performance Analysis with stalls

$$S = \frac{\text{Avg. Ins}^n \text{ ET}_{\text{nonpipe}}}{\text{Avg. Ins}^n \text{ ET}_{\text{pipe}}}$$

$$S = \frac{CPI_{\text{nonpipe}} * \text{cycle time}_{\text{nonpipe}}}{CPI_{\text{pipe}} * \text{cycle time}_{\text{pipe}}}$$

$$IC * CPI * \text{cycle time}$$

$$1 * CPI *$$

$$= CPI$$

- Ideal CPI of pipeline is always 1 but due to dependency problem stalls are created so,

$$S = \frac{CPI_{nonpipe} * cycletime_{nonpipe}}{(1 + \#stalls/\Sigma ns^n) * cycletime_{pipe}} \quad (tn)$$

(tn) → instruction execution time in non pipe.

- When the pipeline stages are perfectly balance then one task execution time in non pipe is also equal to a no. of stages in the pipeline.

i.e $tn = k \text{ cycles} = k \cdot tp$

On this condition S:

$$S = \frac{k \cdot tp}{(1 + \#stalls/\Sigma ns^n)\, tp}$$

$$S = \frac{k}{(1 + \#stalls/\Sigma ns^n)}$$

when η is 100% stalls is zero.

- When the s/s is operating with 100% η (efficiency) then no stalls present then:

$$S = \frac{k}{(1+0)}$$

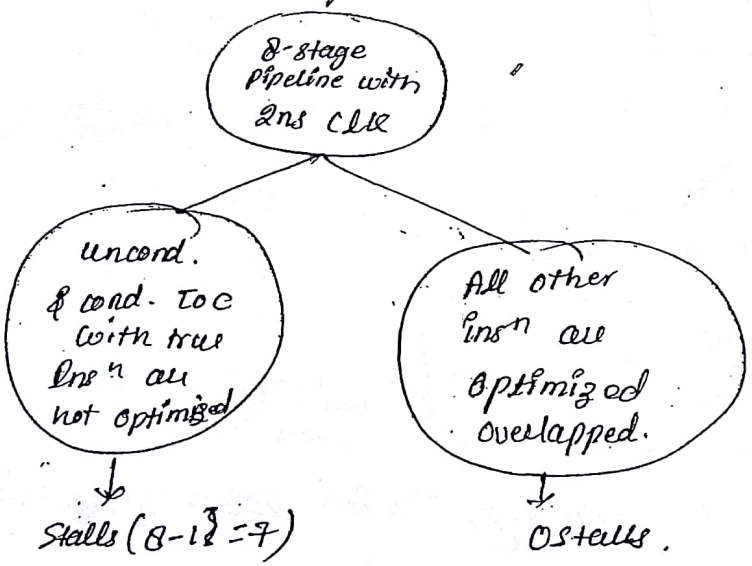
$$S = k$$

24/8/2017 Thursday class-11

Q) Consider 8 stage pipeline & allows all the ins except branch ins. Processor stops the fetching of sequential ins after the jump ins until the target add is available. In the pipeline all the inst are proceed thro' all the stages. Program contain 40% branch instr among them 30% are conditional in wch 60% of ins doesnot satisfy the condi, when the condi is false, then the foll. ins are overlap pipeline operated with a 2ns clk. calc the foll.

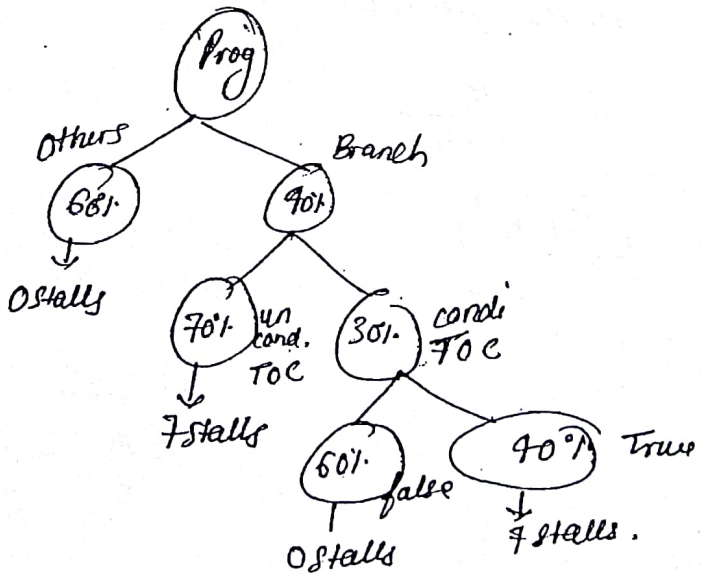
a) what is the avg ins ET

b) " " " performance gain.



2·27
1
3·27
×2
6 5 8

This is the way to solve control dependency prob.



stalls / Ins^n = $\begin{bmatrix} (0.6*0) + (0.4*0.7*7) \\ + (0.4*0.3*0.6*0) \\ + (0.4*0.3*0.4*7 \end{bmatrix}$ = 2.29

a) Avg. Ins^n ET pipe = (1 + # stalls / Ins^n) cycle time
= (1 + 2.29) 2ns
= 6.58ns

b) Speed up = $\frac{K}{(1 + \# \text{ stalls / Ins}^n)}$ → n is ∞ in this case that's why we use formula

= $\frac{8}{(1+2.29)}$ = 2.43

Note → when the Q^n states the Target adr^s availability in the pipeline as:

① stage no. given then | Branch penality = stage number − 1.

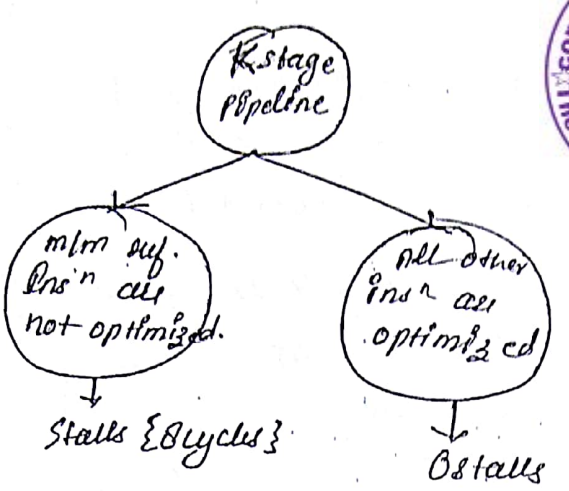
② stage name given then

Branch pending = corresponding stage no. w.r.to stage no. − 1

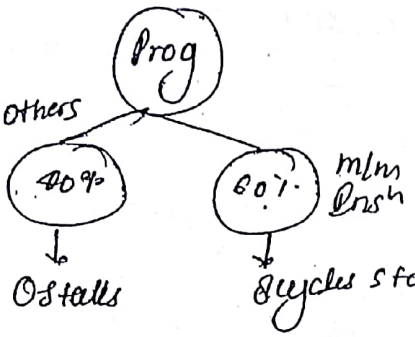
③ All ins^n proceed thro' all stages (or) until the ins^n is completed then.

Branch penality = last stage no. − 1

Q2) consider k-stage pipeline w/c allows all the ins^n except m/m reference ins^n. Penality of a m/m ins^n is 8 cycles. Prog. contain 60% m/m inst^n wt is the avg. ins^n exe^n time in the pipeline.



→ Cycle time cycle freq. not given so answer shud be in cycles.

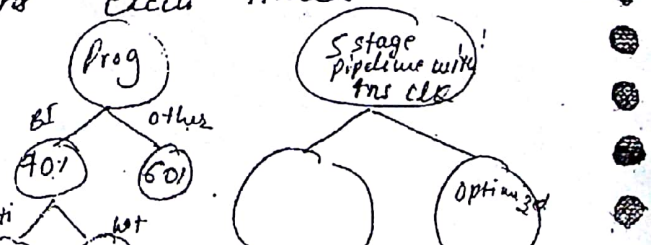


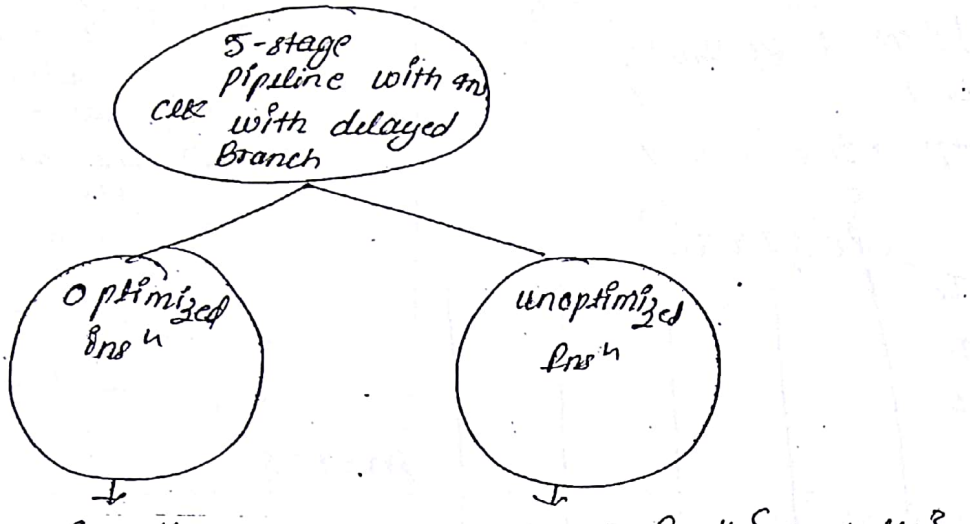
~~#~~ stalls / Ins'n = [(0.4 * 0) + (0.6 * 8)]
= 4.8

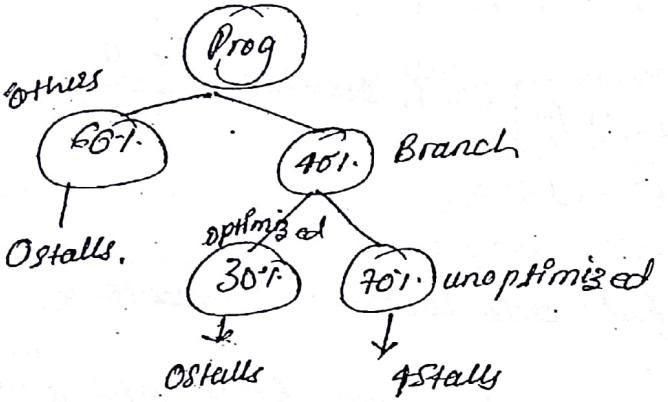
a) Avg. Ins'n Exec'n pipe = (1 + ~~#~~ stalls / Ins'n) cycle time
= (1 + 4.8)
= 5.8 cycles.

Q3) In the 5 stage pipeline, stages are balanced with a 4ns clk. In the pipeline & control penalities optimize with a delayed branch technique. This technique substitute the 4 NOP of Ins'n after the Branch Ins'n if not possible to optimized. Prog. contain 40% Branch Ins'n among them 30% are optimized, What is the average ins'n execution time.

~~#~~ Stall / Ins'n = {(0.4 * 0.3 * 0)







~~#~~ Stalls / Ins^n = [(0.4 * 0.7 * 4)]

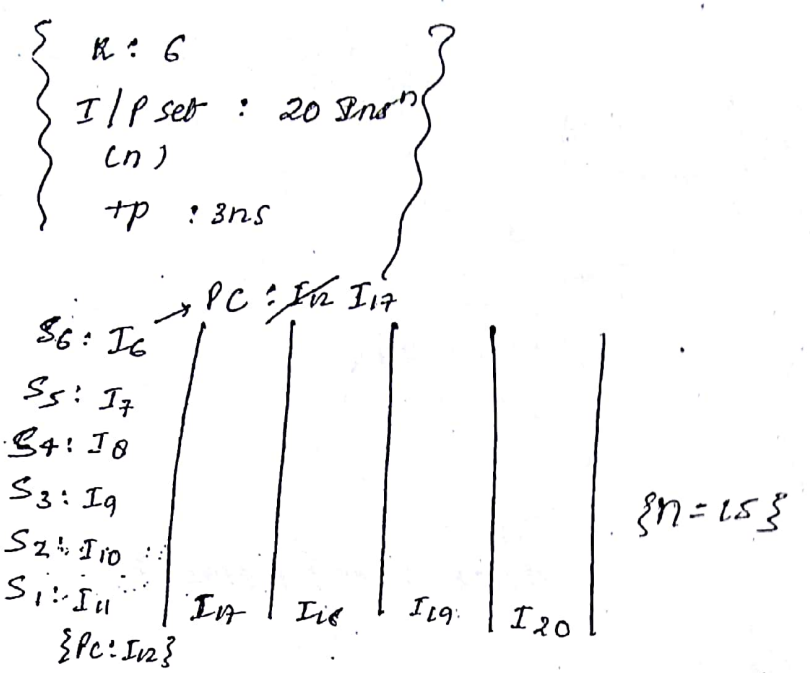
= 1.12

a) Avg. Ins^n ET pipe. = (1 + # stalls / Ins^n) cycle time / when n is unlimited use formula

= (1 + 1.12) * 4ns

= 8.48ns

Q) consider 6 stage pipeline with a cycle time 3ns used to execute the prog. code w/c contain 20 Ins^n (I_1 to I_{20}), I_6 Ins^n is a ~~to~~ uncondi^nl jump w/c transfers the control to I_{17} Ins^n. In the pipeline target add^rs is available when the Ins^n is completed. so what is the program execution time:



When I6 is completed previous 5 are overlaped. I6 exit from pipe PC updated due to jump some of the instr never execute w/c ours we have delete.

$$ET_{pipe} = (k+n-1)\,tp$$ → Here n is limited. n = 20 than n = 15

$$= (6+15-1)\,3ns$$

$$= 60ns.$$

Qs consider the foll. code used to execute on a pipeline processor.

I_1 : $r_0 \leftarrow r_1 + r_2$
I_2 : $r_2 \leftarrow r_0 + r_1$
I_3 : $r_0 \leftarrow r_2 / r_2$
I_4 : $r_2 \leftarrow r_2 - r_3$
I_5 : $r_3 \leftarrow r_2 + r_3$
I_6 : $M[x] \leftarrow r_3$

How many RAW, WAR & WAW Hazards are possible in the pipeline. How many dependency occured).

True data dependency (adjacent verification)

[in − out]

$I_2 - I_1 [r_0]$
$I_3 - I_2 [r_2]$
$I_5 - I_4 [r_2]$
$I_6 - I_5 [r_3]$

4

Anti data Dep -

[out - In]

$I_2 - I_1 [r_2]$
$I_3 - I_2 [r_0]$
$I_4 - I_1 [r_2]$
$I_4 - I_3 [r_2]$
$I_5 - I_4 [r_3]$

5

out/p data dep.

[out - out]

$I_3 - I_1 [r_0]$
$I_4 - I_2 [r_2]$

2

NON LINEAR PIPELINE

it is having forward & Backward direct

- This pipeline contain both forward & Backward ~~di~~ connection so Reservation table is used to execute the i/p in the ~~pipeline~~ pipeline.
- let us consider the sample reservation table used to execute the i/p on a hypothetical pipeline.

i/p is → S_1
o/p is → S_1

multiple check mark in row indicate

	1	2	3	4	5
S_1	X				X
S_2		X		X	
S_3			X		

- Multiple check mark in the row indicate repeated use of a same stage in different cycles.
- contigious check mark in the row indicate extended use of a same stage ~~in~~ ~~sam~~ upto some no. of cycles
- Multiple check marks in the column indicate ll use of a different stages, in the same cycle.

- **Latency Analysis**
  - Latency means clock cycle difference b/w the two successive initiation in the pipeline.
  - In the linear pipeline latency is always one.
  - In the non linear pipeline latency value depends on the reservation table.
  - In the non linear pipeline some latencies causes the collisions called as forbidden latency or (non permissable latency) & some of them doesn't cause the collision called as non forbidden latency (Permissable latency).
  - To detect the permissable latency we need to take the diff (clk diff) b/w any two check marks in the row.

latency 4 means start inserting from 5th cycle.

row1: (5-1) = 4 → we are inserting i/p to 5th cycle.

row2: (4-2) = 2 latency 2 & 4 is collision

row3: ___ 1, 3, 5 no collision.

  - Based on the forbidden latency collision vector is genrated.

Ex: $C_v : [C_n \; C_{n-1} \ldots C_2 \; C_1]$

$C \rightarrow$ 0 Permissable
$C \rightarrow$ 1 non-Permissable.

collision vector : $[C_5 \; C_4 \; C_3 \; C_2 \; C_1]$
0 1 0 1 0

if i take latency 3 is it always latency 3 possible.

① latency sequence w.r.t "C_1"

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_1	1	2			1	2	3	4			3	4	5		
S_2		1	2	1	2			3	4	3	4			5	
S_3			1	2					3	4					5

latency cycle

diff. b/w two successive initiative blw clk cycle.

(2-1) = 1

(7-2) = 5

(8-4) = 1

(13-8) = 5

⋮

(1,5) (1,5) (1,5) . . .

Avg. latency = $\left(\frac{1+5}{2}\right)$ = (3)

② latency seq. w.r.t "C_3"

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S_1	1			2	1		3	2		4	3			4
S_2		1		1	2		2	3	....	3	4		4	
S_3			1			2			3			4		

latency cycle

(4-1) = 3

(7-4) = 3

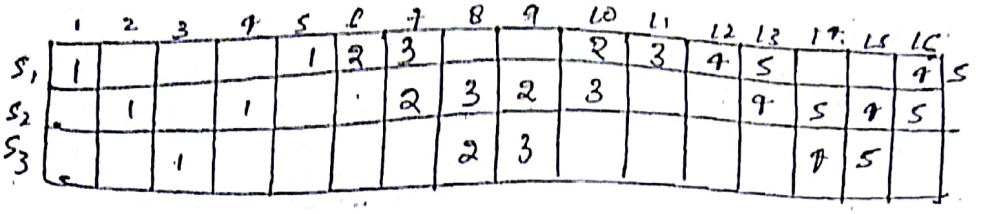
(10-7) = 3

⋮

3, 3, 3, 3 . . . . .

∴ Avg latency = (3)

(3) Latency sequence w.r.t "S1"



Latency cycle:

$(6-1) = 5$

$(7-6) = 1$

$(12-7) = 5$

$(13-12) = 1$

(5,1) (5,1) (5,1) (5,1) - - - -

Avg. latency $= \frac{(5+1)}{2} = 3$

MAL = 3

[Min avg. latency]

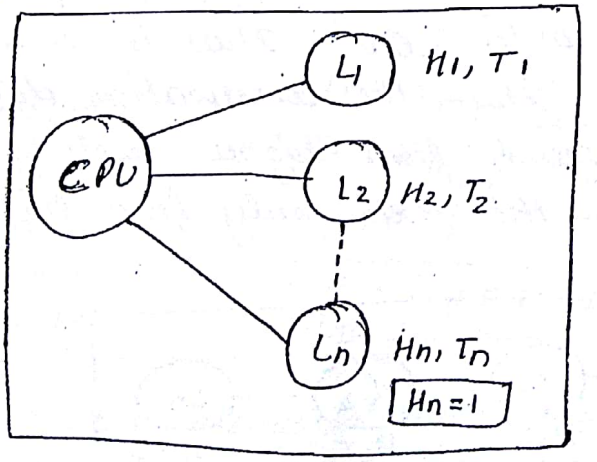
MEMORY ORGANISATION

- Based on the style of accessing the m/s support memories
- M.O is of two kinds.

(1) simultaneous Access m/m
(2) hierarchial Access m/m

(1) simultaneous Access m/m organization.

- In this m/m design CPU directly connected to all the levels of memories. But accessing the m/mies in a sequence i.e when there is a miss on level one (L1) m/m CPU directly access the data from L_2 m/m w/o copying into (L_1).
- When there is a miss on a (L_2) m/m then CPU directly access the data from level 3 (L_3) m/m w/o copying the data into a (L_2) & (L_1) & so on.
- In this m/m design data block is neednot be required in the (L_1) m/m



data present in L_1 hit
data not present in L_1 miss.

$H + M = 1$

$M = (1-H)$

Here,

- **Hit ratio (H) = $\frac{\#\text{ hits}}{\text{Total } \#\text{ Accesses}}$**
- $H : \{0 \text{ to } 1\}$
- $T_1, T_2, \ldots T_n \rightarrow$ Access times of a respective m/m.
- Time required to access one word data from the m/m is called as average access time i.e.

$H_1 T_1$ — Hit in 1st level access data from 1st level.

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

↓ avg. access tym of the m/m.

- 1 word — T_{avg}
  2. # words — 1 sec

$$\eta_{mem} = \frac{1}{T_{avg}} \text{ words/sec}$$

24/8/2017 (Tu)
Class - 11
CO (Remaining)

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

This is not practically use.

$P \propto \frac{1}{AT}$ (Access time) → m/m latency (time req. to prepare the data)

Simul. Accessing time is low.

$T_{avg}(simul) < T_{avg}(hier.)$

$P_S > P_H$

Hier. Access time is high.

Code

I_1 : Data (L_2-mem) ——
I_2 :
I_3 : data
I_4
I_5 : data
I_6
I_7 : data
I_8
I_9 : data
I_{10}

} Re-use the "I_1" data.

Execn

Simultaneous

I_1 : (ML_1, T_2) miss in L_1, access from T_2
I_3 : ML_1, T_2
I_5 : ML_1, T_2
I_7 : ML_1, T_2
I_9 : ML_1, T_2

Total access time : $5 * T_2$

Hierarchial

I_1 : ML_1, $(T_2 + T_1)$
I_3 : T_1
I_5 : T_1
I_7 : T_1
I_9 : T_1

} Re-usable data space } locality of reference

Total access time : $(1 * T_2) + (5 * T_1)$

in this case Hier. is better than simul.

Note → In the hierarchial m/m design avg. access time of the m/m is minimized beco of locality of the reference is present.

Locality of reference means CPU performance of

- It is two kinds.

a) Temporal locality: same word in the data block is frequently refered by a CPU in a near future.

b) ~~Special~~ locality: - means adjacent word in the spatial. same block will be refered by a CPU in a sequence.

- To satisfy the above principle we need to transfer the data from level 2 to level 1 mlm in a block wise.
- Block contain group of words.

Note → when the Q^n contain hierarchy word or hierarchy meaning or cache mlm word then use hierarchal organization, otherwise use simultaneous organization.

Q In a 2 level mlm organization memory is 10 times faster than L2 mlm & its access time is 40ns less than the avg. access time let level L1 mlm access time is = 30ns, what is the hit ratio.

- simultaneous mlm org.
- $L_1 \rightarrow H_1, T_1$

  $L_2 \rightarrow H_2, T_2$

  $\{H_2 = 1\}$

$$S = \frac{T_2}{T_1}$$

$$10 = \frac{T_2}{T_1}$$

$$T_2 = 10T_1$$

$$T_1 = T_{avg} - 40ns$$

$$\boxed{\quad}$$

let,

$T_1 = 30ns$

then,

$T_2 = 10 * 30$

$= 300ns$

$T_{avg} = 30 + 40$

$= 70ns$

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

$70 = (H_1 * 30) + (1 - H_1) 1 * 300$

$H_1 = 0.85$

Q2 consider 2 level m/m organization, L_1 m/m is a cache m/m w/c is reference by the CPU 80% of time to accessed by ~~the~~ the data. Access time of cache m/m is 50ns & level 2 m/m access time is 200ns what is the average access time.

Hierarchial m/m organization.

- $L_1 \rightarrow H_1, T_1 \{80\%, 50ns\}$
- $L_2 \rightarrow H_2, T_2 \{1, 200ns\}$
- $T_{avg} = ?$
- $T_{avg} = H_1 T_1 + (1 - H_1) H_2 (T_2 + T_1)$

$= (0.8 * 50) + (1 - 0.8) 1 (200 + 50)$

$= 90ns.$

Q3 3 level m/m org? has the following specification,

level	Access Time word	Block size (in words)	Hit ratio	Block access time
1	20ns	–	0.7	$T_1 = 20ns$
2	100ns	2	0.9	$T_2 = 200ns$
3	200ns	4	1	$T_3 = 800ns$

If the refered block not in L_1 then transfer the data from L_2 to L_1 if not in L_2 then transfer

• Hierarchial m/m org?

- 1st decide access time per block. From L_1 CPU always access data in words.

$$T_{avg} = H_1T_1 + (1-H_1)H_2(T_2+T_1) + (1-H_1)(1-H_2)H_3(T_3+T_2+T_1)$$

$$= 0.7 \times 20 + (1-0.7) \times 0.9 (100+20) + (1-0.7)(1-0.9)1(200+100+20)$$

$$= 1.4 + (0.3) \times 0.9 (120) + (0.3)(0.1)(320)$$

$$= 104ns$$

Q) In a two level m/m sys, level 2 m/m bandwidth is 1GBPS data will be transferred from L_2 to L_1 in a block wise. Block size is 128 bytes. Access time of L_2 m/m is 40 ns. What is the total time required to transfer the data block from L_2 to L_1.

Soln

Total time = Access time + Transfer time

↓ 40ns

↓ depends on the Bandwidth {data transfer rate}

↓

1GB ——— 1sec

128B ——— ?

↓

$\frac{128B}{1GB}$ sec

128ns.

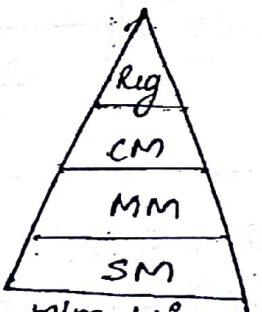
$= 40 + 128$

$= 168ns.$

★ Memory hierarchy design

- In the computer s/s design hierarchial m/m organization is used to design the computer m/m.
- Acc. to a m/m hierarchy, s/s supported m/m standards is as follows.

Level	1	2	3	4
Name	Register	cache m/m (CM)	main m/m (MM)	secondary m/m (SM)
Typical size	<1kB	<16MB	<16GB	>100GB
Implementation	customized multiports	SRAM (flip-flop)	DRAM (capacitor)	Magnetic
Access time (ns)	(0.25-0.5) At low	(0.5-25)	(80-250)	50,00,000
Band width (MB/sec)	(20,000-100,000) Performance high.	(5000-1000)	(1000-5000)	(20-150)
Managed by	compiler	H/W	OS	OS
Backed by	CM	MM	SM	compact Disc (CD)



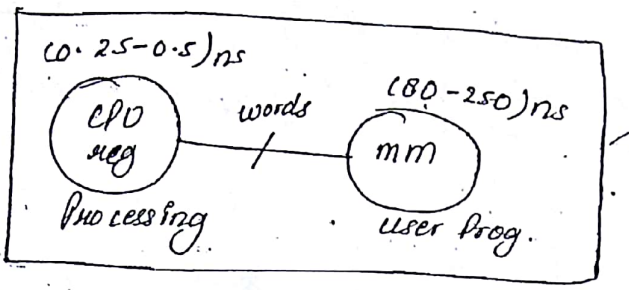
m/m Hierarchy

Bottom to top size is in ↓ order.

- Bottom - top - approach
  1. capacity ↓
  2. Access time ↓

- with influence to a mlm hierarchy design access sequence of a d/s supported mlm is as follows.

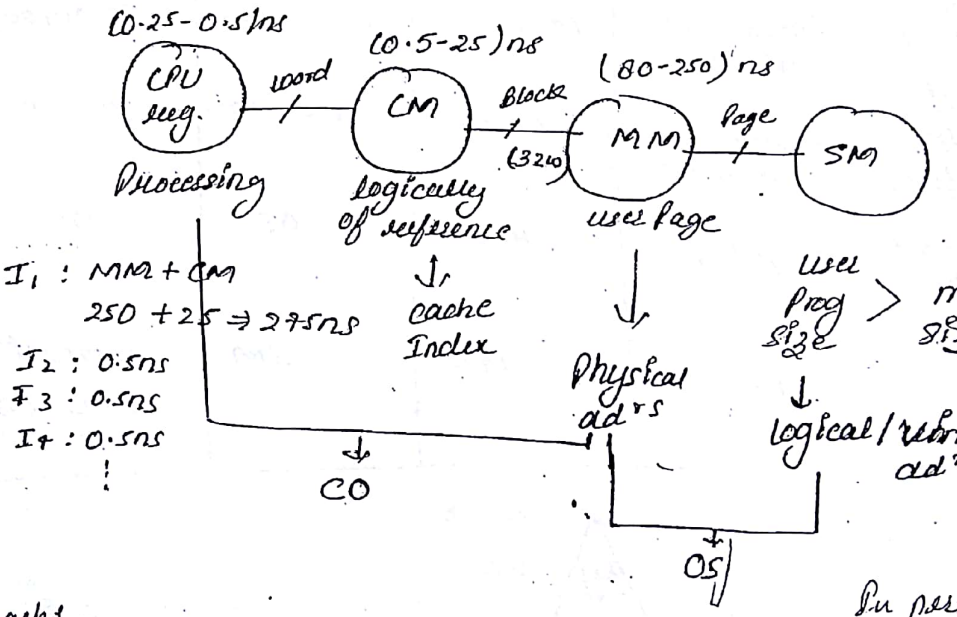
System old



CPU attached to mlm to access word by word adq.
if not present take worst case if present take best case.

Prog. size = 32 Rns
Rns size = word size
{32 MR} to access the prog.

System New



In personal comp. CPU always generate logical adrs
In OS, CPU generate logical adrs
In CO, CPU generate physical adrs

C → cache
S → secondary mlm
M → main mlm

$$T_{avg} = H_c T_c + (1-H_c) H_m (T_m + T_c) + (1-H_c)(1-H_m) H_s (T_s + T_m + T_c)$$

$$\downarrow$$

$$H_s = 1$$

consider the foll. mem specification.

mem	Access time	Hit Ratio
I-cache	10ns	0.7
D-cache	15ns	0.8
MM	100ns	0.9
SM	200ns	1

In the prog. of ins request are generated for Ins^n access & 40% of request are generated for data access. what is the total access time of mem time.

1st cal. 1 ins^t A.T , 1 ins^t D.T

Tavg Ins^n = HcTc + (1-Hc) Hm (Tm + Tc) + (1-Hc)(1-Hm) Hs (Ts + Tm + Tc)

= (0.7 * 10) + (1-0.7) 0.9 (100 + 10) + (1-0.7) (1-0.9) 1 (200 + 100 + 10)

= 46 ns

Tavg data = HcTc + (1-Hc) Hm (Tm + Tc) + (1-Hc)(1-Hm) Hs (Ts + Tm + Tc)

= (0.8 * 15) + (1-0.8) 0.9 (100 + 15) + (1-0.8)(1-0.9) 1 (200 + 100 + 15)

= 39 ns

Total time req. for prog. to access Ins^n & data.

= $\left(f_{Ins^n \text{ req.}} * Tavg_{Ins^n}\right) + \left(f_{data \text{ req}} * Tavg_{data}\right)$

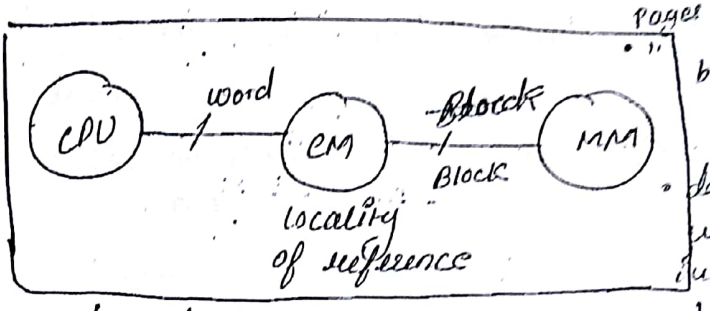
= (0.6 * 46) + (0.4 * 39)

= 43.2 ns.

25/01/2017
Friday
class-12.

cache memory

cache m/m access a intermediate b/w the CPU & main m/m. used to hold the image of a main m/m data. ∴ CPU will be accessing the main m/m data from cache m/m with in a less amt of time ∴ speed gap is synchronized b/w the CPU & main m/m.



- cells are grouped into pages
- " " " blocks.
- data move from MM to CM in the form of block.

Design elements

1. m/m org"
2. mapping techniques
3. Replacement algorithm
4. of updating techniques
5. multilevel cache design.

(1) memory org"

Acc. to a m/m hierarchy design data will be transfer from main m/m to cache m/m in a form of block. so both of the m/m are organized into a block based on the block size.

$$\#\text{ Blocks in mm} = \frac{\text{m/m size}}{\text{Block size}}$$

⇒ consider 8B cache with 2B Block.

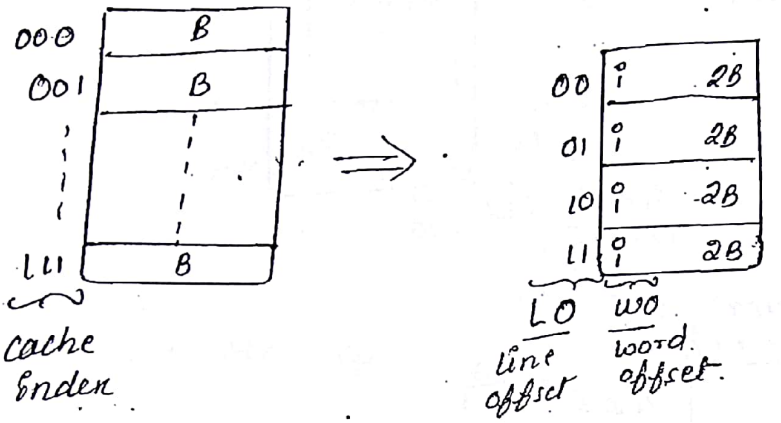
Before org^n

8B → CM

After org^n

- # lines = $\frac{8B}{2B}$ ⇒ 4
- 4 * 2B ⇒ 8B

capacity same internal structure is different.



cache index
← 3bit →

line offset (LO)	word offset (WO)

LO → line No. in the CM → $\log_2$ # lines → $\log_2 4$ → 2bit

WO → word add's in the block → $\log_2$ Block size → $\log_2 2$ → 1bit

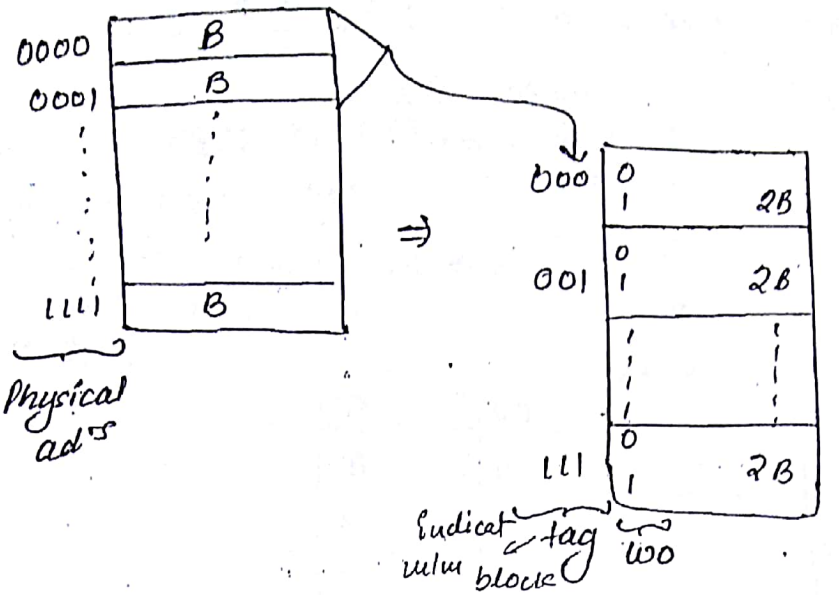
⇒ consider 16B MM with 2B block

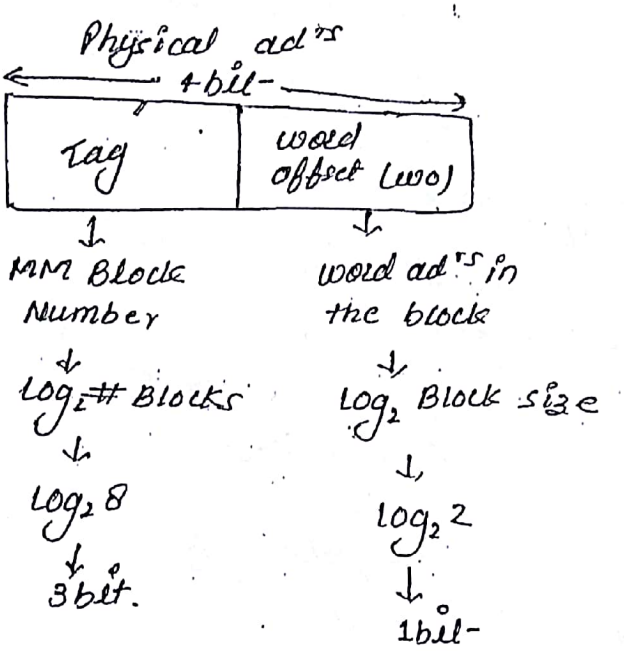
Before org^n

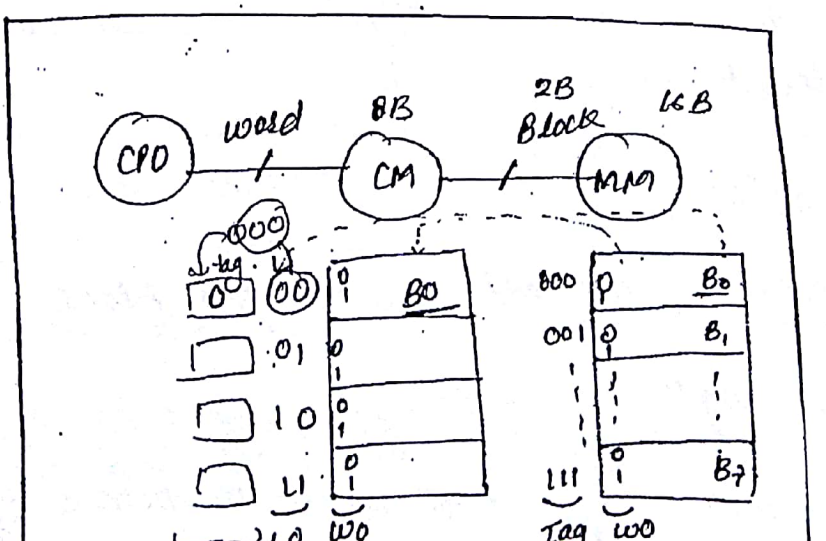
16B → MM

After org^n

- # Blocks in mm = $\frac{16B}{2B}$







Extra bit are stored in tag directory.

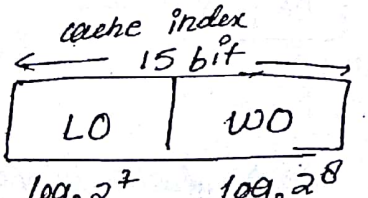
Q Consider a 64 bit hypo. Proc^r w/c supports 32KB cache mm organized into a 32 word block. cache m/m data is a subset of a 2^{32} m/m space. How many lines & how many blocks are possible in the respective m/m. & show the add^rs formats.

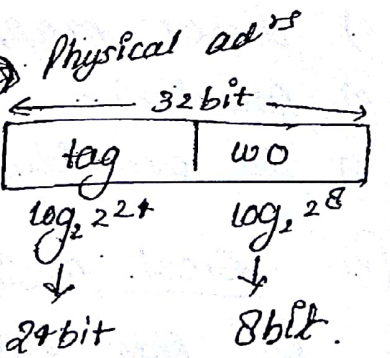
Given
- word length = 64 bit = 8B
- CM size = 32KB
- Block size = 32 words = 32 x 8B = $2^5 \times 2^3$ B = 2^8 B
- MM size = 2^{32} B

cache data is always a subset of mm data

$$\#\text{Lines}_{CM} = \frac{CM\ size}{Block\ size} = \frac{32K}{2^8} = \frac{2^{15}}{2^8} = 2^7$$

$$\#\text{Blocks in mm} = \frac{MM\ size}{Block\ size} = \frac{2^{32}}{2^8} = 2^{24}$$





② Mapping techniques

- The process of transfering the data from main m/m to cache m/m is called as mapping.
- During the mapping process data block is transfer to ~~cache along with~~ a Physical add^{rs}.

In this process, some extra space is maintained in the cache line used to accomodate the Physical add^{rs} info^{n} called as ~~tag~~ tag directory.

Tag directory size in the cache controller is

> tag memory size = # lines in CM * # tag bits in the line.

> data m/m size = # lines in CM * Block size

> Total cache = Tag m/m size + data m/m size.

Whenever cache refered by default data cache.

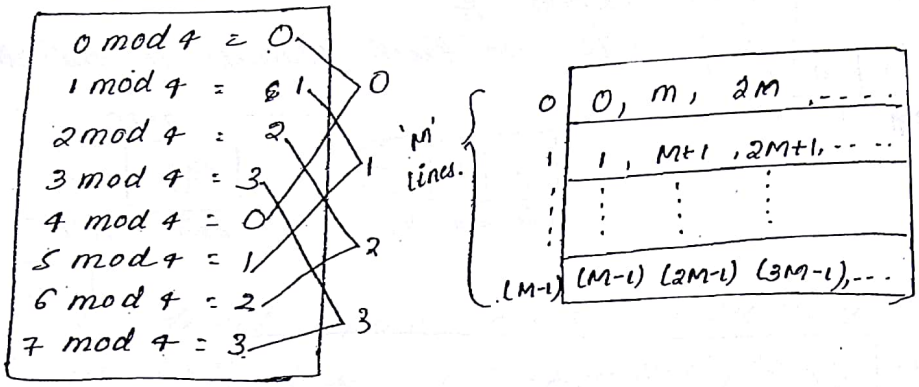
- In the cache design three kinds of mapping fun^{n} is used to ~~map the~~ data.

1) direct mapping
2) Associative "
3) Set associative "

① direct mapping

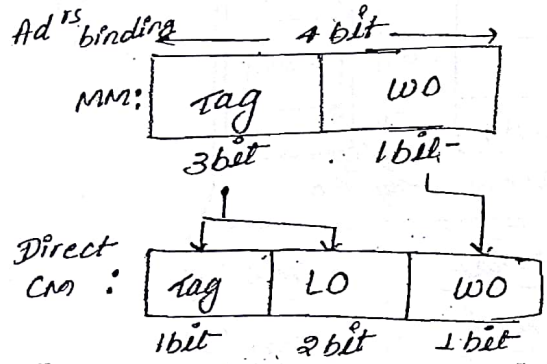
In this technique mod fun^{n} is used to map the data i.e $K \bmod N = i$

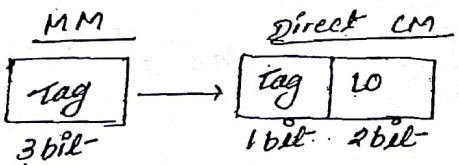
- K → MM Block
- N → # lines in CM
- i → CM line



- Mapping fun shows the relationship b/w the main m/m block & cache m/m line ∴ adrs binding

Ex



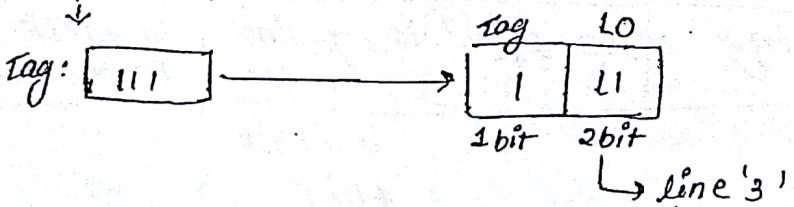


B0[000] → 0/1 | 00 → line '0'

B4 [100]

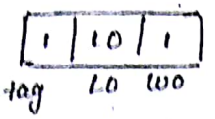
Ex

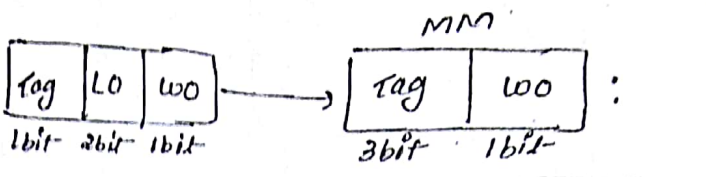
B_7 mapped into which line

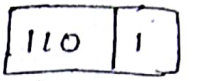


Ex: direct cache format is

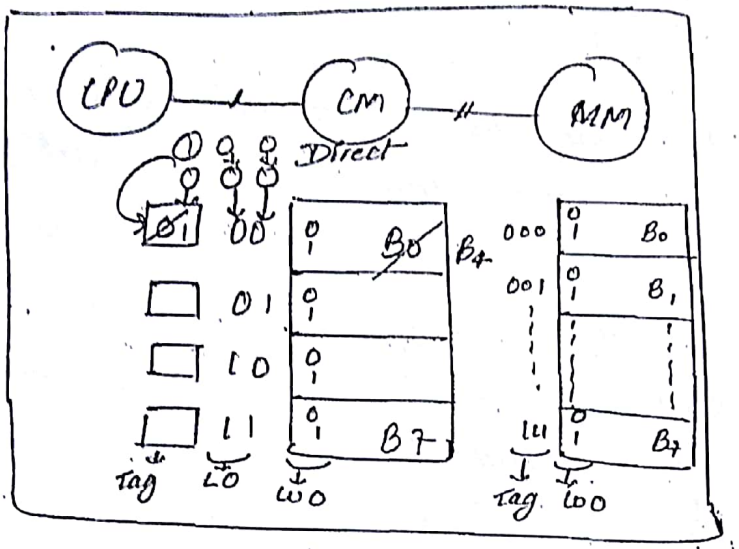
w/c MM Block presents in the line







→ "B_6"



MM Block	Direct Map	CM Line
B0 [000]	k mod N = i → 0 mod 4 = 0	line '0'
B7 [111]	k mod N = i → 7 mod 4 = 3	line '3'
B4 [100]	k mod N = i → 4 mod 4 = 0	line '0'

- During the mapping process some of the tag bits are connected to a line offset & rest of the bits are stored in the cache controller as a tag.

tag m/m size is = # lines in CM * # tag bits in the line

• Total cache = Tag mlm size + data mlm size

= 4bits + 8B

= 4bits + (8 * 8bits)

= 68bits.

• In the direct cache design explicit replacement technique (FIFO) are not required to replace the data becoz every main mlm block is having fixed locaⁿ in the cache mlm.

Accessing sequence

mlc Insⁿ : mov r0, [1001]

↓

CPU generates the mlm request

1001	R0

↓

direct cm

↓

add's format

Tag	LO	WO
1	2	1

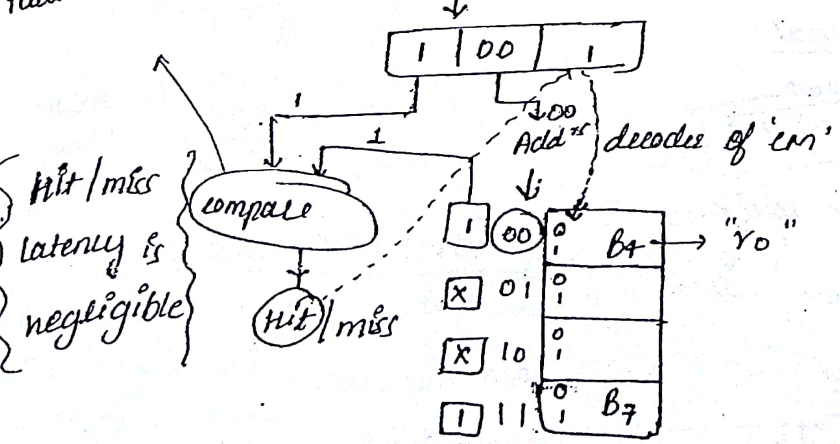
↓

Hit & miss latency is always zero, becoz of h/w design

avg. accessing time ↑

performance decreases.

If in compare both are 1 then it is hit.



Q consider a 32 KB direct mapped cache organized into a 32 Byte Blocks. Main mlm data w/c is addrs with a $(CBCDE)_{16}$ is mapped into a cache m/m.

a) to w/c line the respective main mlm block is mapped.

b) what is the tag mlm size in the cache controller when each line also main 1 valid bit & 1 update bit.

15 bit.

Sol^n tag not given. main mlm adrs given.

15 bit

0 X CB CDE

Hexa decimal Notan.

Given

CM size = 32KB

Block " = 32B

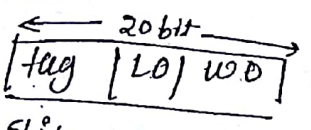
MM adrs = $(CBCDE)_H$

.... = 20bit

direct mapped.

Sol^n # lines = $\frac{32K}{32}$ = 1K no. of lines in cache.

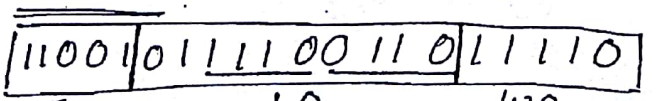
adrs format



5bit $\log_2 1K$ $\log_2 32 = 5$

10bit 5bit

CBCDE:



LO
↳ Line NO.
↳ $(1EG)_H$
↳ $(486)_{LO}$

Tag mlm size = # lines in CM * # tag bits in the line

= 1K * (5bit tag + 1valid bit + 1update bit)

= 1K * 7 bits

= 7K bits.

Q Consider the foll. direct cache add^{rs} design wlc indicates the data infon in the cache mlm.

Tag	LO	WO
101001	101011	010101

wlc main mlm block is mapped in the above cache line. What is the main mlm size in bytes. Also cal the tag space in the cache controller.

Direct CM add^{rs} format

101001	101011	010101
Tag	LO	WO

mlm add^{rs}: Tag (12bit) | WO (6bit)

101001101011	010101
tag	WO

↳ MM Block NO.

↳ $(A6B)_H$ → (2667)

101001101011	010101
A 6 B	

↳ mlm block no.

MM size = 2^{16} cells

$\downarrow$

$2^8 \times 2^{10}$ cells

$\downarrow$

256 K cells

$\downarrow$

256 KB

Tag m/m size = # lines in CM * # tag bits in the line

$= 2^6 *$ 6 bits

$= 384$ bits.

Note limitation in the direct cache is:

a) cache line is capable to hold only one block [& tag] at a time so no. of conflict missing is ↑.

b) when the CPU frequently refers the multiple blocks w/c are mapped into a same cache line then the cache block are continously ~~swapping~~ so mapping so hit ratio will be falling down. This phenomena is called as thrashing.

Thrashing

In CM design w/o map log: now can we map data. CM: 4 lines (initially empty)

MM Block: B0 B4 B0 B4 - - - thrashing occurs because of B0 & B4 both are sitting the same line.

Direct CM

00	B0 B4	B0
01		B1
10		B2
11		B3

B0 → M → 0 mod 4 = 0 → compulsory miss 1 at a time.

B4 → M → 4 mod 4 = 0 → compulsory miss time. B0 B4 occupy line hold at same place

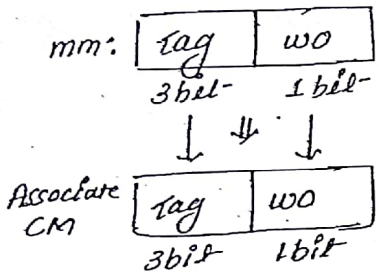
B0 → M → 0 mod 4 = 0 → conflict miss

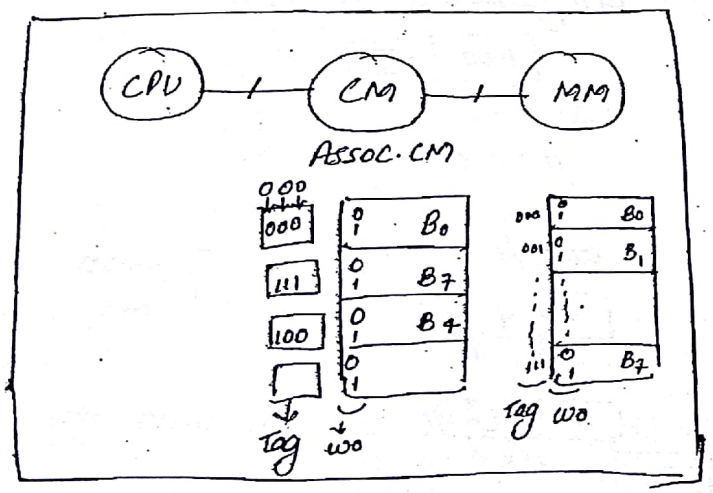
B4 → M → 4 mod 4 = 0 → conflict miss because of mod function

H = 0

- Thrashing Problem
- to handle the thrashing problem alternative cache is require i.e associative cache mm.
- Associative cache is designed w/o adrs called as content addressable mm.
- In this design sequence fun is used to map the data .'. any main mm block can be mapped into a any cache mm line in a sequence adrs binding fs.

Associative CM





MM Block Associative map CM line

B0 [000] — sequence → Any line in a sequence

B7 [111] — sequence → Any line in a sequence

B4 [100] — Sequence → Any line in a sequence.

- Tag m/m size = # lines in CM * # tag bits in the line.
  = 4 * 3 bit
  = 12 bits.

- Explicit replacement policies [FIFO/LRU] required to replace the data when the cache is full.

Accessing sequence:

m/c insn : mov r0, [1000]

↓

CPU generates the m/m req.

| 1000 | $\overline{RD}$ |

↓

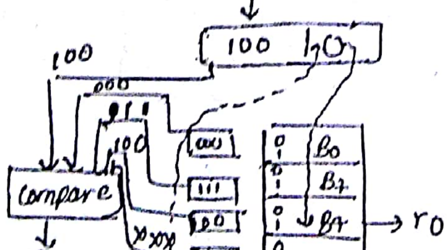
ASSOC CM

↓

add's format

tag	w0
3	1

↓



Note → limitation of associative cache is:
All the tags controller compare with a CPU generated tag in parallel so implementation of a comparision ckt become very complex & expensive, so search time is included in the avg. access time. ↑ ∴ for alternative required i.e set associative cache.

Q consider 64 KB fully associative cache organize into a 32 byte block mm contain 16 EM blocks. what is the tag m/m size in the cache controller.

Given
- CM size = 64 KB
- Block ll = 32B

Blocks in mm = 16 M
= 2^{24}

MM size = $2^{24} * 32B$
= 2^{29} Bytes

MM: | tag | wo |

$\log_2 2^{24} = 24$ bit $\log_2 32 = 5$ bit

assoc. CM: | tag (24 bit) | wo (5 bit) |

Tag m/m size = # lines in CM * # tag bits in the line

$\frac{64K}{32} * 24$ bits

3) Set associative cache

- This cache is used to compromise the disadvan. in direct & associative cache designs.
- In this design lines are grouped into a sets to accomodate more than 1 block in the set at a time.
- **No. of sets (S) = $\frac{N}{P\text{-way}}$**

  N : # lines in CM

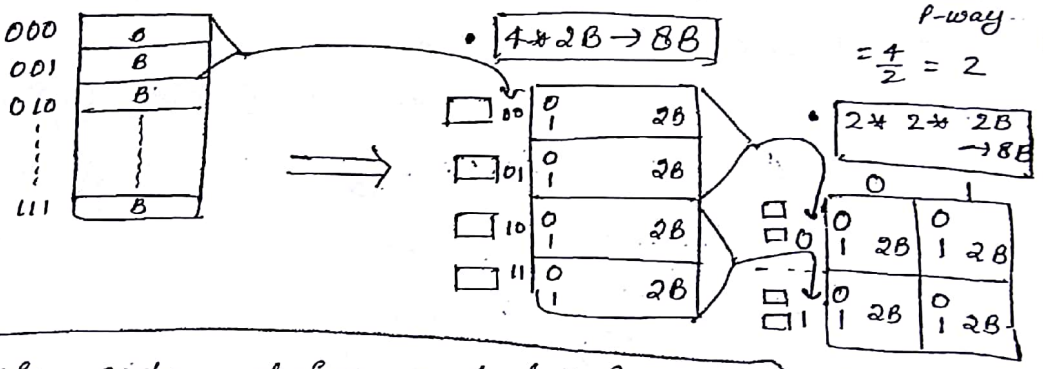
  P : # " " the set

- Before organization → After organized into a lines. → After organized into a sets

  8B → CM

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

  # sets (S) = $\frac{N}{P\text{-way}} = \frac{4}{2} = 2$



- In this cache design mod fun is used to map the data i.e **K Mod S = i**

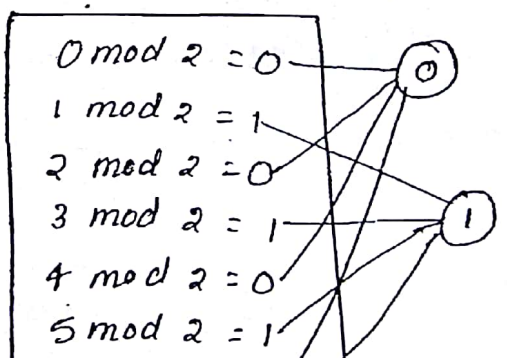
  K : MM Block No.

  S : # sets in CM

  i : CM set no.

  S : rows

  P : cols



Add^{rs} Binding –

MM: | tag (3bit) | wo (1bit) |

Set assoc CM: | Tag (2bit) | set (log$_2$#sets) | offset (wo) (1bit) |

$\log_2 2$

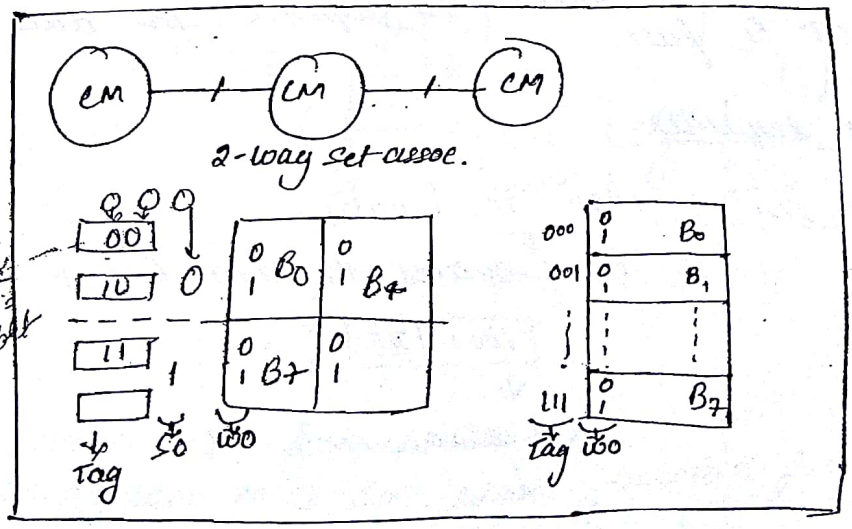
1 bit

MM ⟶ Set Associ. CM

tag (3bit) - - - ⟶ Tag (2bit) | so (1bit)

B_0 [000]
B_2 [010]
B_4 [100]
B_6 [110]

⟶ 00/01 · 10/11 | 0 ⟶ set '0'



MM Block	set Associ. MAP	CM set-
B0 [000]	k mod s = i / 0 Mod 2 = 0	set '0'
B7 [111]	k mod s = i / 7 mod 2 = 1	set '1'
(B4) [100]	k mod s = i / 4 mod 2 = 0	set 0

direct cache + associat. cache ll set associa. MAP.

Set is full in cache will replacement required by explicit.

- During the mapping some of tag bits are connected to a set o/p ⁿ least of the tag bits are store in a cache controller.

tag mlm size = # set in CM * # lines in set * # tag bits in line

= S * P * # tag bits

= 2 * 2 * 2 bits

= 8.

Explicit replacement techniques (FIFO / LRU) used in this cache to replace the data when the set is full.

Access sequence

mlc Ins ⁿ : MOV r0, [1001]

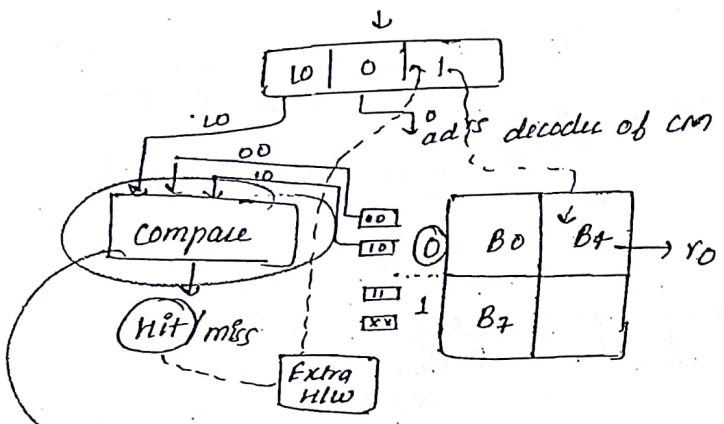
CPU generates the m/m Request.

1001	$\overline{RD}$

↓

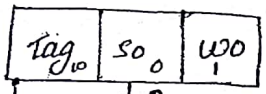
set assoc. CM

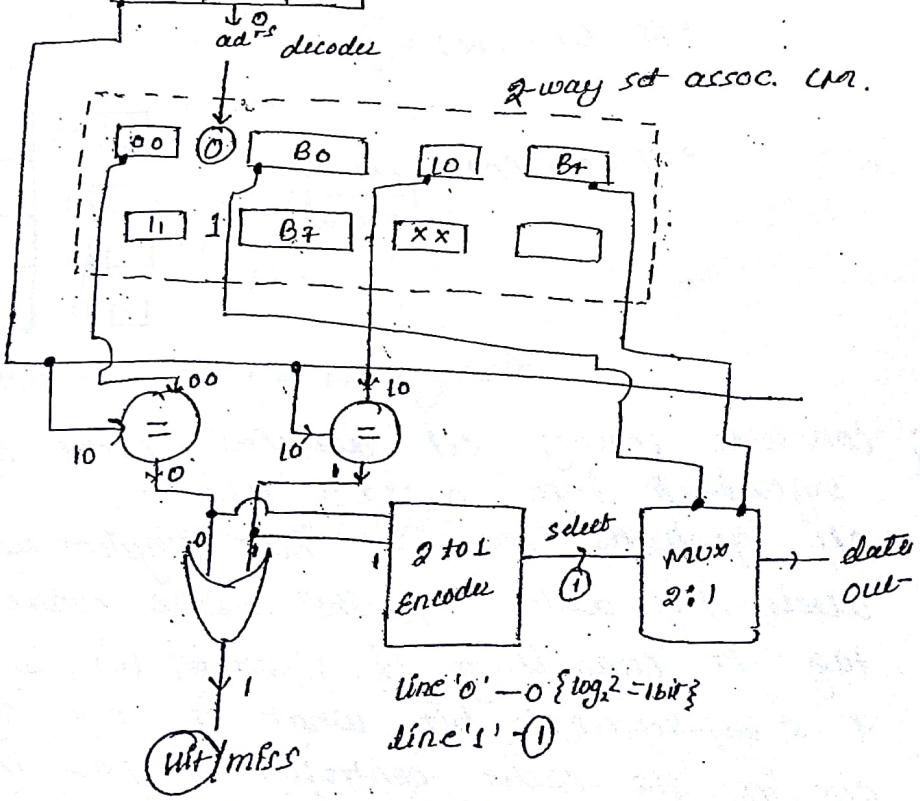
↓



• Comparison Ckt

Set associative adrs format

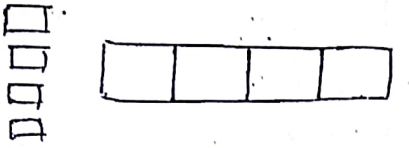




Note: when P is approach to N then S is approach to 1 & Cache become full associative.

4-way set assoc:
cache of 8B with 2B Block.

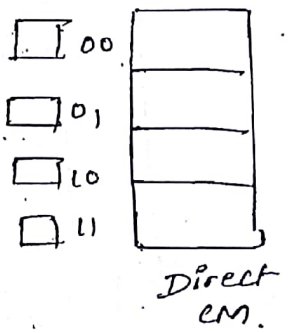
- # lines (N) = $\frac{8}{2} = 4$
- # sets (S) = $\frac{N}{P\text{-way}}$
  $= \frac{4}{4}$
  $= 1$



- When P is approach to 1 then S is approach to 1 then cache become direct.

1-way set assoc. cache of 8B with 2B Block

- # lines (N) = $\frac{8}{2}$
  $= 4$
- # sets (S) = $\frac{N}{P\text{-way}}$
  $= \frac{4}{1}$
  $= 4.$



Q Consider 16 way set associative cache of 64 KB organized into a 32B Blocks. CPU generates the 32 bit Physical add^rs to access the data, in the cache controller each tag is comprising of 1 valid bit, 1 update bit & 2 replacement bit. What is the tag mem size in the cache controller & how many tag bits are in the line.

- CM size = 64 KB

2^{32} 16 way.

$\log_2 2^4 = 4$ bit.

2^{16}

- MM size : $2^{32}B$
- 16 - way set assoc.

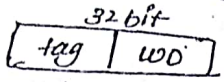
ol

- # lines (N) = $\frac{64K}{32}$

  = 2^{11}

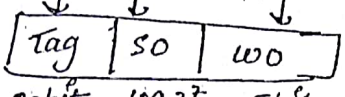
- # sets (S) = $\frac{N}{P\text{-way}}$ = $\frac{2^{11}}{16} \Rightarrow 2^7$

MM:



27 bit $\log_2 32 = 5$ bit

Set assoc. CM



20 bit $\log_2 2^7$ 5 bit

↓

7 bit

20 bit tag requ. in the line.

Tag mlm size = $\underbrace{S * P}_{\text{"N"}} * \#$ tag bits

= $2^7 * 16 * (20\text{bit} + 1 + 1 + 2)$

= $2^7 * 2^4 * 24$ bits

= 48 K bits.

Replacement algorithm

Replacement techniques are used in the cache design to replace the data when the cache is full.

1. Random [Random - Basis]
2. FIFO
3. LRU

In FIFO, replace the block w/c is present in the cache longest time (highest time stamp)

In LRU replace the block w/c is present in the cache longest time w/o reference.

Q consider 4 block cache initially empty with the foll. main m/m block references.

4, 5, 7, 4, 12, 4, 5, 13, 4, 5, 7

Identify the hit ratio

a) FIFO
b) LRU
c) Direct CM
d) 2-way set assoc. CM with LRU.

① FIFO : [sequence]

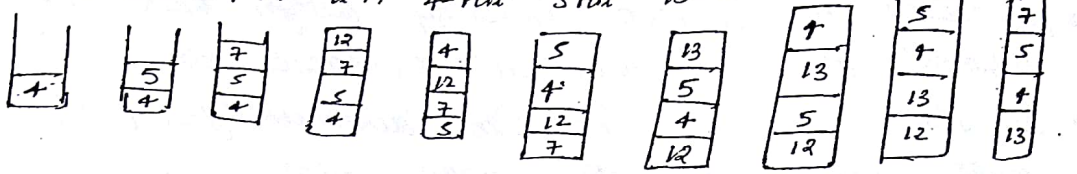
4 → M → C
5 → M → C
7 → M → C
12 → M → C
4 → H
5 → H
13 → M → C
4 → M → capacity
5 → M → "
7 → M →

H = 2/10

(MS)

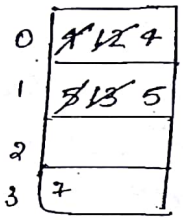
- LRU : [always keep the recent entries on the top] push down one by one.

4-M 5-M 7-M 12-M 4-Hit 5-Hit 13-M 4-hit 5-Hit 7-M



- direct mapping

C → compulsory.



4-M : 4 mod 4 = 0 (C → compulsory)
5-M : 5 mod 4 = 1 (C)
7-M : 7 mod 4 = 3 (C)
12-M : 12 mod 4 = 0 (C)
4-M → 4 mod 4 = 0
5-hit → conflict
13-M : 13 mod 4 = 1 (C)
4-hit → conflict
5-M : 5 mod 4 = 1
7-H

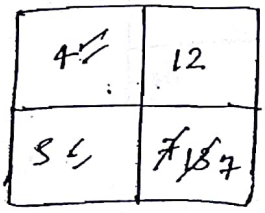
$H = \frac{3}{10}$

$H = 0.3$

$k \bmod N = i$

- 2 way set assoc

- # sets (S) = $\frac{N}{\text{p-way}} = \frac{4}{2} = 2$



4-M ; 4 mod 2 = 0
5-M ; 5 mod 2 = 1
7-M ; 7 mod 2 = 1
12-M : 12 mod 2 = 0
4-Hit
5-hit
13-M : 13 mod 2 = 1
4-Hit
5-Hit
7-M ; 7 mod 2 = 1

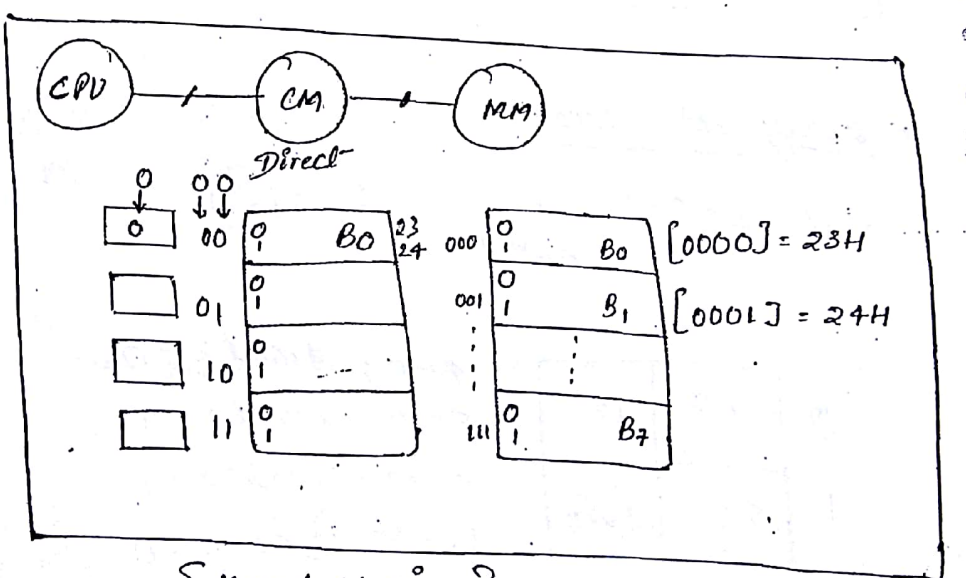
{ k mod s = i, LRU }

$H = \frac{4}{10}$

H = 0.4

Updating Techniques

- According to a mlm hierarchy design CPU always perform the o/p^n only on a cache m/m.
- On this record check the availability of a data block in the cache m/m.
- When it is present in cache then o/p^n become read hit & write hit otherwise o/p^n become read miss or write miss.
- Due to a miss o/p^n the corresponding block will be transferred from main m/m to cache called as read allocate or write allocate.
- After the allocate only on a cache m/m.



$$\left\{\begin{array}{l} K \bmod N = i \\ B_0 \bmod 4 = 0 \end{array}\right\}$$

Prog-

Block zero → 0th Byte move to reg. r0.

I_1 : MOV r0, [0000] ; B0 − 0th Byte → r0
(B0)

I_2 : Add r0, #23 ; r0 + 23 → r0

Exe^n

I_1 : Read Hit : $r_0 = B\,23H$

I_2 : $r_0 + 23 \rightarrow r_0$; $r_0 = 46H$

I_3 : write Hit ; CM

[0000] : ~~23~~ 46

MM

[0000] : 23

CM

[0000] : 46

Cache coherence

- During the execu^n of I_3 Ins^n CPU puforms the ryt o/p^n only on a cache m/m. The corresponding updation is not performed in the main m/m. so same adrs contain diff. values at diff. places. This kind of data inconsistency problem in the m/m S/S is called as cache coherence. It causes data loss. i.e

I_4 : MOV r_0, [1000] : B4 - 0th Byte $\rightarrow r_0$

(B4)

I_5

I_6

I_7

I_8 : move r_0, [0000] ; B0 - 0th Byte $\rightarrow r_0$

(B0)

Execution

I_4 : Read m&s : Read $\rightarrow$ allocate

{k mod N = i}

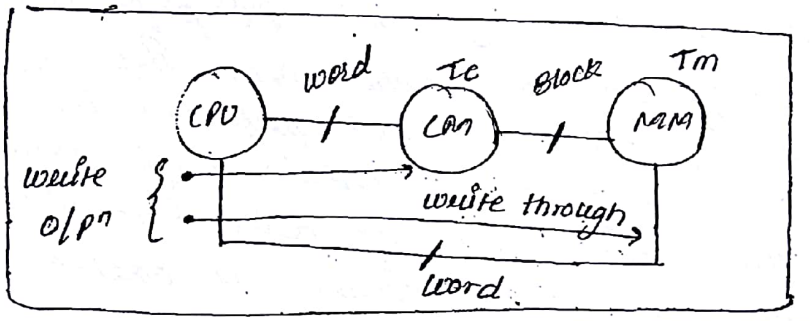
{4 mod 4 = 0}

- To handle the above problem updating techniques are used in the m/m design this is of two types.

(a) write - through
(b) write - back.

(1) write - through.

- In this technique CPU performs the simultaneous write o/p in both cache m/m & main m/m so coherence is not present Inclusion is success.
- Inclusion means lower level m/m data always become the subset of a higher level m/m data.



$$\text{Simultaneous write opn } (T_W) = \max\left[\begin{matrix}\text{word updation time in CM}, & \text{word updation time in MM}\end{matrix}\right]$$

- Time required to a read one word data from the m/m is called as read cycle time i.e

$$T_{avg\ read} = H_r T_c + (1 - H_r)(T_m + T_c)$$

H_r → read hit, T_c → read data, $(1 - H_r)$ → read miss, T_m → read allocate, T_c → read data.

- Time required to ~~access~~ write one word data

$$Tavg_{write} = H_W T_W + (1 - H_W)(T_m + T_W)$$

- $H_W T_W$: write hit, simultaneous write
- $(1 - H_W)$: write miss
- T_m: write allocate
- T_W: no cache write simultaneous write

- Average access time of the mem when consi-dering both read & write o/p^n is:

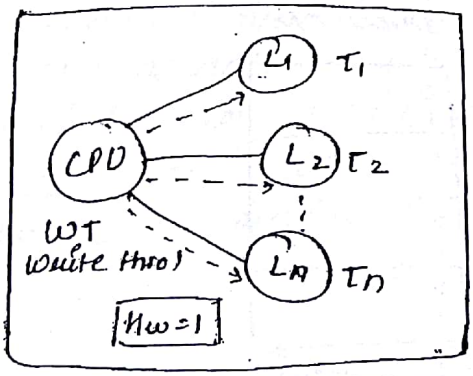
$$Tavg_{WT} = (Freq_{read} * Tavg_{read}) + (Freq_{write} * Tavg_{write})$$

Freq → % of read o/p

$$\eta_{WT} = \frac{1}{Tavg_{(WT)}} \text{ words/sec}$$

write thro'

Note: When the write thro' technique is implem-ented in a simul. access mem org^n then hit ratio for write o/p^n always become 1, beco2 data block is need not be required in the 1st level.



→ read is normal read
→ write is simul write

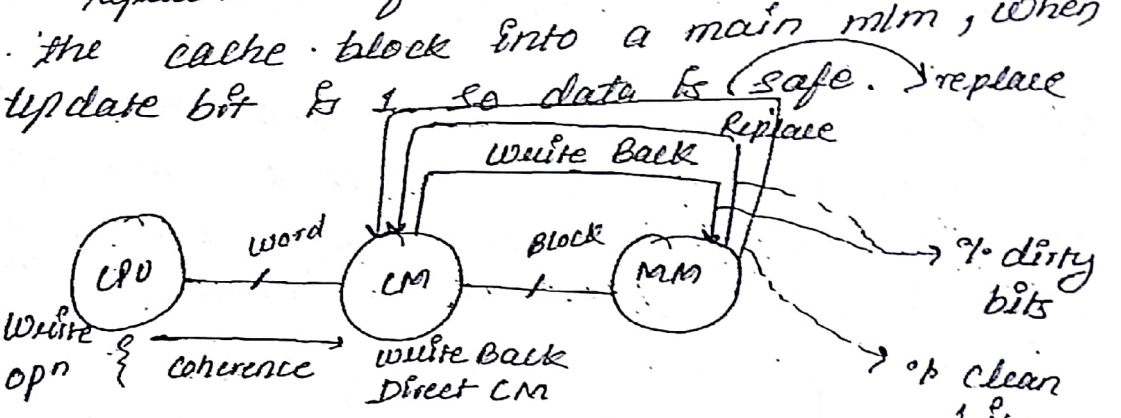
$$Tavg_{write} = T_n$$

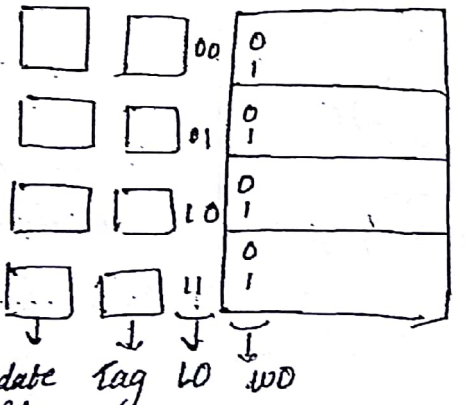
$$Tavg_{read} = H_1 T_1 + (1 - H_1) H_2 T_2 + \ldots$$

Note → When the p^n contain cache mem code, write thro' technique & HW = 1 then use simultaneous

(2) Write Back

- In this technique CPU performs the write o/pn only on a cache m/m so coherence present.
- This coherence doesn't cause the data loss because CPU reads the status of a update bit before replacement.
- In the write back cache design each line contain one extra bit name as update bit.
- It is set when the cache block is updated otherwise reset.
- Before replacement of the cache data, CPU write back the cache block into a main m/m, when the update bit is 1 so data is safe. replace





0 (clean Bit) → 1 (dirty Bit)

$$T_{avg_{write}} = H_w T_c + (1 - H_w)\left[\%\ dirty\ (T_m + T_m + T_c) + \%\ clean\ (T_m + T_c)\right]$$

- $H_w T_c$ → write hit: write data
- $(1 - H_w)$ → write miss
- dirty bits → write back: write allocate: write data
- clean bits → write allocate: write data

- Avg access time Both read & write

$$T_{avg_{WB}} = (freq_{read} * T_{avg_{read}}) + (freq_{write} * T_{avg_{write}})$$

Q In a 2-level mlm s/s level 1 mlm is cache m/m w/c is referenced by the CPU with a access time of 50ns. Level-2 mlm is a mm it access time is 200ns. CPU generates 60% requests for read o/pn & remaining for write o/pn. Hit ratio for read o/pn is 80%. what is the avg. access time of a mlm when considering both read & write o/pn wing write thro' protocol.

Q" contains 2 level
- Level-1 → Cache
- L-2 → mm

Given
- L_1 → CM ; $T_C = 50ns$
- L_2 → MM ; $T_m = 200ns$
- $f_r = 60\%$, $H_r = 80\%$
- $f_w = 40\%$, $H_w = 1$

{ simultaneous access mlm org" }

$$T_{avg\ read} = H_r T_c + (1 - H_r) T_m$$
$$= (0.8 * 50) + (1 - 0.8)\, 20$$
$$= 80ns$$

$T_{avg\,write} = T_m$ (freq. of read * read cycle time)
$= 200ns$

$T_{avg\,wt} = (f_r * T_{avg\,r}) + (f_w * T_{avg\,w})$
$= (60\% * 80ns) + (40\% * 200ns)$
$= 128ns$

Q Consider a write back cache design with access time of 40ns. Hit ratio for read & write o/p^n & 70% & 80% respectively. mm access time is 160ns SLS generates 70% of request for read o/p^n & remaining for write o/p^n What is the average access time of mm for read & write o/p^n both.

- write - back CM
- CM : $T_C = 40ns$
- MM : $T_m = 160\ ns$
- $f_r = 70\%$ [% clean bits]
- $f_w = 30\%$ [% dirty bits]
- $H_r = 70\%$
- $H_w = 80\%$.

{$H_w \neq 1$}
↳ Hierarchial org^n.

$T_{avg\,read} = H_r T_C + (1-H_r)$ [% dirty bits $(T_m + T_m + T_C)$ + % clean bits $(T_m + T_C)$]
$= (0.7 * 40) + (1-0.7)$ [30% [160 + 160 + 40] + 70% [160 + 40]]
≈ 28 102.4ns

$T_{avg\,write} = H_w T_C + (1-H_w)$ [% dirty $(T_m + T_m + T_C)$ + % clean $(T_m + T_C)$]
$= (0.8 * 40) + (1-0.8)$ [30% (160 + 160 + 40) + 70% (160 + 40)]

$= 81.6ns$

$Tavg_{wb} = (fr * Tavgread) + (fw * Tavgwrite)$

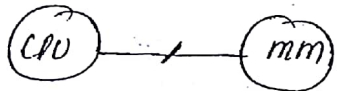
$= (0.7 * 102.4) + (0.3 * 81.6)$

$= 96.16ns$

Multilevel cache m/m

To reduce the miss panc penality multilevel caches are used in the computer design.
~~miss penality~~ miss penality means time required to access the data from higher level to lower level due to a miss o/p in lower level m/m.

System old

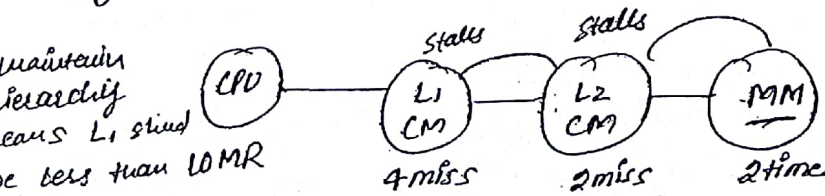


LOMR — 10 times MM accessed.

System with cache

LO MR — 4 miss — 4 times MM accessed.

System with multicaches



- In the multilevel cache design level 1 to level 2 capacity is in a ↑ order & associativity is in a decreasing order.
- Two kinds of miss rates can be calculated in the multilevel cache design.

① Global miss rate (GMR) = $\frac{\#\text{ misses in cache}}{\text{Total \# CPU generated references.}}$

② Local miss rate (LMR) = $\frac{\#\text{ misses in cache}}{\#\text{ accesses to the cache}}$

$$GMR_{L_1} = \frac{4}{10} \Rightarrow 0.4$$

$$LMR_{L_1} = \frac{4}{10} \Rightarrow 0.4$$

$$GMR_{L_2} = \frac{2}{10} \Rightarrow 0.2$$

$$LMR_{L_2} = \frac{\#\text{ miss}_{L_2}}{\#\text{ miss}_{L_1}} \Rightarrow \frac{2}{4} = 0.5$$

*****

$$T_{avg} = \text{Hit time}_{L_1} + (\text{miss rate}_{L_1} * \text{miss penality}_{L_1})$$

$$\text{miss penality}_{L_1} = \text{Hit time}_{L_2} + (\text{miss rate}_{L_2} * \text{miss penality}_{L_2})$$

MM access time

*****

Avg m/m stalls / ins^n = (# misses$_{L_1}$ / ins^n * Hit time L_2) + (# misses$_{L_2}$ / ins^n * miss penality L_2