Computer organization :

Introduction

Computer organization	Computer Architecture
- physical organization	- logical organization
- Touchable	- non touchable

- Computer organization / Architecture deals with
  1) Instructions
  2) Addressing modes
  3) Data format

1) Instructions

based on capability

RISC	CISC
☺	☺
- Reduced Instruction set Computing	- Complex Instruction set Computing
- Limited capability	- More capability
- Clocks per unit Instruction = 1	- Clocks per Instruction ≥ 1
- Fixed size Instructions → reduces Compiler capacity	- variable size Instructions → Increases compiler complexity
- H/w Control unit design	- micro operation Control unit design
- parameter passing is more efficient because of register windows	
- Rich set of registers	
☹	☹
- programmer overhead is more	- programmer overhead is less
- Limited Addressing modes (AM)	- More Addressing modes
- AM ∝ Flexibility of pgm development (less flexible)	- More flexible

Note :
Instruction pipeline is efficiently used in RISC systems due to fixed size.

2) Addressing modes : The way how operand is given in the Instruction

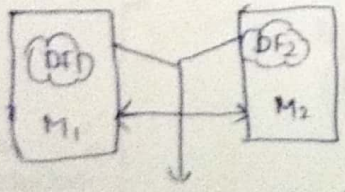
① Immediate : operands are provided along with the Instruction (fastest)

② Direct : Reference to the operands

③ Indirect : Reference to the Effective Address

3) Data format : How to Interpret the binary string

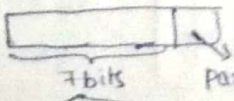
- For meaningful Communication

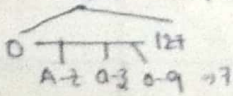


They should settle with one format

Data formats

ASCII





EBCDIC

Extended binary Coded decimal Interchange Code
- 8bits / char

- Computer organization deals with physical devices and their Interconnections with the aim of improving performance → metric → MIPS (Million Instructions per second)

Ex:

m/c	Mips
A	10
B	>10

More performance

Mips depends on
- nature of Instructions
- nature of H/w
- nature of Ins set Arch & Compiler

different on diff m/cs → Sol → (metric) FLOPS (Floating point operations per second)

Goals of Computer organization

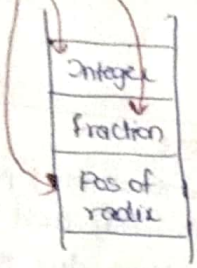
- Improving performance
- reducing time

Performance is improved based on the way data is represented, stored, accessed, processed and communicated.

How Data represented :

Data : 3.5 2

Rep 1 :



3 Memory Access

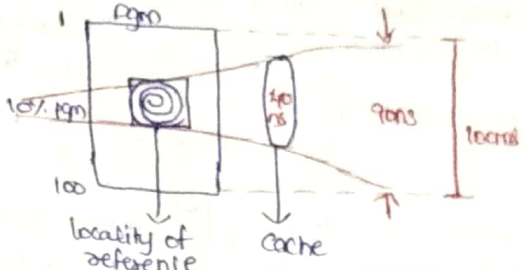
Rep 2 :

1 Memory Access (More performance)

How Data stored :

Patterson 90 to 10% rule :

10% of pgm consume 90% of the time.



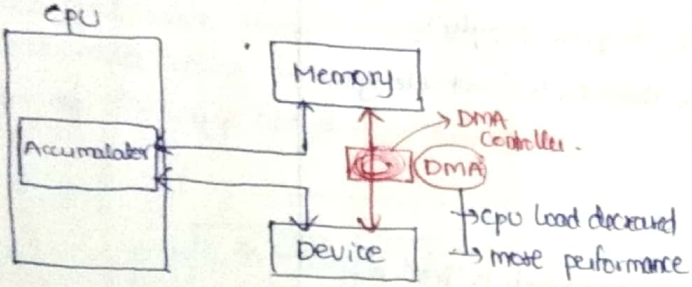
locality of reference

Cache

Without cache → 90 + 10 = 100ns (10% pgm, 90% pgm)

With cache → 40 + 10 = 50ns (More performance)

How Data Accessed :



→ CPU load decreased
→ more performance

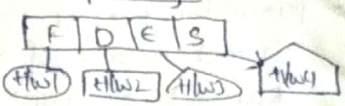
How Data processed :

(More performance)

Sequential

Clks = 4 * num of ins

Pipelining



1st ins → 4 cycles

next ins → 1 cycle

Speed up Factor (S) = Time without pipeline / Time with pipeline

How Data Communicated

- Message passing
  - Message process interface (More performance) ↓ supports ⇒ Parallel Computation
- shared memory
  - work division
  - ↓
  - work assignment
  - ↓
  - work collection

Note :

How data	More performance
Represented	M E
Stored	Cache
Accessed	DMA
Processed	Pipelining
Communicated	Message passing interface

Computer Architecture Classification :

Von-neumann

→ First memory based CA

Basis : store of pgm and data

↓

Stored pgm Concept

∴ Problem size and its dependency memory space

Harvard

Basis : Separate pgm store & data store

- Stored pgm Concept

Flyn

→ Basis : Instruction stream and data stream

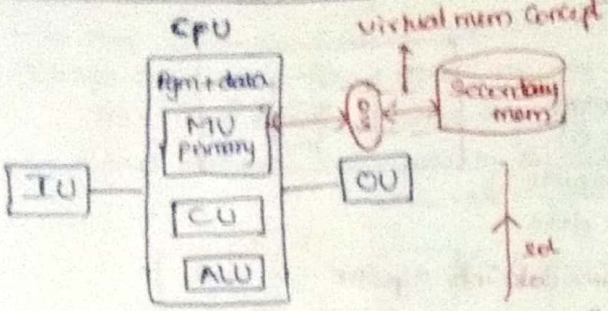
Note : Fastness

Reg > Cache > Primary (volatile) > Secondary

RAM, ROM (non volatile)

Boot pgm

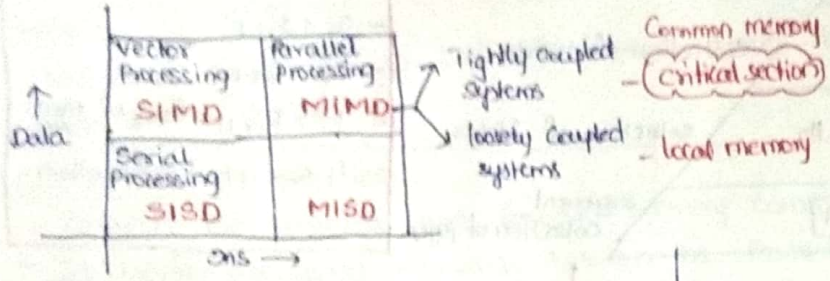
Von neumann Architecture :



∴ If prob size is more than our memory then this Arch fails to execute.

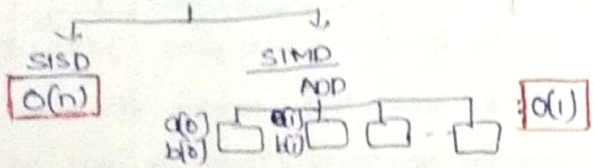
Flyn Architecture :

- It mainly focuses on flow of Instructions

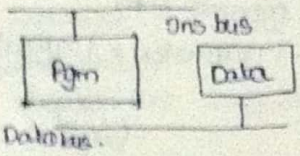


SIMD - single Instruction multiple data

Ex:- for(i=0; i<n; i++)
c[i] = a[i] + b[i]



Harvard Architecture (Patterson)

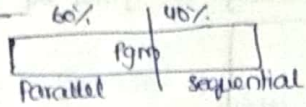


- changes in data will not affect pgm & viceversa
- parallel buses are used so more performance is obtained.

Note :

Streams	Control unit CU	Processing Element	Memory
SISD	1	1	Single unit
SIMD	1	M	Modules
MISD	M	1	Single unit
MIMD	M	N	Modules

Practically Implemented Arch



Single Core M/C = 100 x sec

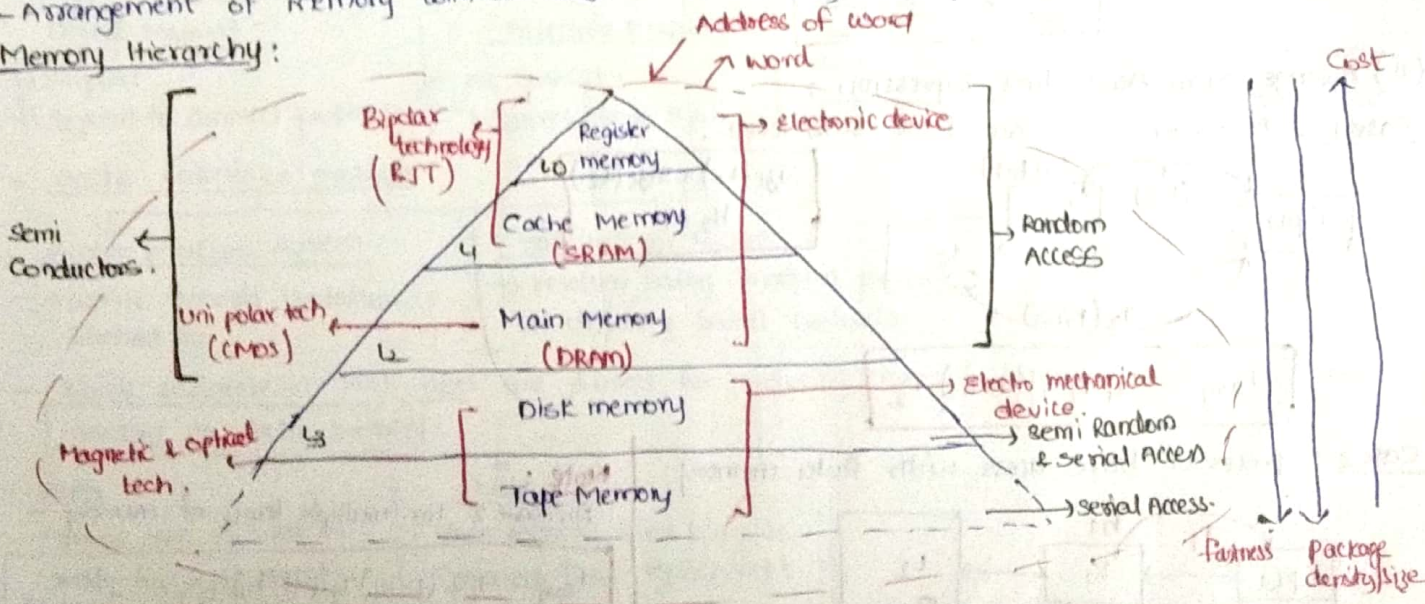
Dual Core M/C = $\frac{40\% + 60\%}{2}$ sec.

Speed up = $\frac{100\%}{\frac{140\%}{2}}$ ⇒ 1.42 ⇒ 42% Increase.

Memory organization :

- Arrangement of memory with an objective of reducing Average Access time.

Memory Hierarchy :



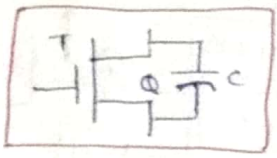
CMOS - Complimentary metal oxide semiconductor field effect transistor

BJT - Bipolar Junction transistor.

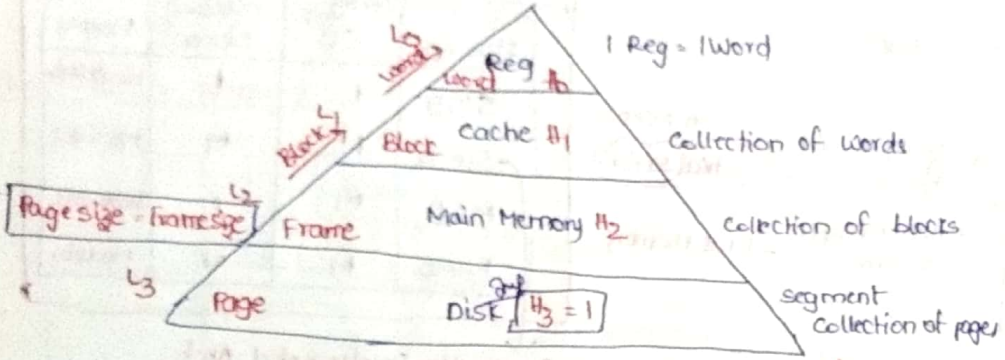
- L_0, L_1, L_2, L_3 are levels according to distance b/w processor

SRAM
- Static RAM
- 6 Transistors
- ☺ - Faster
- No refreshing
- Costly bcoz to store 1 bit we require 6 transistors

DRAM
- Dynamic RAM
- 1 Transistor + 1 capacitor
  - charged (Logic 1)
  - discharged (Logic 0)
- Discharging is done at regular intervals so refreshing is done
  - Refresh cycle
  - placing same data into capacitor
- slower compared to SRAM
- Cheaper compared to SRAM



- *)



Note: 'i' → level.
- $H_0 < H_1 < H_2 < H_3 = 1$
- $T_i < T_{i+1}$
- $S_i < S_{i+1}$
- $C_i > C_{i+1}$
- $F_i > F_{i+1}$ ⇒ Freq of mem Accen
- $I_i \subset I_{i+1}$ ⇒ Information

- Performance of Memory Hierarchy is given by Hit ratio

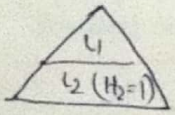
$$H \propto S \text{ (size)}$$

$$H \propto \frac{1}{\text{Avg mem Accen time}}$$

- The sideeffect of memory hierarchy is Data Inconsistency at diff levels
  - Sol: Proper write operns

Mathematical Expression for Memory Hierarchy:

(I) Average Cost Expression



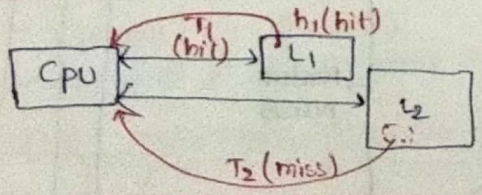
Total size = $S_1 + S_2$

Total cost = $C_1S_1 + C_2S_2$

$$\text{Avg cost} = \frac{C_1S_1 + C_2S_2}{S_1 + S_2}$$

(II) Average Mem Access time Expression:

Case 1: Processor have Access to both levels.

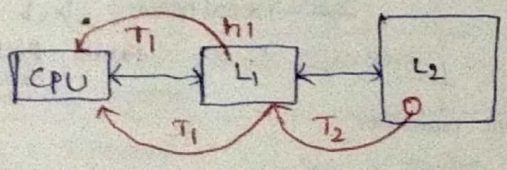


size(L_1) < size(L_2)

$H_2 = 1$

$$T_{avg} = h_1T_1 + (1-h_1)T_2$$

Case 2: Processor have acess with faster memory



$$T_{avg} = h_1T_1 + (1-h_1)[T_2+T_1]$$

$$T_{avg} = T_1 + (1-h_1)T_2$$

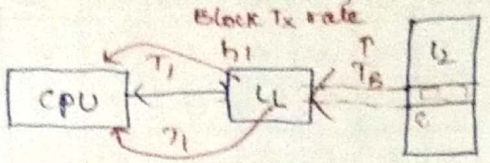
Note:
In Case 2 For multiple levels of memory

$$T_{avg} = T_1 + (1-h_1)T_2 + (1-h_1)(1-h_2)T_3 + \cdots + (1-h_1)(1-h_2)\cdots(1-h_{L-1})$$

Case 3 : Exploring locality of reference

- In this case the current address is not available in faster memory, then a block of words are moved from slower memory to faster memory

This Ensure the subsequently addressed words are supplied from faster memory hence in future reference miss penality is reduced.



$$T_{avg} = h_1 T_1 + (1-h_1)[T_B + T_1]$$

$$T_{avg} = T_1 + (1-h_1) T_B$$

Note :

For multiple (k) levels of memory

$$T_{avg} = T_1 + (1-h_1) T_2 + (1-h_1)(1-h_2) T_3 + \cdots + (1-h_1)(1-h_2)\cdots(1-h_{k-1}) T_B$$

Note :

Hit value never changes (upon increase / decrease) Individual mem accen but affects effective mem accen time

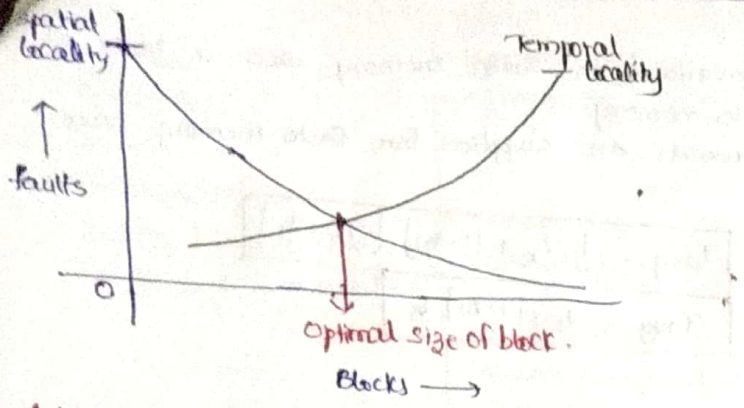
Cache Memory : overview

- Cache is the smallest and fastest memory component in hierarchy
- It bridges the speed mismatch between the fastest processor and slowest memory component
- It maintains locality of reference
- Cache memory
  - Based on Contents
    - Ins cache
    - Data cache
    - Combined cache
  - Based on which address used
    - Physically Indexed cache (PA is used)
    - Virtually Indexed Cache (VA is used)
- The cache memory and main memory are divided into equi sized blocks
- The num of blocks in cache memory is less than the num of blocks in main memory
- So, Address Mapping decide which main mem block has to be placed in Cache memory
  - Direct Mapping
    - Simplest
    - Exposed to conflict problems
  - Associate mapping
    - No Conflicts
    - requires more H/W
  - set Associate mapping
    - reduces Conflicts Compared to Direct mapping
    - reduces H/W Compared to Associate mapping
- Cache Coherence problem
  - Single processor system
    - Write through updation Resolved by
  - Multiprocessor system
    - resolved using Snooping protocol or directory based protocols.
- Block replacement techniques are aimed for reducing num of block moments from main memory to cache memory.
  - FIFO
    - Arrival time is used as basis for replacement
  - LRU
    - Most recently used block is Prevented from replacement

Block size vs performance :

- Block size selection must minimize the faults associated with locality of reference
  - Spatial locality
    - Bigger blocks are better
  - Temporal locality
    - smaller blocks are better

Optimal block size is selected



Address mapping:

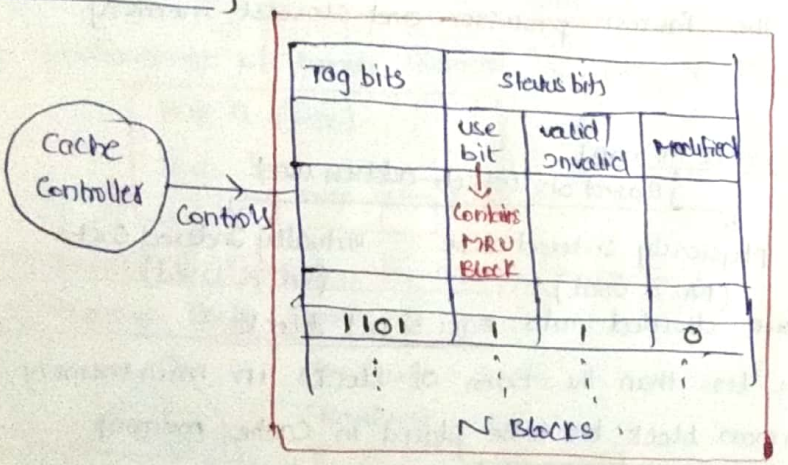
- The Address mapping decide the location of main mem block in cache mem

Tag bits : They denote which main memory block is present in that cache position at that time.

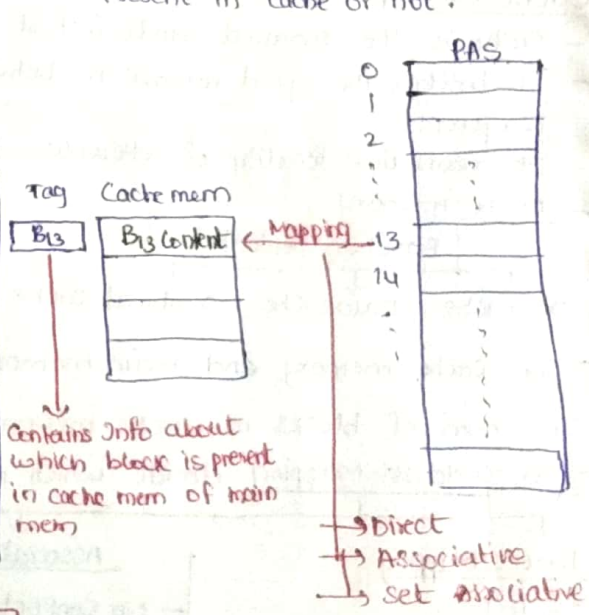
- All this tag information is placed in a new memory called **Cache directory**
  - used to know whether addressed word is present in cache or not.

*) Space reqd for cache directory = N * (Tagbits + Status bits)

Cache directory



Info about Main mem in Cache directory



① Direct Mapping:

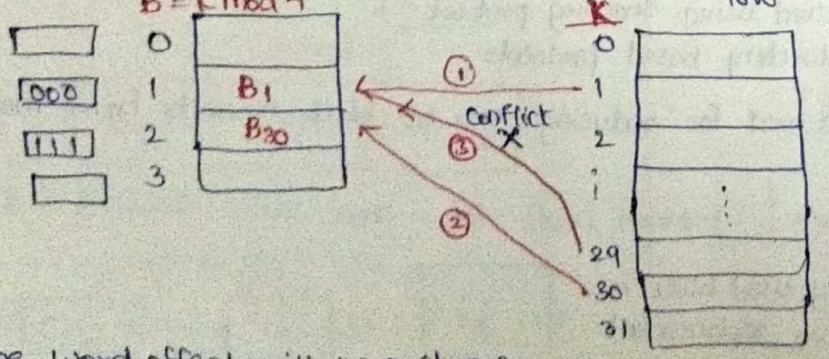
- place the k^{th} main memory block in **k Mod N** Cache block.
  - N: cache size

Ex:- Cache = 64 words

Blocks (N) = 4 ⇒ **1B = 16W**

Main memory = 512 words

Blocks = $\frac{512W}{16W}$ = 32 Blocks.



PA

Blocks	offset
5 bits	4 bits

→ Tag (3 bits) | Cache offset (2 bits)

In Direct Addressing: (PA)

Tag	Cache offset	Word offset
$\log_2(\frac{m}{N})$	$\log_2 N$	$\log_2 W$

Cache index.

- The word offset will never change whether the block is in cache or main mem

Conflict problem: let k_1, k_2 be two main mem blocks

If $k_1 \bmod N = k_2 \bmod N$ ⟶ Conflict

- The Direct mapping need

Least num of mappings
Simpler to design
Uses only one tag comparator

	tag 000	001	010	011	100	101	110	111
00	0	4	8	12	16	20	24	28
01	1	5	9	13	17	21	25	29
10	2	6	10	14	18	22	26	30
11	3	7	11	15	19	23	27	31

cache offset

Tag		
010	0	B_8
100	1	B_{17}
010	2	B_{10}
001	3	B_7

Direct mapping

0, 1, 2, 3, ..., 31

Note :

$$k = f(T, B, N) = N * T + B$$

word 300

$\frac{300}{16} \Rightarrow 16*18+12$

18^{th} Block of 12^{th} word

Not present in cache

word 167

$\frac{167}{16} \Rightarrow 16*10+7$

10^{th} Block of 7^{th} word present in cache.

(or)

010 10 0111 ⇒ 167

Tag cache offset word offset

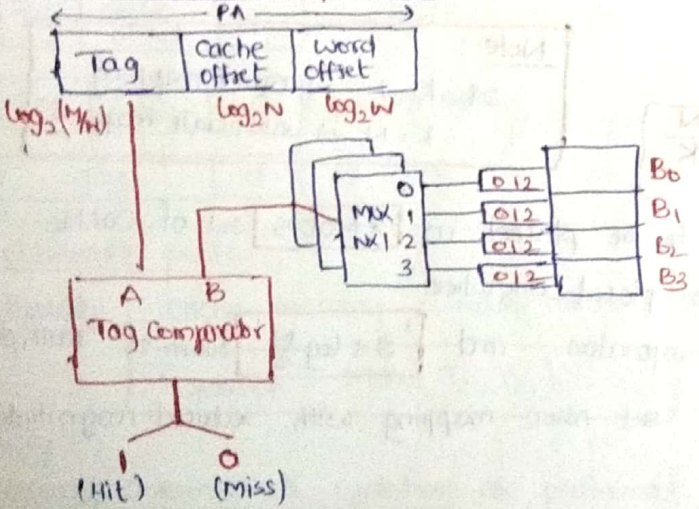
- A word is said to be hit if tag info of PA is matching with the tag info of the contained cache block which was refered by cache block offset

Block	Tag	Main mem block (k)	word range (k*16, k*16+15)
0	2	8	128 – 143
1	4	17	272 – 287
2	2	10	160 – 175
3	1	7	112 – 127

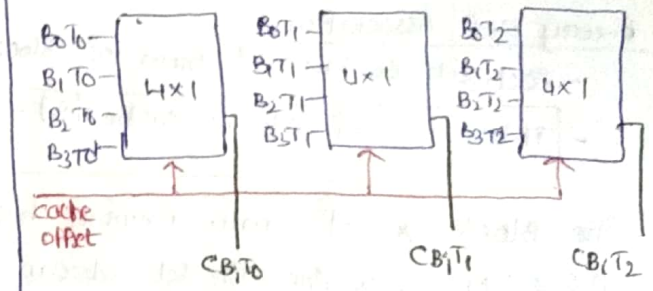
☺
Easy to Implement
(N-1) MUX are need
1 (N-1) comparator need

☹
Conflicts are present

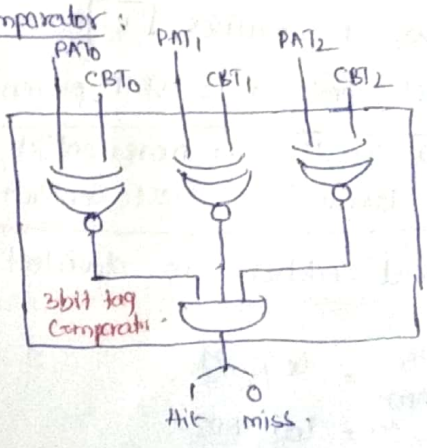
H/w Approach for hit Computation :



Multiplexer :



Comparator :



Hit latency = Mux delay + Comparator delay

(2) Associate Mapping :

- In Associate mapping any block of main memory can be placed any where in Cache and hence there is no Conflict problem.
- To know the address word present in cache, not all cache blocks have to be Checked so we require n tag Comparators

- The physical address is splitted into two portions

Tag	word offset
$\log_2 M$	$\log_2 W$

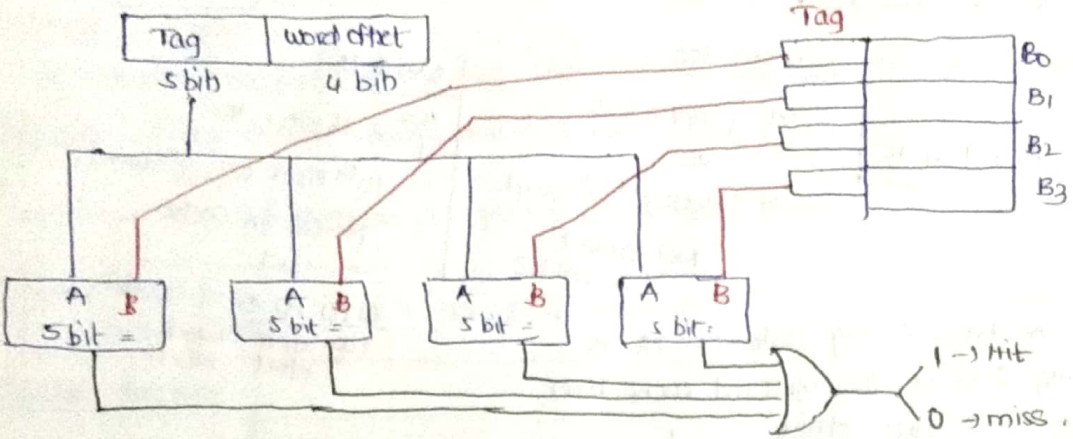
Note:
Tag bits AM = Tag bits DM + Cache offset DM

- More tag bits are present in Associate mapping

☺ No Conflicts | ☹ More H/w reqd
N 'm' bit Comparator + 1 or gate | Imum reqd

Implemented as

H/w Approach for hit Computing:



Hit latency = Comparator delay + OR gate delay / MUX delay

(3) Set Associate Mapping:

☺ Conflicts reduced
less H/w than Associative | ☹ Slower than Associative

- In Set Associate mapping the cache is divided into logical sets

k-way Set Association:
- Each set contains 'k' num of blocks
- Total num of sets in cache (S) = $\frac{N}{K}$

Note:
If K = 1 ⇒ Direct mapping
K = N ⇒ Associate mapping

- The Block x of main memory has to be placed in x MOD S set of cache where $0 \le x \le M-1$, within the set it can be placed anywhere
so, it requires 'K' num of Comparators and '$S * \log \frac{M}{S}$' Num of multiplexers
- Conflict problem is still present with set Assoc mapping with reduced magnitude

K → Small ⇒ more Conflicts
K → large ⇒ less Conflicts

- Physical address is divided into:

Tag	set offset	word offset
$\log_2 \frac{M}{S}$	$\log_2 S$	$\log_2 W$

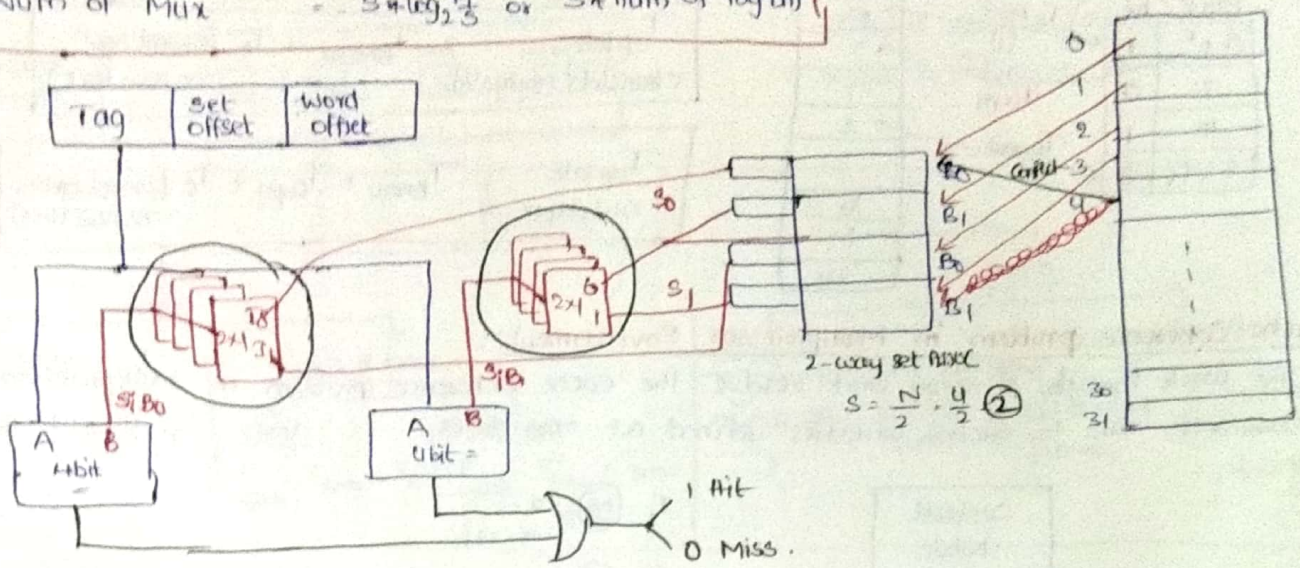
- Tag bits SAM = $\log_2 \frac{M}{S}$ = $\log_2 \frac{MK}{N}$; $S = \frac{N}{K}$

Tag bits SAM = $\log_2 \frac{M}{N} + \log_2 K$
used when block size is not

Note:
$\log_2 \frac{M}{N}$ (min) < $\log_2 \frac{M}{N} + \log_2 K$ < $\log_2 M$ (max)
DM | SAM | AM
Tag bits

Hardware Approach for hit Computing :

- Num of Comparators = k (N bit Comp)
- Num of Mux = $S * \log_2 \frac{M}{S}$ or $S *$ num of tag bits



Hit latency = Mux delay + Comparator delay + OR gate delay | Mux delay

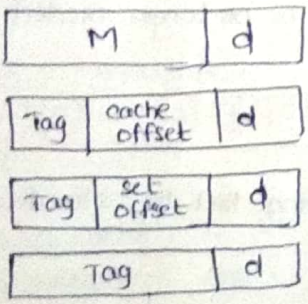
Points to remember

- If block size is not known

  Num of tag bits = $\log_2 \frac{M}{N} + \log_2 k$

M - capacity of main memory
N - capacity of cache memory
W - Num of words
B - Num of Blocks
k - set Assoc splitting value

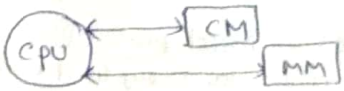
*



Cache Coherence problem :

Write through : Cache memory & main memory are updated simultaneously

$T_{updation} = Max(T_c, T_m) + T_M$



☺ Simplicity

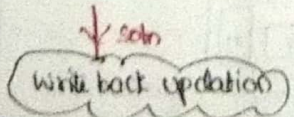
☹ unnecessary intermediate updations are performed in main memory.

↓

for more write opers write through gives worst performance

Ex:- for(i=0; i<1000; i++) → a is updated 1000 times in both cache & main memory
a = a+i;

Write back updation :

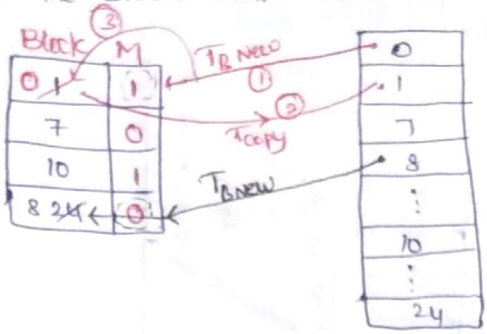


- Current updation is performed in cache memory
- Main memory is updated when Considered block in cache is Choosen for replacement

☹ Updation o/H is more. → Soln It uses modified bit Concept

Modified bit (Dirty bit) $\begin{cases} 0 \Rightarrow \text{not modified only refered ie read} \\ 1 \Rightarrow \text{Modified ie refered and write are done} \end{cases}$

- Denote the status block modification in cache memory

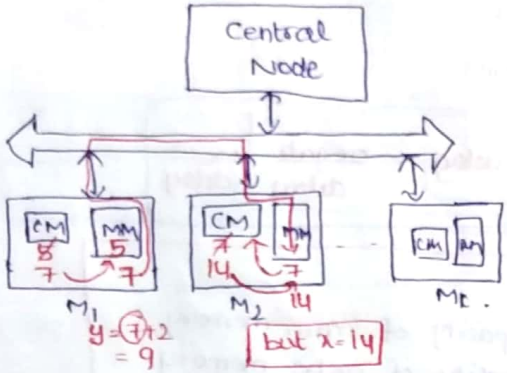


$T_{update\ clean\ slot / Empty\ slot} = T_{BNew} + T_c$ (current oper^n on new block)

$T_{update\ Dirty\ block} = T_{BNew} + T_{Copy} + T_c$ (current oper^n on new block)

Cache Coherence problem in Multiprocessor Environment:

The write through/write back strategy can't resolve the cache coherence problem in multiprocessor environment due to Global variables defined at other places.



T_1: M_1 : x = 5, x = x+2

T_2: M_2 : x = x+7

T_3: M_1 : y = x+2

∴ write through, write back can't resolve cache coherence problem

↓ soln

Snooping protocol, Directory based Protocol

→ central node perform updation of shared var → Update all / selective update

Update all:

- updation of shared variable at all places

∴ - unnecessary updations are performed if that shared variable is no longer needed ie (Next use → Nill)

- central node will take responsibility

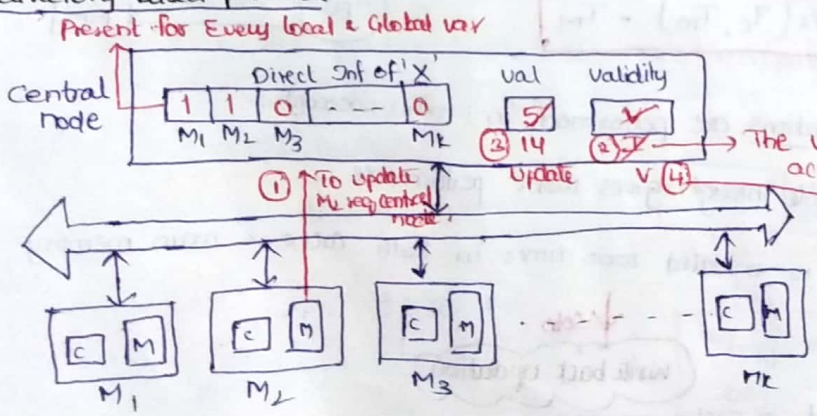
Selective Update:

- Individual machines will take updation responsibility by broadcasting that the shared variable is updated.

Directory based protocol:

In directory based approach the central node maintain the variable and its latest value and the bit string representing which m/c are having that shared variable.

1) Full directory based protocol

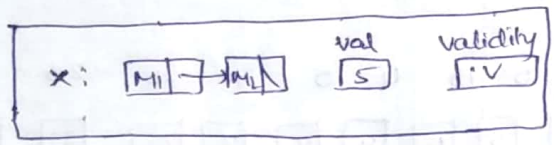


∴ It Consume more space for directory Information.

2) Limited directory protocol :

For each variable a list is maintained for directory information

- Num of lists = Num of nodes that access a particular variable.



☺ Effective directory space

☹ Complex to implement.

Points to remember :

- Processor
  - Uni
    - Write through: $T_{update} = Max(T_c, T_p)$
    - Write back
      - clean: $T_{update} = T_{BNew} + T_c$
      - dirty: $T_{update} = T_{copy} + T_{BNew} + T_c$
  - Multi
    - Snooping
      - update all
      - selective update
    - Directory based
      - full directory → Array
      - limited directory → linked list

Block replacement strategies :

- These strategies are aimed to reduce the num of block moments so that the performance is improved.

Techniques

- Optimal
  - Benchmark
  - practically not implemented
- FIFO
  - Imp using Queue
  - with spatial locality its performance is better
- LRU
  - Fully Associate cache
    - our Assumption is MRU block may require again & again
    - Queue, (stack) are used for implementing LRU (+ use bit)
  - Set Associate Cache

① Optimal

- K = 4

Conflict : Arrival time

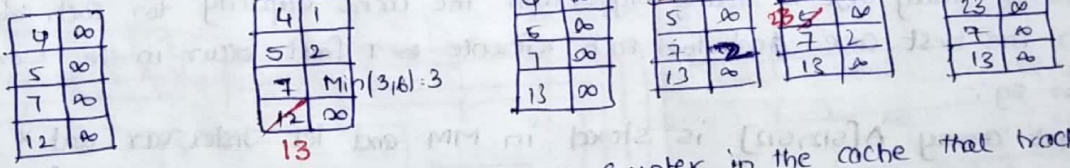
K : (4) (5) 7 (12) [4] [5] 13 [4] [5] [7] 25 23 [7]

Hits - 6
removals - 3
Blocks in cache : 25 23 7 13

→ Remove
→ Hit
↳ → K movement

Optimal replacement Algo uses forward reference distance.

ie K : 4 5 7 12 4 5 13 4 5 7 25 23 7



- To know which arrived first in the cache we place a counter in the cache that tracks all the blocks coming in the cache.

② FIFO

- K = 4

K : (4) (5) (7) (12) [4] [5] (13) (4) 5 7 25 23 [7]

Hits = 3
Num of removals = 6
Last block moved out : 4
Blocks in cache : 5 7 25 23

$$\text{Efficiency} = \frac{\text{Hits}_{FIFO}}{\text{Hits}_{opt}} * 100$$

$$= \frac{3}{6} * 100 = 50\%$$

③ LRU : [ ] → old use (Fully Associate cache)

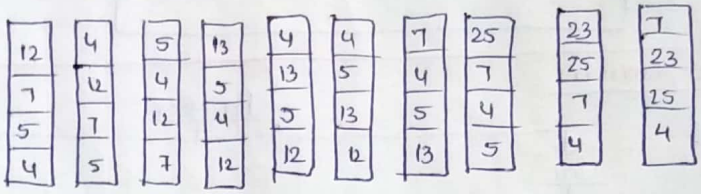
- k = 4

k : 4 5 7 12 4 5 13 4 5 7 25 23 7 → $\eta = \frac{5}{6} * 100 = 83.3\%$

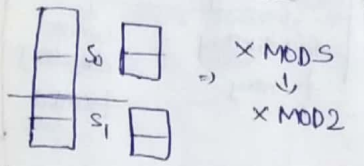
Queue implementation : →

k: 4 5 7 12 4 5 13 4 5 7 25 23 7

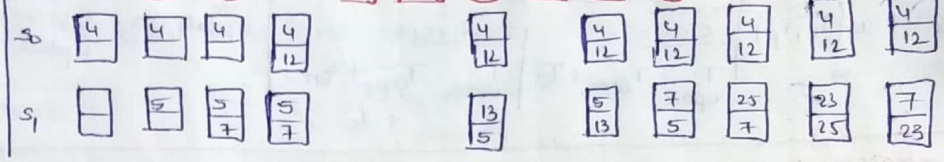
Hits = 5
Removals - 4
Last block moved out - 4
Blocks in cache : 7 23 25 4
MRU Block : 7



Set Associate Cache :



k: 4 5 7 12 4 5 13 4 5 7 25 23 7



Hits = 4
Removals = 5
Blocks in S0 - 4, 12
S1 - 7, 23

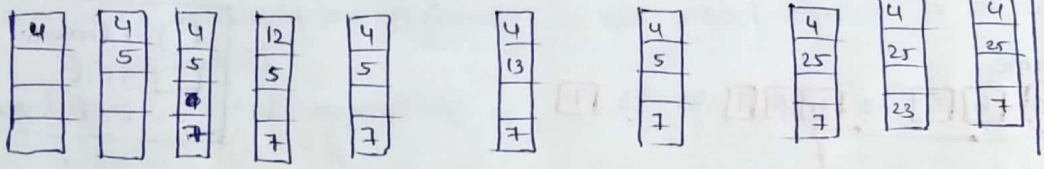
$\eta = \frac{4}{6} * 100 = 66.66$

Direct cache :

X MOD N → X MOD 4

k : 4 5 7 12 4 5 13 4 5 7 25 23 7

X MOD 4 : 0 1 3 0 0 1 1 0 1 3 1 3 3



Hits - 3
Removals - 7
Last block taken out - 23
Blocks in cache - 4 25 7

$\eta = \frac{3}{6} * 100 = 50\%$

Effect of Array placement in Memory and its influence on performance :

- The array is normally stored in RMO, If the Accessing is performed row wise then it gives optimal performance however the column wise references degrades the performance
- If the array size is fairly bigger than the cache capacity for each block 1 fault occurs in the best case and for each reference 1 fault occur in the worst case.

ie for eg :-

Float array A[512,512] is stored in MM and let index var and x avail in processor registers. Direct mapped cache of 64KB with 64B blocks are used for implementing Pgms P1 and P2. If M_1 and M_2 are miss references and each array element occupies 8B then

(i) M_1 32K
(ii) M_2 256K

```
P1 : for i = 1 to 512
       for j = 1 to 512
         x+ = A[i][j];
```

```
P2 : for j = 1 to 512
       for i = 1 to 512
         x+ = A[i][j]
```

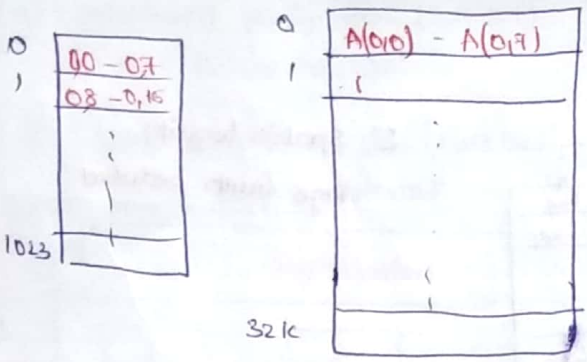
Sol) Direct Mapped cache = 64KB

Blocks : $\frac{64KB}{64B}$ = 1024 Blocks in Cache

Main Memory size = $2^9 * 2^9 * 8 = 2$ MB

Total Element references - $2^9 * 2^9 = 2^{18} = 256K$

Blocks read = $\frac{2MB}{64} = 32K$ Blocks



For P_1 : A[00] [01] - - - - [07] — 8 ref

M H - - - - H ⇒ 1 Miss 7 Hits

8 ref → 1 Miss

256k ref → x

$x = \frac{256K}{8}$ ⇒ 32K miss (best case)

For P2 : A[00] [10] [20] - - - [70]

M M M - - - M

8 ref → 8 miss

256k ref → x

x = 256k miss (worst case)

Consider P_1 which Array Element replaces A[0][5] → A[16][5]

Note : A[i][j] ⟶ A[i+16][j]

If cache is organized into 2 way set Assoc then A[0][4] is occupied with.

A[8][4] ⇒ 8*64 ⇒ 512 ⇒ ✓

A[16][4] ⇒ 16*64 ⇒ 1024

A[4][4] ⇒ 4*64 ⇒ 256

A[4][0] ⇒ 4*60 ⇒ 256.

Sets = $\frac{1024}{2}$ ⇒ 512 — B0

x MOD 512

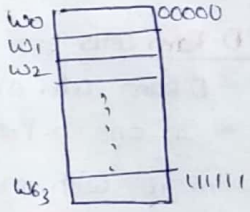
0 512 1024 1536

Performance Analysis of main memory :

The main memory is divided into modules and Info is distributed across the modules So that the data transfer rate is more and thereby Increasing the performance

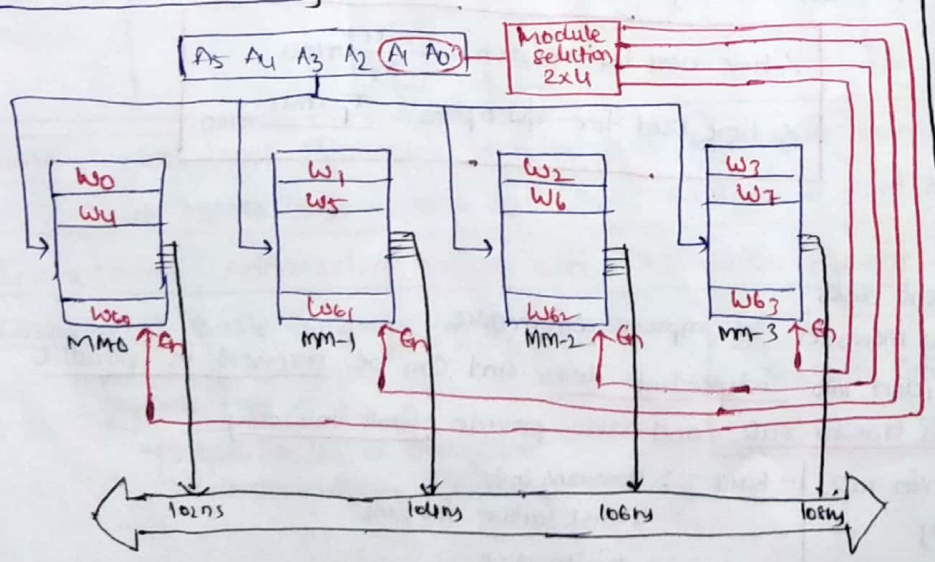
This distribution of Info is called Interleaving

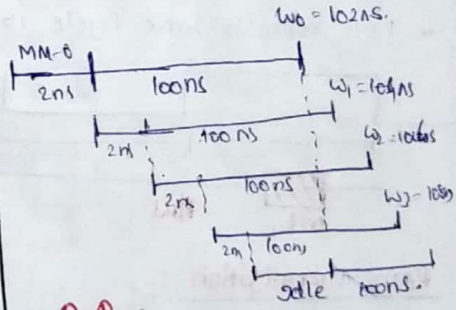
- Higher order → Improves spatial locality
- lower order → fast response.



- The modules are accened in parallel ie (Internal opens can be overlapped in time)

lower order Interleaving :

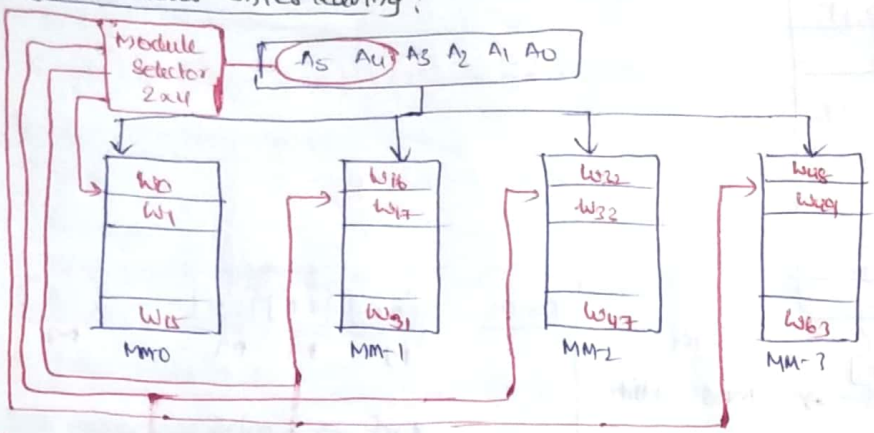




Intf

☺ faster Access

Higher order interleaving:



∵ Spatial locality
Page fault reduced

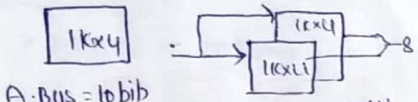
Memory Expansion:

- Num of Basic capacity IC's = $\frac{\text{Target capacity}}{\text{Basic size}}$
- Resultant memory is organized in rows & cols.
- For 2^m rows we require $m \times 2^m$ decoder is used to provide Enable lines.

Memory Expansion

Data Expansion:

B: 1Kx4 T: 1Kx8

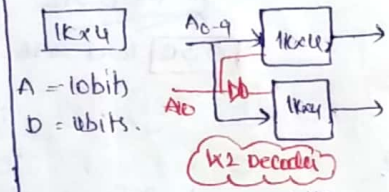


A-Bus = 10 bit
D-Bus = 4 bit

A-Bus = 10 bit
D-Bus = 8 bit

Address Expansion:

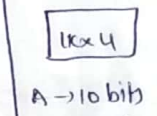
B: 1Kx4 T: 2Kx4



A = 10 bit
D = 4 bits.

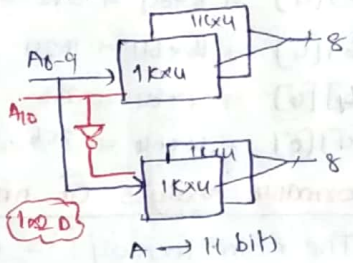
A → 11 bits
D → 4 bit

B: 1Kx4 T: 2Kx8



A → 10 bit
D → 4 bits

Data & Address Expansion



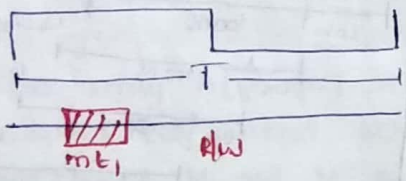
A → 11 bit
D → 8 bit

D-Ram Cells:

- D Ram cells are arranged in 2D (R & C)
- In one refresh Entire row get refreshed
  ie with m-rows and row refresh time t_1 then

Refresh overhead = mt_1

- Let refresh time cycle is 'T' then $T - mt_1$ is used for memory read/write opens.



% time used for refresh = $\frac{mt_1}{T_1} * 100$

% time used for r/w opens = $\frac{T_1 - mt_1}{T_1} * 100$.

RAID organization:

- Redundant array of independent disks
- It is adopted for secondary memories to improve performance
- The secondary mem is divided into Independent disks and can be accessed in parallel
  Thus Raid is Increasing Data transfer rate and also provide fault tolerancy

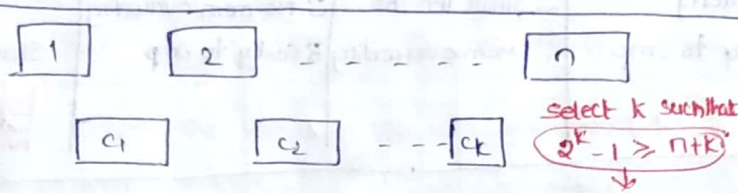
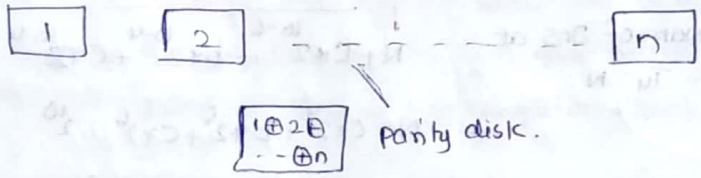
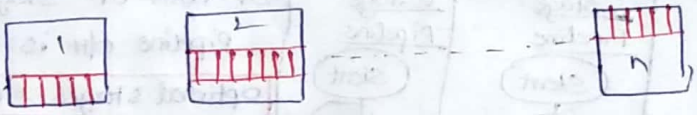
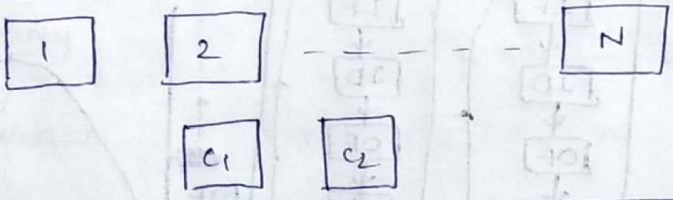
- Raid-0: Improve data transfer rate
  No fault tolerancy
- Raid-1: Disk mirroring
  1 Disk failure tolerant
- Raid-2: Hamming code
  1 Disk failure tolerant
- Raid-3: Parity disk
  1 Disk failure tolerant

Raid-4 : Interleaving parity disk (Byte level)
1 Disk failure tolerant

Raid-5 : Interleaving parity disk (Block level)
1 Disk failure tolerant

Raid-6 : 2 parity disks
2 disk failure tolerant

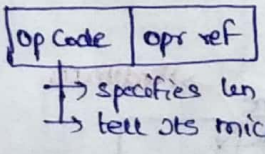
Note :

Raid	Organization	Remarks
0		$D = N * [\text{non-raid}]$ - No Fault tolerancy
1		$D_{Read} = N * (\text{Raid-0})$ $D_{write} = \frac{N}{2N} * (\text{Raid 0}) \Rightarrow \frac{\text{Raid-0}}{2}$ - Disk mirroring - 1 Disk fault tolerant
2	 select k such that $2^k - 1 \geq n+k$ $\Rightarrow x$ $k = \min(x)$	$D = \frac{N}{N+K} * (\text{Raid-0})$ - Hamming Code - 1 Disk fault tolerant
3		$D_R = \text{Raid-0}$ $D_W = \frac{N}{N+1} (\text{Raid-0})$ - 1 Disk Fault tolerant - parity disk
4		- Byte level parity Interleaving - 1 Disk Fault tolerant
5		- Block level parity Interleaving - 1 Disk Fault tolerant
6		- 2 parity disks - 2 Disk failure tolerant - Not yet implemented - More statistics read

Note : Block level Interleaving is better than byte level interleaving
If Block size = Disk size then Raid-5 $\Rightarrow$ Raid-3

Instructions, Addressing Modes and Instruction pipeline :

Instruction : Basic Component in a program



Note :
Ins Arch Contains 2^n Ins
where n = opCode bits

Classification of Instructions

Based on opcode

- Data transfer / Copy Ins
- Arithmetic Ins
- Logical Ins
- Conditional Ins
- Iterative Ins
- (Special Ins) M/c Control Ins → deals with m/c rather than data

Based on oprnd:

num of oprnd

4-Address Ins
- No pgm Counter
- Self Seq Ins
- Ex:- ADD 100, 200, 300, 400 (400 → Addr of next Ins)
- ∵ Simplicity
- ∴ lengthy

3-Address Ins
- Pgm Counter
- Ex:- ADD, 100, 200, 300 (+)
- ∵ Simplicity
- ∴ Costly to Imp

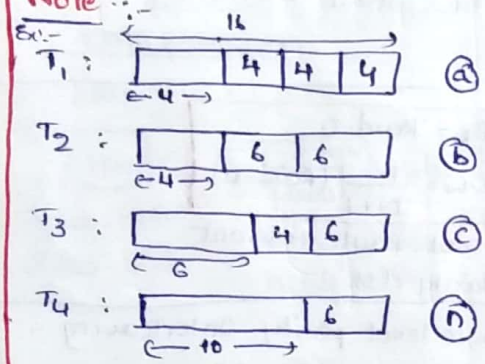
2-Address Ins
- Pgm Counter
- Ex:- ADD 100, 200 (+)
- ∵ small len Ins
- ∴ Mem overwriting

1-Address Ins
- Pgm Counter
- Accumulator ✓
- Ex:- ADD 100 (Res stored in Acc)
- ∵ No mem overwriting
- ∴ Costly to Imp

0-Address Ins
- Pgm Counter
- Acumulator
- stack pointer
- Ex:- ADD → Stack addr Ins

Note :-

Ex:-

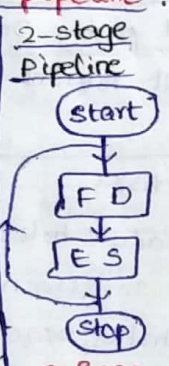


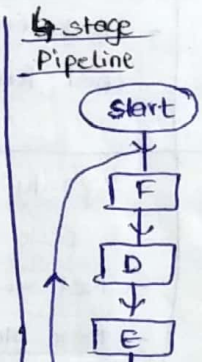
Num of Ins at T_4 N = $N + C*2^{10-6} + b*2^{10-4} + C*2^{10-4} = 2^{10}$

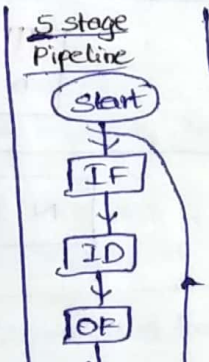
(Num of Ins; sacrificed; Total possible Ins set)

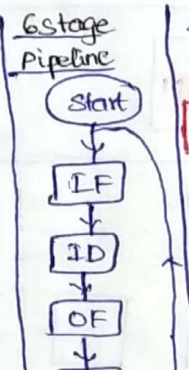
$= N + C*2^4 + b*2^6 + C*2^6 = 2^{10}$

Instruction pipeline :



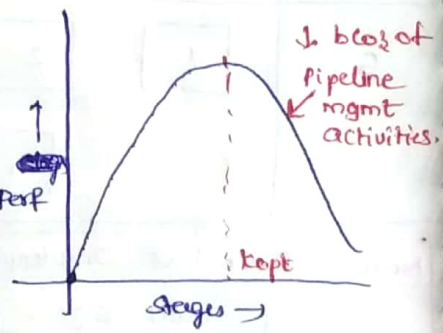




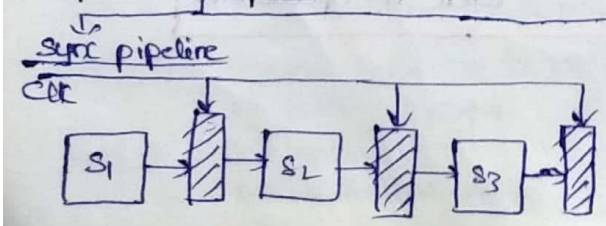


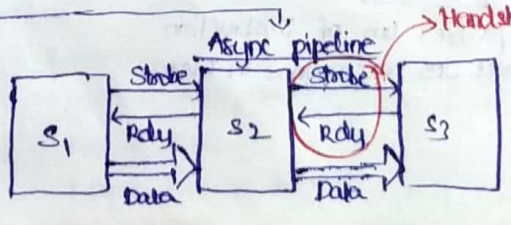
If num of stages > 6 then Pipeline o/H is more. So,

optimal stages = 6



Types of Pipeline



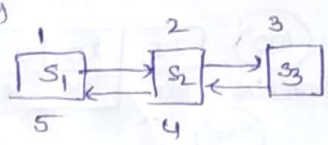


Types

Linear pipeline
- Feed forward pipeline
- Moves only in forward direction

Non-linear pipeline
- Feed forward and feedback pipeline
- Reservation table : which stage is using desired stage

	1	2	3	4	5	Clks
S1	X				X	
S2		X		X		
S3			X			



Performance of Instruction pipeline:
- Given by speedup factor $(S) = \frac{\text{Time without pipeline}}{\text{Time with pipeline}}$
- The pipeline produces optimal performance if the ins are independent.

Ideal case for speed up factor = Num of stages (K) (η = 100%)

- For n instructions the k-stage pipeline will consume $K+n-1$ clocks.
- The clock period of pipeline = Max (Si delay) + Buffer delay.
- For the first ins, Non pipeline system will give better performance than pipelined system however, on Avg the pipeline system will give better performance than Non pipeline system
- In General, the pipeline is expected to complete 1 ins/clk on average

$CPI_{avg} = 1$

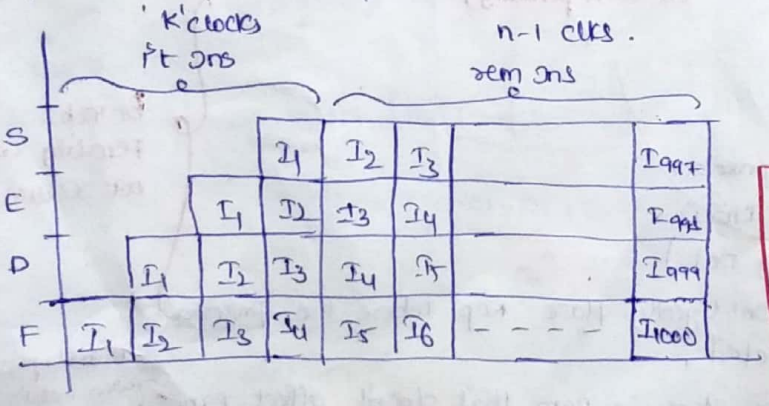
- The pipelining will not reduce individual ins time (The ins has to do same amt of work whether it is pipelined or non-pipelined)
- The dependencies among ins require reorganization of ins pipeline and reduces the performance of ins pipeline

Dependencies

Data dependency
Caused by
Arithmetic instructions
↓ sol
Ins rescheduling
Operand forwarding

Control dependency
Caused by
Conditional instructions
↓ sol
Delay load
Prediction dependencies

Structural dependency
Caused by
Resource limitation
↓ sol
Resource duplication

- 

Speed up factor = $\frac{n * K * clk}{K+n-1}$ (ins)

If $n >> K$ then

$S = \frac{nK}{n} \Rightarrow K$ (ideal)

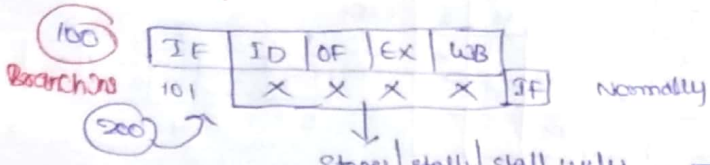
$CPI_{avg} = \frac{T_n}{n} = \frac{K+n-1}{n} \Rightarrow \frac{n}{n} = 1$ (ideal)

Note:

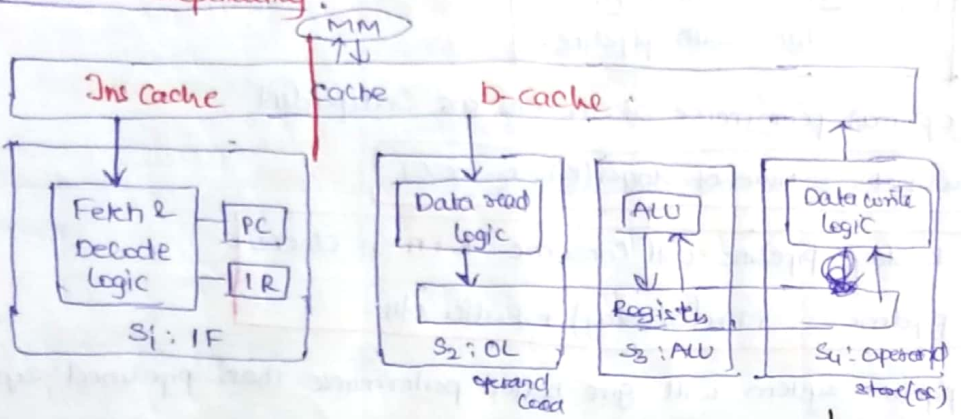
For certain problems

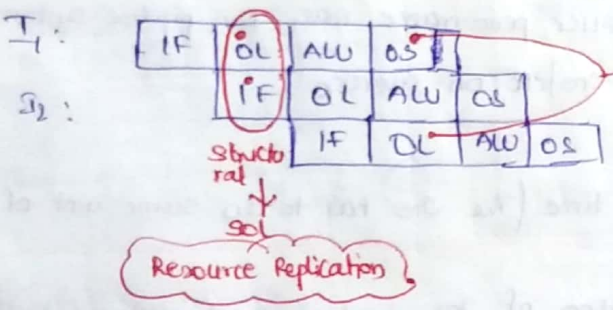
Total clks need to Complete a process = Num of clks for 1st ins + Sum of clks of unique stage except 1st ins

Dependencies / Hazards



(1) Structural dependency:

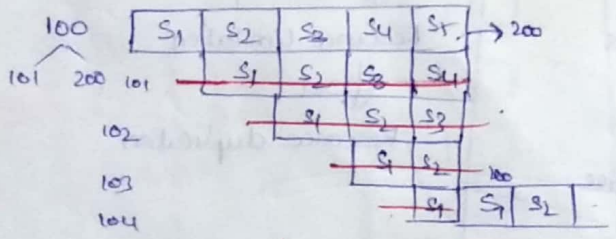




Many reasons are possible for structural dependencies
- Due to registers
- Due to cache

(2) Control dependency:

caused due to (i) Conditional branches
(ii) Unconditional branches
(iii) Function calls / return
(iv) Pgm control that may effect PC



Waiting time (Branch penality)

↓ solution

(i) flushing (clearing the contents of pipeline)

(ii) stalling. (In case of branch ins the processor should stall some time to know whether it is branched or not)

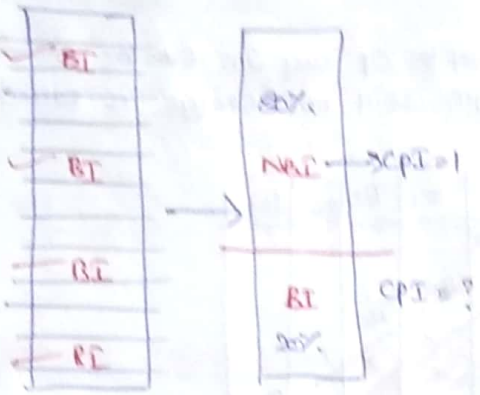
(iii) Nop (No operation: compiler should intelligently place Nop where the processor do work but it was not effected.

(iv) Reschedule and rearrange of some steps in pgm that doesn't affect pgm

Branch penality was not reduced

Branch penality was reduced to some Extent

*) Consider a pgm

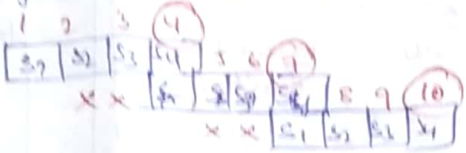


If Target Addr known after stage 2

CPI = 2

Avg CPI = 0.8(1) + 0.2(2) ⇒ 1.2

If Target Addr known after stage 3



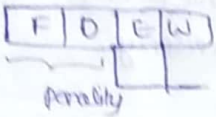
CPI = 3

Avg CPI = 0.8(1) + 0.2(3) = 1.4

- Branches

Unconditional

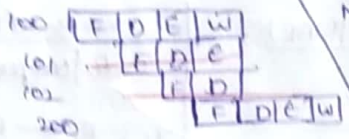
Jmp 200.
- Target Addr known after decode phase.



Conditional

True

If (a>b) Jmp 200.



False

No penality and Execution flows as it is

Effective CPI :

n → Total num of Ins
p → prob that a Ins is Conditional branch
q → prob that a Cond branch is true
m → num of stages after which the target addr is known

∴ Total Cond Ins = np

Total Cond Ins that are true = npq

Not branched Ins = n − npq

$$\text{Avg CPI} = \frac{npq*m + n(1-pq)}{n} = mpq + 1 - pq.$$

$$\boxed{\text{Avg CPI} = 1 + pq(m-1)}$$

Ways to avoid stall cycles in Control dependencies :

1) Branch prediction and speculative Execution (Guess)

100 → 200 (Branch), 101 (NB) → Flushing involved if Branching taken

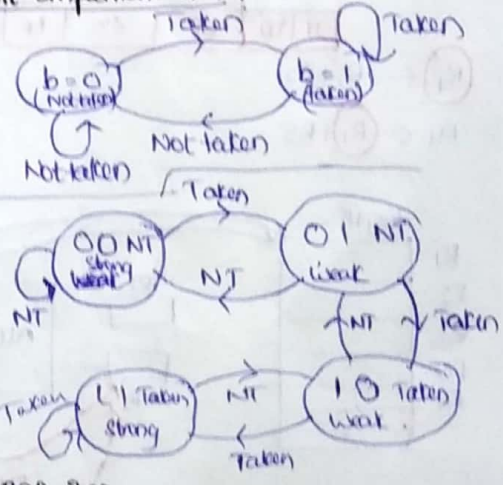
→ Branch always taken. (100, 200, 201, 202, 101) ✗ if NB taken

→ Branch never taken (100, 101, 102, 103, 200) Flushed if B taken

→ Dynamic prediction
  - → bit Implementation
  - → Branch prediction buffer Implementation

All are prediction based, not 100% guarantee

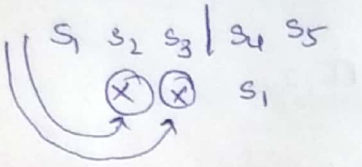
Bit Implementation :



BPB Imp

B Ins Addr	Target Addr
100	~~200~~ 101

2) Delayed Branching :

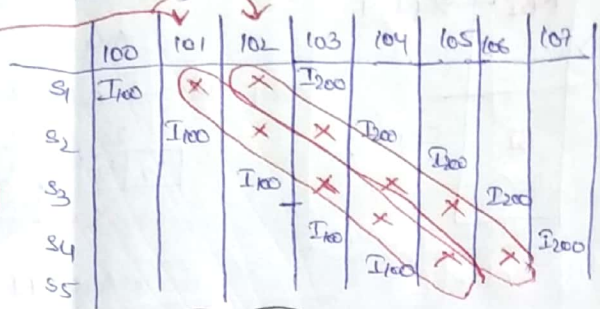


⇒ Instead of wasting two stall cycles of any Ins can be moved such that the meaning of pgm will not change is called Delayed Branching

Ex:-

98	ST	R1, B
99	LDA	A, R0
100	BZ	200

Delayed Branching



60% Stalls can be reduced A/C to research

Requirements

(i) Pgm meaning shouldn't change

(ii) Smart Compiler

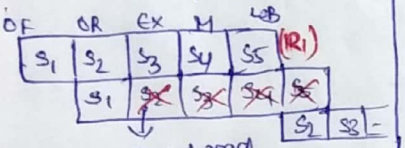
3) Data dependency :

Read after write (RAW)

- True dependency

I_i : $R_1 \leftarrow R_2 + R_3$

I_j : $R_4 \leftarrow R_1 + R_5$



Range (write) ↑ Domain (read values)

$R(I_i) \cap D(I_j) \neq \phi$

Write after read (WAR)

- Anti dependency

I_1 : $R_1 \leftarrow R_2 + R_3$ (20)

I_2 : load M, R_2 ($R_2 = 10$)

Ins are read in such a way that the loaded R_2 is read by I_1.

$D(I_i) \cap R(I_j) \neq \phi$

Write after write (WAW)

- write / output dependency.

I_1 : $R_1 \leftarrow R_2 + R_3$ (10 10)

I_2 : $R_1 \leftarrow R_5 + R_6$. (5 10)

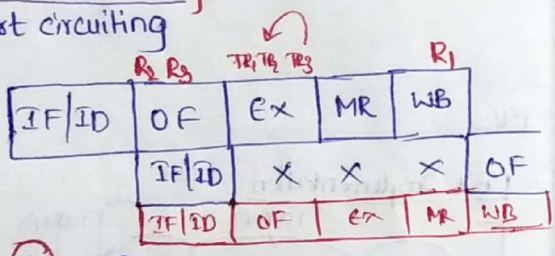
Actual: 100, 15

But: 15, 100

$R(I_i) \cap R(I_j) \neq \phi$

Operand forwarding : (Resolves RAW Conflict)

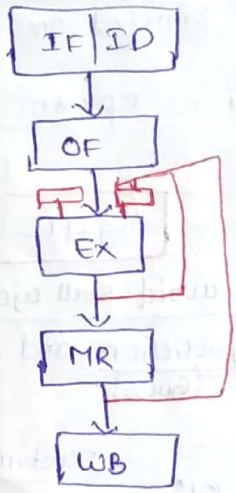
- short circuiting

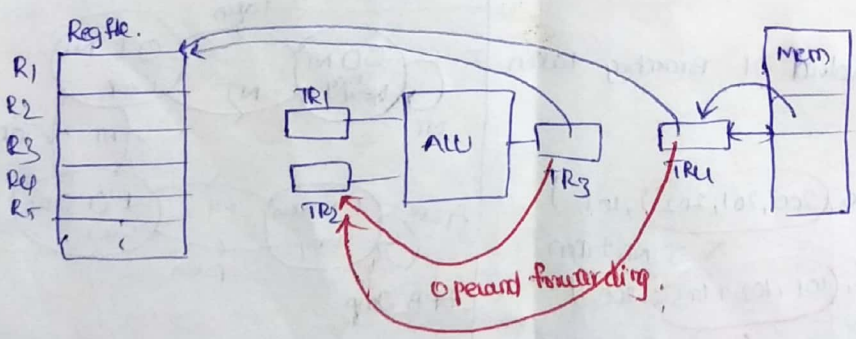


Due to operand forwarding

I_1 : $R_1 \leftarrow R_2 + R_3$

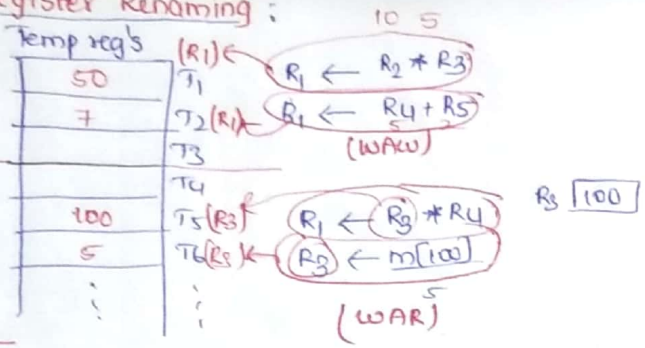
I_2 : $R_4 \leftarrow R_1 + R_5$





- To Resolve WAR, WAW → Register renaming

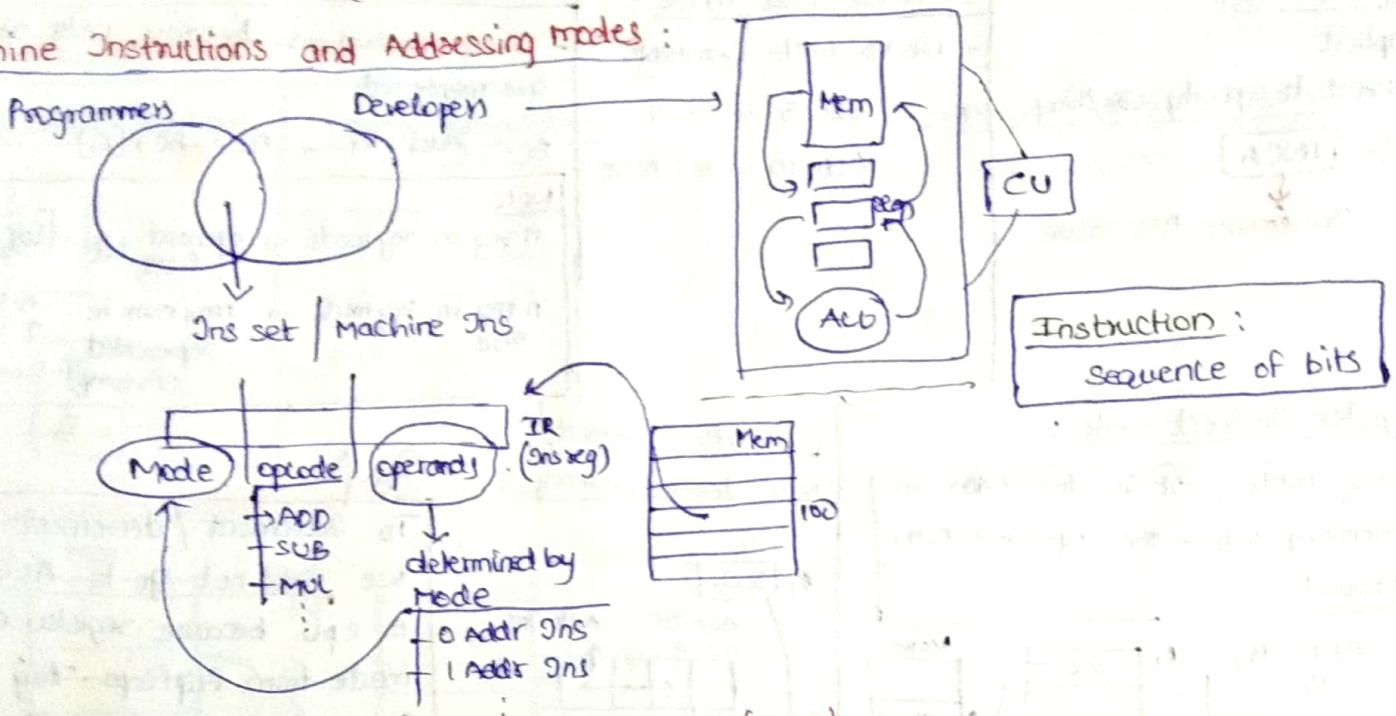
Register Renaming :



Machine Instructions and Addressing modes :

(WAR)

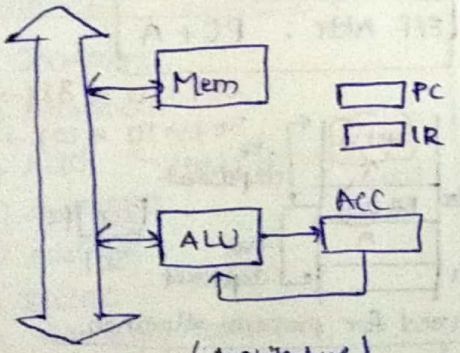
Machine Instructions and Addressing modes :



Types of cpu organizations :

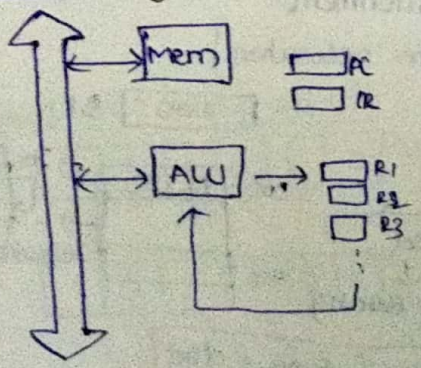
- Single accumator organization – 1 Addr Ins (ADD X) , 0 Addr Ins (INCA)
- General register organization – 2 Addr Ins (ADD R_1,R_2) , 3 Addr Ins (Add R_1,R_2,R_3)
- stack organization – 0 Addr Ins (PUSH, POP) , 1 Addr Ins (Push X)

Single Accumalator organization :



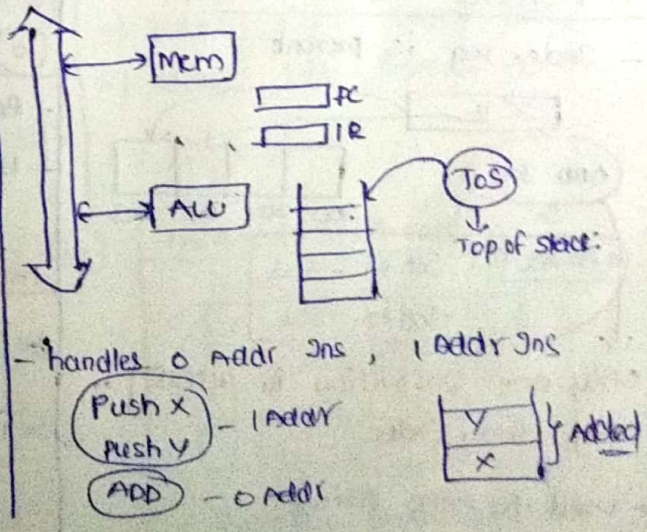
- handles 1 Addr Ins
  ADD X → AC ← AC + M[X]
- handles 0 Addr Ins
  INCA → AC ← AC + 1

General register organization :



- handles 2 Addr Ins
  ADD R_1,R_2 → $R_1 \leftarrow R_1 + R_2$
- handles 3 Addr Ins
  ADD R_1,R_2,R_3 → $R_1 \leftarrow R_2 + R_3$

Stack organization :



- handles 0 Addr Ins , 1 Addr Ins
  Push X
  Push Y – 1 Addr
  ADD – 0 Addr

Addressing modes :

∴ - To give programmers facilities such as pointers,
Counters for for loops control
indexing of data
Program relocation

- To reduce the number of bits in the addressing field of the Instruction

Modes :

1) Implied mode
- Implicit
- No need to specify anything

Ex :- INCA → Increments Acc value

2) Immediate mode :
- Deals with Constants

Ex :- LD 5 ⇒ AC = 5
ADD 10 ⇒ AC ← AC + 10

3) Register mode :
- short Instructions because only reg's are mentioned

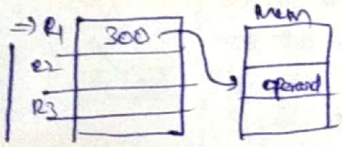
Ex :- Add R_1 ⇒ AC ← AC + [R_1]

Note :-
n reg in reg mode → operand field bits $\lceil \log_2 n \rceil$
n reg in reg mode → max num ie represented (2's comp) $2^{n-1} - 1$
n reg in reg mode → reg size

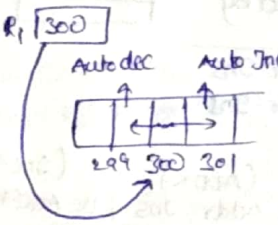
4) Register Indirect mode :

Effective Addr : It is the Addr in the memory where the operand Addr is present

Ex :- ADD R_1 ⇒ AC ← AC + M[R_1]



5) Auto Increment / Auto decrement mode :



Imp
To Increment / decrement we need not go to ALU or cpu because registers are made from flipflops. They can perform Inc, dec, load without ALU

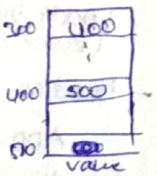
6) Direct Addressing mode :
- Absolute Addressing mode

Ex :- Add X ⇒ AC ← AC + M[X]

Eff Addr : X

7) Indirect Addressing mode

Ex :- ADD X (300)

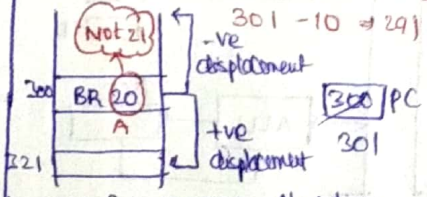


∴ 2 Memory Accesses
∴ pass parameters
Implement pointers

8) Relative Addressing mode :
- To go for next Ins how many addr should be ascended / descended

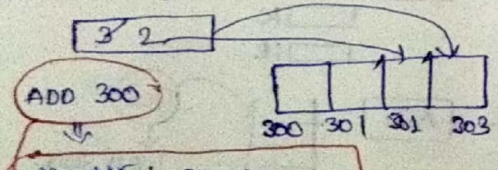
Eff Addr : PC + A

301 + 20 ⇒ 321 → +ve
301 - 10 ⇒ 291 → -ve



∴ used for Program relocation

9) Indexed Address mode :
- Index reg is present



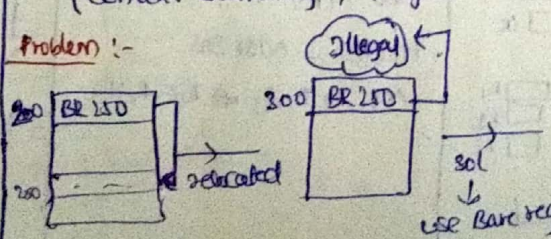
Eff addr : 300 + 3 = 303
300 + 1 = 302

only one Instruction to access any Array Index ∴

- used to access Array

10) Base register method
- Base offset register method / Base offset method
- used in multiprogramming Environment (Context switching, Program relocation)

Problem :-



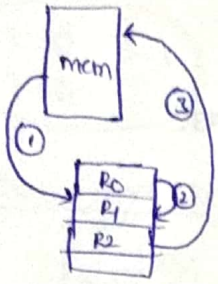
Eff Addr : Base + offset

Note :

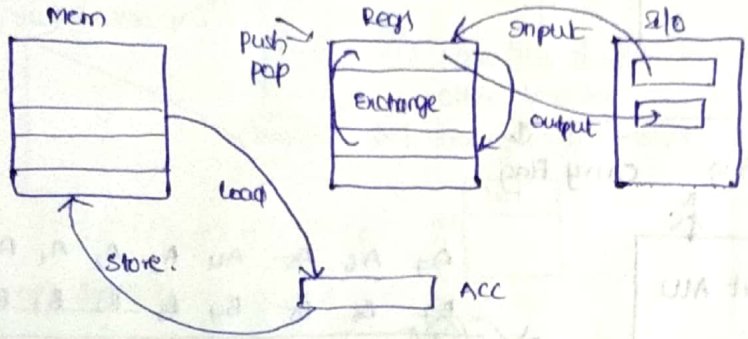
- Indirect Addressing – Pointers, Parameter passing
- Immediate Addressing – Constants
- Auto decrement Addressing – loops
- Program relocation – Based Addressing, Relative Addressing
- Indexed Addressing – Array Implementation

Types of Instructions :

- Data transfer Instructions
- Data manipulation
  - Arithmetic: ADD, SUB, MUL, DIV
  - Logical: AND, OR
  - shift: left, right
- Program Control Instructions
  - Decisions Implementation: if, for, while



Data transfer Instructions : (load, store, Exchange, Input, output, push, pop)



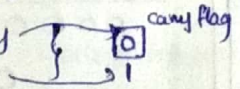
Data manipulation Instructions

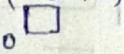
Arithmetic Instructions :

1) Increment
2) Decrement
3) Add (Int / Float (Precision is present))
4) subtract
5) multiply
6) Divide
7) Add with carry
8) sub with borrow
9) Negate (2's comp)

Logical Instructions :

1) clear (All reg bits changes to 0)
2) Complement
3) AND
4) OR
5) EX-OR (odd bit detector) (Mod 2 Arithmetic)
6) Clear carry
7) Set carry
8) Complement carry
9) Enable Interrupt → 1
10) Disable Interrupt → 0

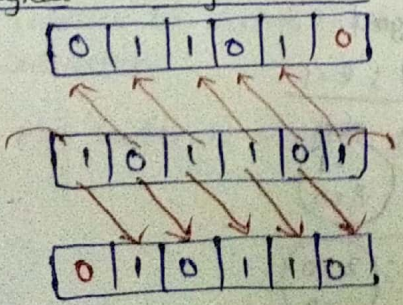




Shift Instructions :

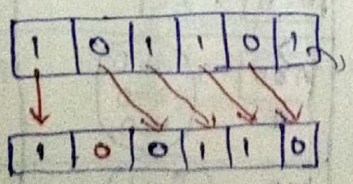
1) logical shift right
2) logical shift left
3) Arithmetic shift right
4) Arithmetic shift left
5) Rotate right
6) Rotate left
7) Rotate right through carry
8) Rotate left through carry

logical shift right and left :

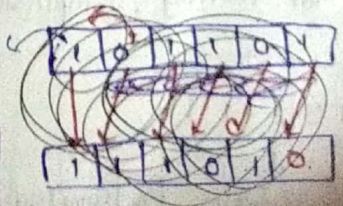


Arithmetic shift right left : (2's Comp)

- sign bit is conserved

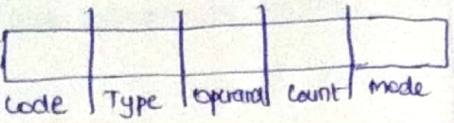


Arithmetic shift left right

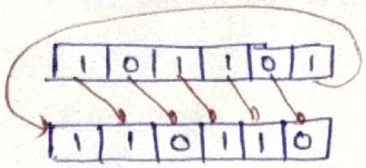


- Same as logical shift

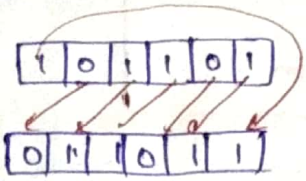
Note : 8086 Arch for shift opern



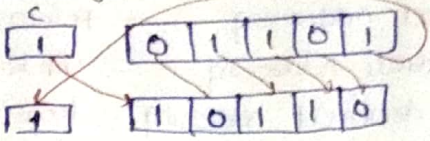
Rotate right :



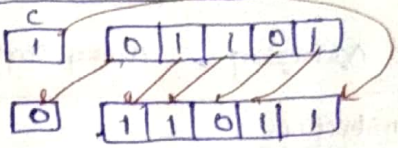
Rotate left



Rotate right through carry



Rotate left through carry :

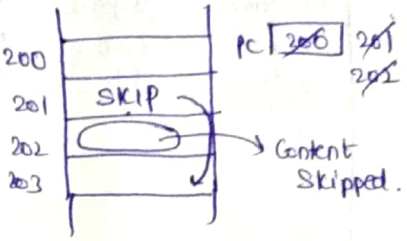


Program Control Instructions :

- It modifies program counter

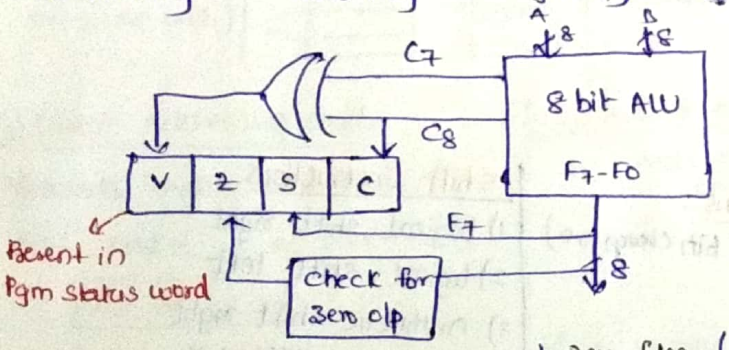
1) Branch } – deals with non-sequential flow execution
2) Jump
3) skip (skips next instruction)
4) Call } – deals with procedure calling
5) return
6) Compare } – help in making conditional decisions
7) test

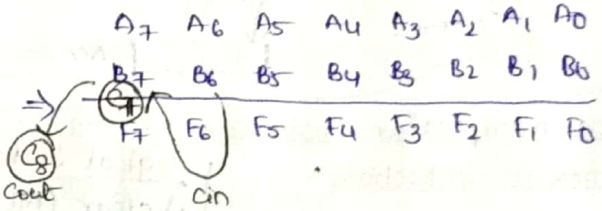
SKIP



FLAGS

Overflow flag | Zero flag | Sign flag | Carry flag.





Overflow flag ($V = 1 \rightarrow$ overflow)

Cond : $C_{in} \oplus C_{out} = 1$

sign flag ($S = 1 \rightarrow$ signed)

- F_7 gives sign for Arithmetic.

Carry flag : ($C = 1 \rightarrow$ carry present)

- detects presence of C_8
- used for testing, comparision.

Testing :-

0 1 1 0 x 1 1 1

placed in x carry flag

1 0

and tested whether it is 1 or 0

Zero flag ($Z = 1 \rightarrow$ o/p is zero)

0 1 1 0 x 1 1 1

AND 0 0 0 0 1 0 0 0

0 0 0 0 x 0 0 0

0 → res = 0, Z = 1

1 → res ≠ 0, Z = 0

→ Check for zero o/p logic

Comparision (A – B) → subtraction.

	signed	unsigned
A = B	; Z = 1	A = B ; Z = 1
A > B	; S = 0, Z = 0, V = 0	A > b ; C = 1, Z = 0
A < B	S = 1, Z = 0, V = 0	A < B ; Z = 0, C = 0

Numerical examples on flags :

If Signed representation

A = 1100 (-4)
B = 0010 (2)

Now, perform A-B

A = 1100
+ 2's B = 1110
(1) 1010 → res ≠ 0 ⇒ Z=0

Z	V	S	C
0	0	1	1

PSW ; S=MSB, C=Cout

V = Cin ⊕ Cout = 1 ⊕ 1 = 0

V=0 S=1 Z=0 → A<B

If unsigned representation :-

A = 1100 (12)
B = 0010 (2)

A-B

A = 1100
2's B = 1110
1010 (10)

Z	V	S	C
0	0	1	1

Z=0 C=1 → A>B

Note :

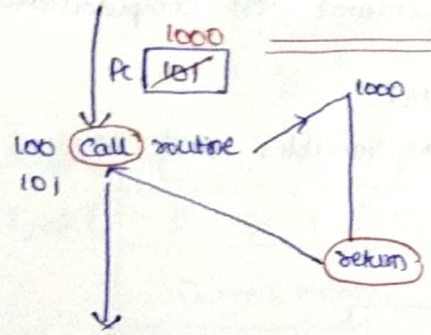
If signed

V	Z	S	Res
1	x	x	Overflow (Invalid)
0	1	x	A=B
0	0	0	A>B
0	0	1	A<B

If unsigned

Z	C	Res
1	x	A=B
0	0	A<B
0	1	A>B

Call and Return program control instruction :



Prob :- 101 Addr lost ie return addr lost

sol

(i) Save the contents in some other register ✗

(ii) storing in sub routine. ✗

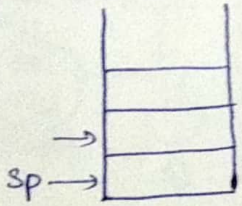
(iii) Store it in fixed memory instead of subroutine memory. ✗

(iv) use memory stack → Best sol

Passes in self recursion.

self Recursion fails

Memory Stack :

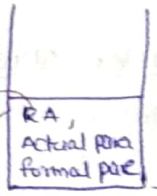


∵ only Addr are stored

Prob → No parameter passing

sol

Modifications to memory stack →



M[sp] ← PC
Sp++ ;
PC ← Eff Addr
call

Sp--
PC ← M[sp]
return

Note :-

Call & return - Ret Addr, parameters are stored

Context switching - Ret Addr, content in Regs are stored (PSW) (Pgm status word)

Interupts :

Difference b/w subroutine & Interupt

subroutine	Interupt
- Transfer of control is initiated by 'call' instruction	- It is initiated because of a signal generated
- Pgm Counter is saved on the stack	① External : (I/O devices, Timer devices, Power supply)
	③ Internal - divide by 0, Reg overflow, Protection violation, Stack
	② s/w Interupts (System call)
	- (PC, PSW, processor regs) is saved on the Context

- Effective Addr is computed by call and placed in PC

- Eff Addr is given by H/w and placed in PC

CISC and RISC

Depending on Type of Ins computers < CISC / RISC

Assembly language Tracing:

It is about calculating the contents of the registers, memory locations, and finding the space and time requirement

One ore more, key Instructions are used to decide the flow of execution and hence the effect of these key Ins on flags have to be carefully analyzed.

Note:-

- Return Addr of HLT is address of halt only

Register Allocation:

- If the Info is available in Registers then the system Consume less Computational time.
- However num of regs available in the processor are limited.
- Register Allocation is a process of assigning Registers to the variables and reusing for the subsequent allocation process

Register Allocation

- local register Allocation → Allocation consider block as a unit
- Global register Allocation → Allocation consider program as a unit

Obj of Reg Allocation: Minimizes the num of spills
Maximizes the resource usage

Belady register Allocation:

Data structure: Allocated list, Free list

Algorithm:, If a variable request a register

(i) If there is a free register Allocate it

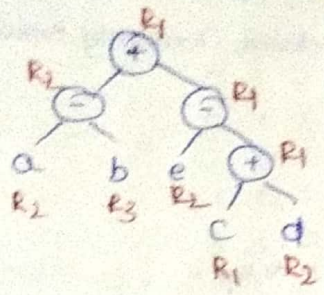
(ii) Else
find a reg that contains deadvariable
if dead var found
release it & Allocate it

(iii) Else
Identify the reg whose live var is read after long time then
a) spill it (save it in memory)
b) Allocate it.

Global register Allocation

- Uses Graph Coloring approach.

Ex:- $(a-b)+(e-(c+d))$



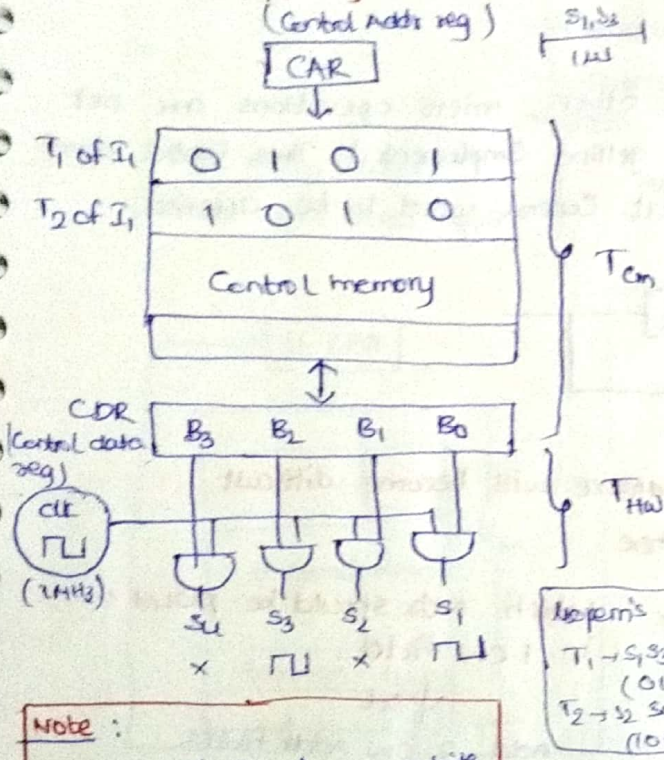
Num of reg's = 3 (min)

Micro programmed Control unit:

- Binary pattern of Control signals stored (decoded (Hup) / Encoded (Vup)) in Control memory
- Control signals are generated after accessing Control word and t/fw using

$$T_{cg} = T_{CM} + T_{HW}$$

Horizontal μ pgming:



Mopems
$T_1 \rightarrow S_1 S_3$ (0101)
$T_2 \rightarrow S_2 S_4$ (1010)

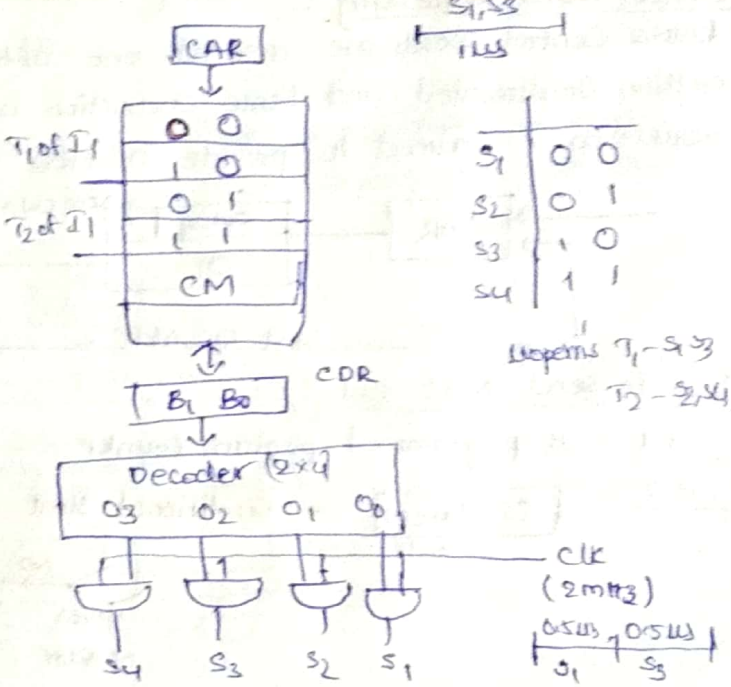
Note:
n control signals → n bits

☺ - No need of external decoders
- Degree of parallelism is high
- DOP_{max} = num of control sig

☹ Control word is too long
Control memory is not efficiently used

Vertical μ pgming:

Obj: To reduce length of Control word.



Mopems $T_1 - S_1 S_3$
$T_2 - S_2 S_4$

☺ Note:
n Control signals → $\lceil \log_2 n \rceil$ bits

☺ More flexible than Hup
reduces length of Control word

☹ Requires external decoder
- Consumes more time than Hup
- Degree of parallelism is low
- $DOP_{max} = 1$
- Faster clock rates

Advantage

↓

Diagonal μ pgming

Diagonal µ pgming :-

- It explore the relationships among the control signals.
- The control signals are divided into groups such that each group maintains mutually exclusive control signals
- Encoding is done group wise
- DOP_{max} = Total num of groups
- B/w the groups it follows HµP, within the groups it follows VµP.

100 control signals ⇒ HµP - 100 bits/CW, VµP - 7 bits/CW

18	12	25	20	15	10	Groups = 6.
5 bits/CW	4 bits/CW	5 bits	5 bits	4 bits	4 bits.	

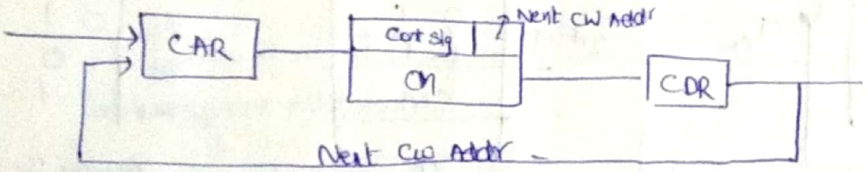
⇒ Num of bits = 27 bits/CW

∵ Dop increased
Num of bits reduced.

$DOP_{max} = 6$

Control word sequencing :

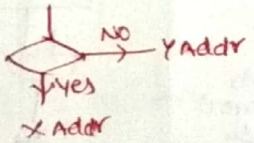
- Unless control words are accessed one after the other, micro operations are not getting implemented and hence instruction is not getting implemented. Thus control word sequencing is aimed to provide address of next control word to be accessed.



How to send Next CW ??

Sol 1 : µ programmed program counter – Maintenance will become difficult

Sol 2 : CS | NCW Addr – Conditional stmt failed here



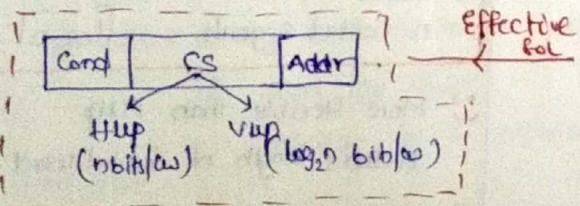
which Addr should be placed ?? in 1 CW field.

↓ sol

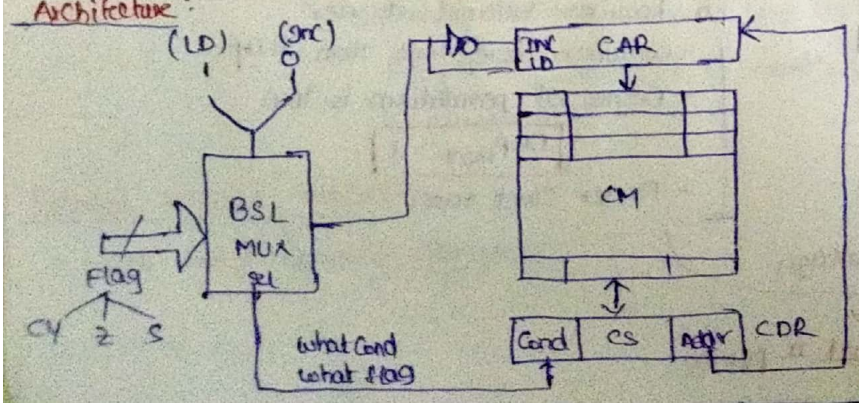
Add 2 CW Addr fields

↓

- length of CW increases
- which address field should be accessed is not known.



Architecture :



INC : Next Addr
LD : Target Addr

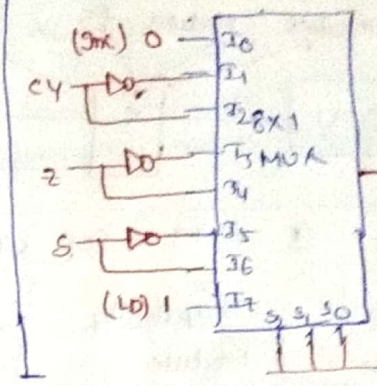
⇒ M.v wolframe µp model

for Ex:-

0 0 0	INC	- Nop (Normal Nop)
0 0 1	JNC	- cy = 0 (cy flag)
0 1 0	JC	- cy = 1 (cy flag)
0 1 1	JNZ	- z = 0 (zero flag)
1 0 0	JZ	- z = 1 (zero flag)
1 0 1	JP	- s = 0 (sign flag)
1 1 0	PM	- s = 1 (sign flag)
1 1 1	unconditional branching	

LD

Design of BSL



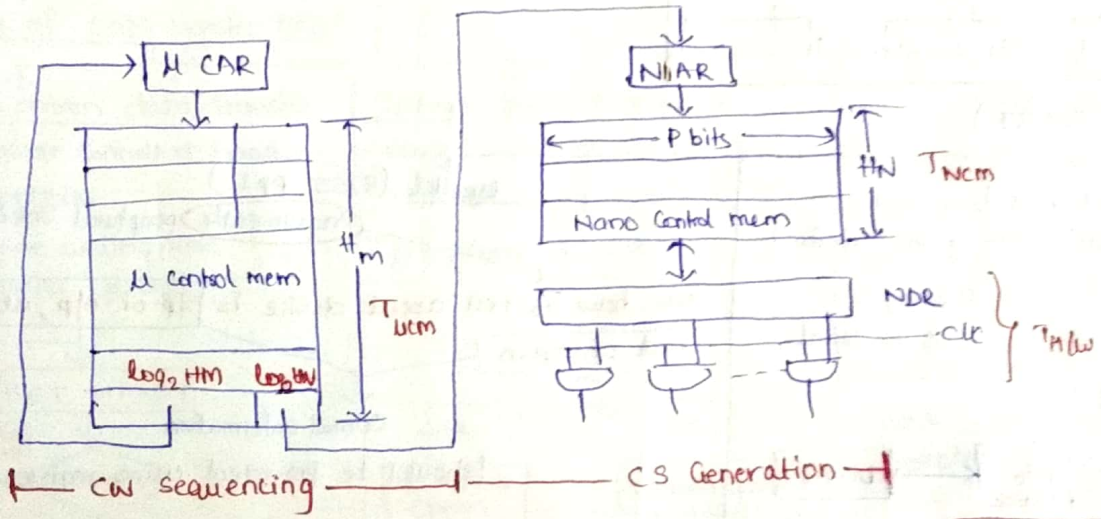
Nano programmed Control unit design :

- Uses two level Control memory
  - micro Control memory → Used for CW sequencing
  - nano Controlled memory → Control signal generation
    - → maintains distinct control signals pattern and can be reused by maintaining addr in CW of nano memory.
    - → Follows H Lup

Apps : boot strap loader, firm ware based apps.

☺ More flexibility

☹ Due to two Control memories it is the slowest control unit design



Control memory size = MCM + NCM

$= H_M \left[\log_2 H_M + \log_2 H_N\right] + H_N (P)$

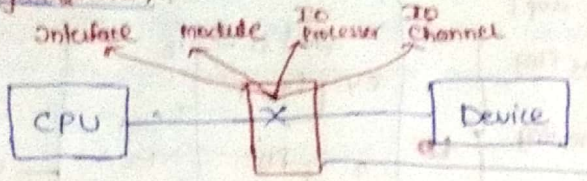
$T_{CS\ Gen} = T_{MCM} + T_{NCM} + T_{H/W}$

Points to remember :

Characteristic	CU design in Ascending order of Performance.				
Speed	Nano,	V Lup,	D Lup,	H Lup,	H/W
Response time	H/W	H Lup	D Lup	V Lup,	Nano
Flexibility	H/W	H Lup	D Lup	V Lup,	Nano
Control mem space.	H/W	V Lup	D Lup	Nano	H Lup

$\text{speed} \propto \frac{1}{\text{Resp time}} \propto \frac{1}{\text{flexibility}}$

I/o organization:



Capability → 1 < M < Iop < Ioc

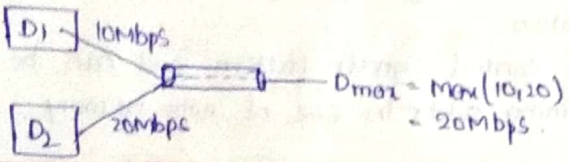
Issues
1) Addressing
2) Data transfer techniques

cpu and device is not directly connected because
(i) speed mismatch
(ii) different device driver
(iii) different data formats used by devices.

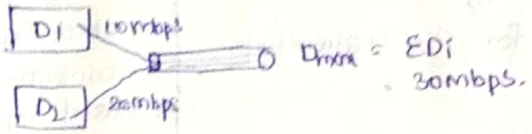
Capability →

Interface	Module	Iop	Ioc
- No Error control	Error control (Hamming code)	Implement IO ins	- Consists of Iop as one element - Types → selector channel, multiplexer channel

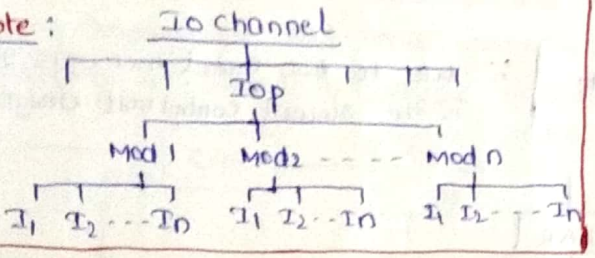
Selector channel:



Multiplexer channel:



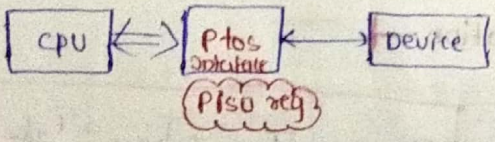
Note:



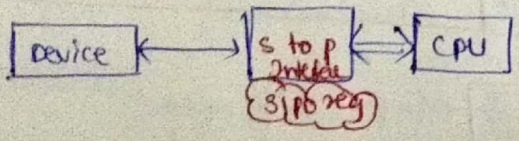
Interface (USART)

- Serial (8251 PCI) — Programmable Comm Interface
  - Convert
    - serial to //
    - // to serial
- Parallel (8255 PPI) — Programmable peripheral Interface
  - we have to tell about device is i/p or o/p, about mode of open
  - Control Information (should be provided using registers)

CPU to device:



Device to CPU:



Addressing of Io:

Goals of Io:

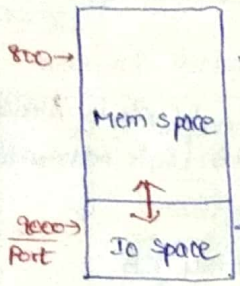
(i) Flexibility - Memory mapped Io
(ii) Expandability - Io mapped Io

Addressing of IO

Memory mapped IO

☺ Mem Addr and IO Addr are distinct
- No need of control line (IO/M)
- All mem Ins Extending to IO
- More flexibility beoz it supports direct, Indirect, Immediate --- etc

Mapping:



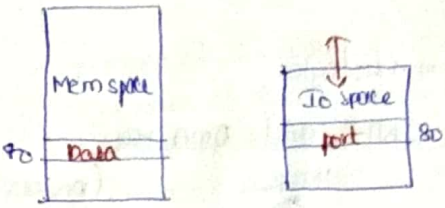
☹ - Changes in IO space Effects mem space

ΔIOs Effect MS

IO mapped IO

☺ - Δ IOs doesn't Effect MS
- More Expandability dueto Independent IOs

Mapping:



☹ - If Addr = 80
Sys doesn't Know whether it is mem space or IO space
↓ sol
Control line (IO/M) is Introduced
I O
- Separate IO Ins are needed (IN, OUT --)
- less Flexible

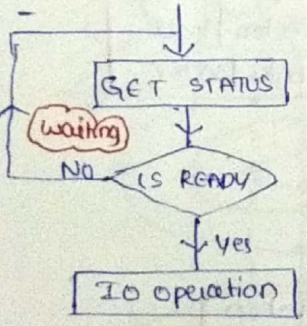
Data transfer techniques:

Obj: Data transfer tech are aimed to move data between device & processor.

Types of Data transfer tech

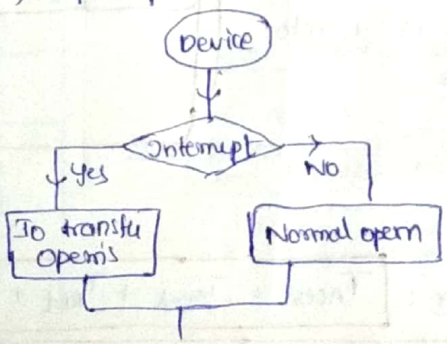
Pgm driven data transfer:

- processor Controlled transfer
☺ Simplicity
☹ More waiting time wastes cpu cycle.



Interupt driven data transfer

- Device Controlled data transfer
☺ Efficiently uses CPU than DMA
☹ Imp Comp is more.



DMA (Direct mem Access)

- In DMA, system bus is shared blw processor & DMA Controller
- Based on when system bus is returned the DMA opern mode can be:
1) cycle stealing - more waiting for DMA Controller
2) Burst - more waiting to CPU
3) Block transfer
Adv of (CS + B)
- DMA Controller (8257 | 8237)

Cycle stealing mode:

% time CPU blocked in cycle stealing mode = $\frac{t_x}{t_x + t_y} * 100\%$

t_x = word transfer time
t_y = word prep time

The sys bus is returned after each word transfer and acquired after each Ins cycle

Burst mode:

% time processor active in Burst mode Contributing = $\frac{t_y}{t_x + t_y} * 100\%$

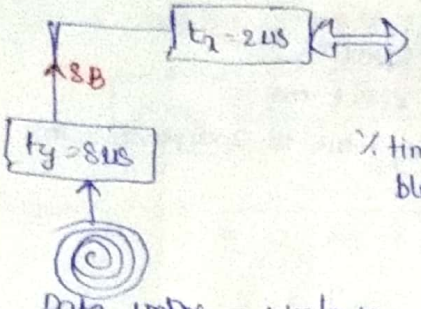
t_x = Initialization & termination time
t_y = Data transfer time.

Sys bus is returned only after Entire data transfer is completed

Block transfer mode:
- In this mode processor Initially fills Count regs & Addr regs of DMA
- on Each word transfer Count decremented & Addr Incremented
- System bus is Exchanged whenever Count is Zero

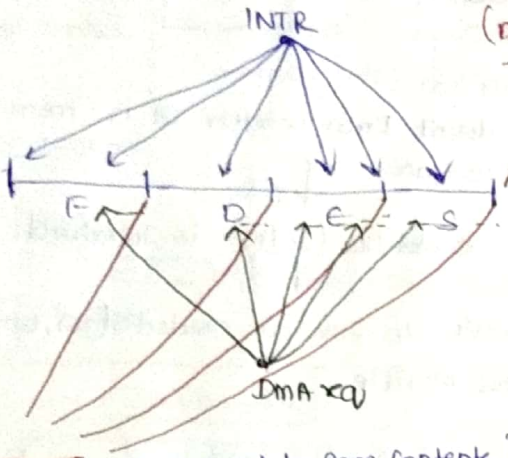
⇒ Burst mode + Cycle stealing
Within block ⇒ Burst mode
Blw Blocks ⇒ Cycle stealing

In cycle stealing mode



% time CPU blocked $\frac{2\,us}{10\,us} \Rightarrow 20\%$

Difference b/w INTR and DMA req:



DMA ACK (No need to save contents)
(DMA controller) (Ack may be given at any stage in 3ns cycle

In Block transfer mode.

Req = 16bits, Data = 29,154 KB, Byte Addressable

Min num of DMA req

CR | 64KB | ← 16 →

1 DMA req ⟶ 64KB

x req ⟶ 29,154 KB

$x = \frac{29,154\ KB}{64\ KB} = 456$ req

In cycle stealing mode.

In cycle stealing mode, 1 cycle is stealed so that we transfer 1B (Byte Addressable)

1 DMA req ⟶ 1B

x req ⟶ 29,154 KB

$x = 29,154 * 1024$

Disk:

- Semi Random access memory
- Seek time is Random Access and Rotational latency is Serial access.
- Based on how data is placed on the tracks the disk construction can be.
  - (i) Constant capacity disks
  - (ii) Variable track capacity disks
- The unit of transfer in disks is 1 sector
- Disk is addressed sector wise.

Note:
Capacity of the Disk = TSB
T = total num of tracks
S = Num of secton / track
B = Num of Byts / sector

Disk Address

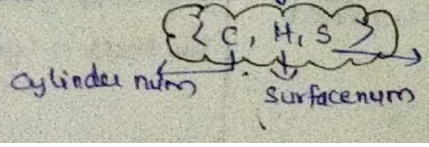
Track offset	sector offset

Time taken to Access a sector: $T_{ACess} = T_{Seek} + T_{Rot} + T_{DIT}$

Note: If position of sector is not known then Avg Rot latency is taken

Avg Rot latency = $\frac{R}{2}$

- In order to Increase the capacity of secondary memory a set of disks are arranged in disk system (Hard disk)
- In order to optimize Access time, Information is stored in cylinders (In one rotation the disk system will cover 1 cylinder capacity)
- The sectors in disk system are addressed using logical addressing



- The linear Address of a sector in disk system = $C*m + H*n + S$

$m = p*n$ $p \rightarrow$ num of rec surfaces

$m \rightarrow$ Num of sectors / cylinder

$n \rightarrow$ Num of sectors / surface track.

- The Address bits regd for a sector in disk system

Surface offset	Track offset	Sector offset

(or)

cylinder offset	sector offset per cylinder

Note:

Capacity of a disk system (C_{DS}) = PTSB

$P \rightarrow$ num of surfaces

$T \rightarrow$ Num of tracks

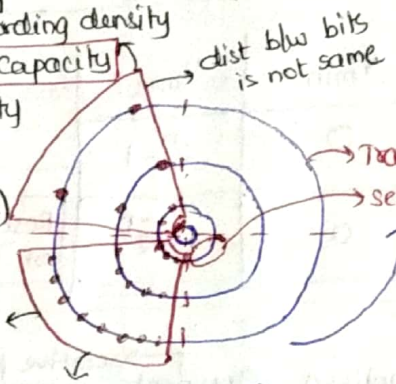
$S \rightarrow$ num of sec / track

$B \rightarrow$ num of Byte / sec

Note:

Num of sectors are same in every track

- Angular velocity
- Variable recording density
- Const track capacity
- recording density
  - max (Inner track)
  - min (outer track)

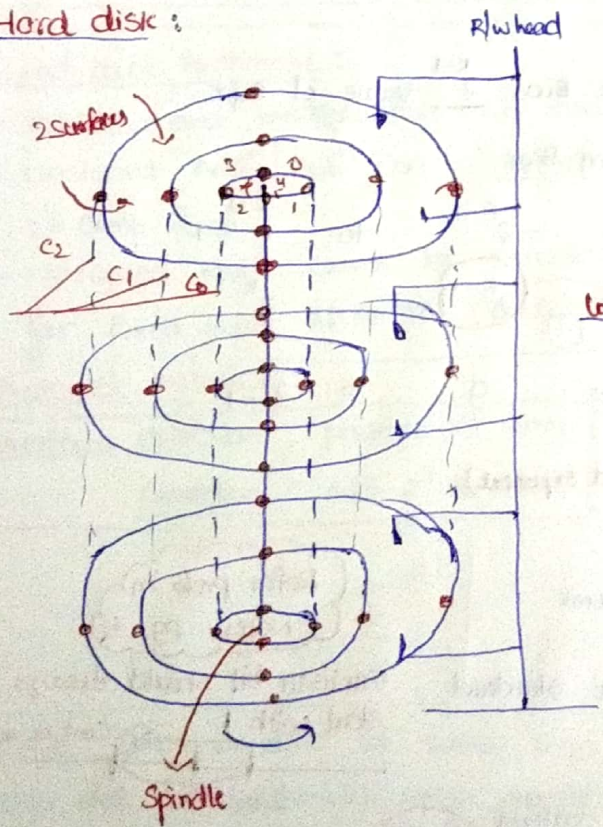


Angular velocity ($v = r\omega$)

- max (outer track)
- min (Inner track)

- Const recording density
- Variable track capacity
- Linear velocity

Hard disk:



72 sectors

Note:

Num of tracks = num of cylinders

- Sectors are addressed cylinder wise

Logical Address of sector 0 : (cy 0, sur 0, sect 0)

$\langle C, H, S \rangle \Rightarrow \langle 0\ 0\ 0 \rangle$

Logical Address of sector 71 (cy 2, sur 5, sect 3)

$\langle C, H, S \rangle \Rightarrow \langle 2, 5, 3 \rangle$

Logical Addr to Linear Addr

Linear Address of sector 71

$LA = C*m + H*n + S$

$m = p*n$ (6)

$m = 24$ sect / cylinder

$n = 4$ sect / surface.

$LA = 2*24 + 5*4 + 3$

$= 71$

Linear Addr to Logical Addr

m) LA (C

n) rem (H

rem → S

(Ex):

24) 71 (2 ⇒ C
48

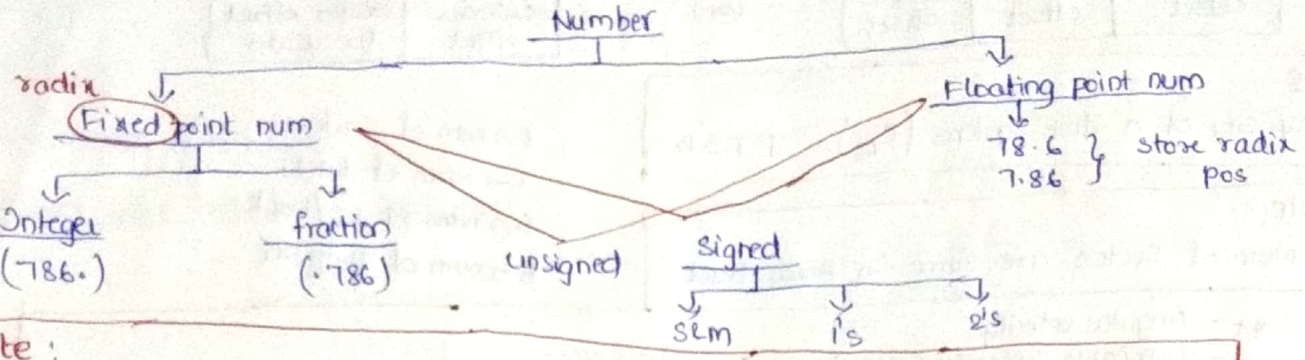
4) 23 (5 ⇒ H
20

3 ⇒ S

$\langle 2, 5, 3 \rangle$

Number Representation :

- Positional weighted system
- value Expr & Range [V_{min} to V_{max}]
  [underflow | overflow]

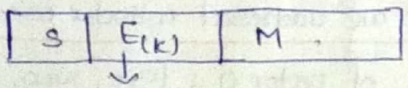


Note :

Number	value Expr	V_{min}	V_{max}	
Integer	$V = \sum_{i=0}^{n-1} a_i * 2^i$; $a_i \in (0,1)$	0	$2^n - 1$	
fraction	$V = \sum_{i=1}^{n} a_i * 2^{-i}$; $a_i \in (0,1)$	0	$1 - 2^{-n}$	Resolution of fraction = 2^{-n}

Floating point representation :

- Floating point nums denoted with mantissa, exponent → relative pos of radix
- Rep 1 : Normalized sign magnitude fraction for mantissa
  (Biased sign Exponent)



Biased Exponent with k bits we use Bias 2^{k-1} value of Expr

Value Expr

$V = (-1)^S (0.M)_2 * 2^{E-Bias}$ (default)
(Explicit)

$V = (-1)^S (1.M)_2 * 2^{E-Bias}$ (If mentioned)
(Implicit)

True Exponent → e
Biased Exponent → E
→ $E = e + Bias$??

Calculating Bias

e (k bits) : -2^{k-1} to $2^{k-1} - 1$

Bias: 2^{k-1} ---- 2^{k-1} (+)

E (k bits) → 0 to $2^k - 1$
(Biased Exponent)

Limitations of Normalized mantissa :

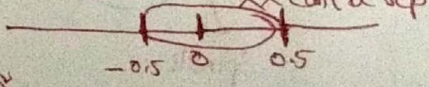
- Zero can't be represented
- Rep Expose to overflow & underflow problems

↓ sol

IEEE 754 floating point standard

Refer prob in Notes Pg - 63

mantissa bit should always start with 1



Rep 2 : IEEE 754 Floating point Standard :

- IEEE 754 gives provison for storing special values (0, ∞)

Types

- Single precision (32 bits)
- Double precision (64 bits)

- The floating point num can be stored in Implicit form or in Fractional form
- Certain Mantissa Exponent pattern doesn't denote any number (NaN)
- This specification uses base 2, and Exponent in biased form

Single precision method (32 bits)

$$\text{Value expr } (V) = (-1)^S (1.M) * 2^{E-127}$$

Note: For fraction $E=0$

$$V = (-1)^S (.1M) * 2^1 * 2^{0-127}$$

$$V = (-1)^S (.1M) 2^{-126}$$

Note:

S(1)	E(8)	M(23)	value (V)
0/1	0000 0000 E=0	000 - - - - - 0 M=0	±0
0/1	1111 1111 E=255	000 - - - - 0 M=0	± ∞
0/1	1 ≤ E ≤ 254	M = x x x - - - x	Implicit $V = (-1)^S (1.M)_2 * 2^{E-127}$
0/1	E=0	M = x x x - - - x	$V = (-1)^S (.1M) * 2^{-126}$
0/1	E=255	M ≠ 0	NaN

Fixed point Arithmetic:

- Addition and multiplication may result overflow
- unsigned Add^n / Sub^n can be extended to signed nums if operands are denoted in 2's comp form.
- unsigned Mul^n cannot be extended to signed num hence separate Algo is needed for fixed point Arithmetic (Booth's Algo)

Unsigned Arithmetic:

Overflow detection : presence of carry

Correction : Add 2^n to result

Signed Arithmetic:

Overflow detection : presence of carry in sign pos

$$Z = X + Y$$

In 1's comp / 2's comp → To detect OF:

$$Z_S + (X_S = Y_S) = V$$

$$C_{in} \oplus C_{out} = V$$

Correction : - Compliment sign bit of result
- Add $\pm 2^{n-1}$ (depends upon sign of)

Note: when operands of same sign are added, if the result is having opposite sign then the Arithmetic open denote overflows

Z_S	X_S	Y_S	V
1	0	0	1
0	1	1	1

$$V = Z_S X_S' Y_S' + Z_S' X_S Y_S$$

Booths Algorithm :

- It provide faster fixed point signed multiplication
- Uses Arithmetic right shift process.
- The sign bit is protected during the competition
- Algorithm shifts the partial sum and hence there is no need to store partial Product
- The Algo support reusing of multiple register and partial sum register for storing the result.

Booths Notation :

- It is used to denote signed nums. It is similar to that of 2's comp notation.
- The multiplier (Q) can have one or more Complete blocks (1 Add^n + 1 sub^n) of 1's and atmost one partial block of 1's (1 sub^n).

Ex:-

Num	Complete block	Partial block	Add^n	sub^n
00 1 1 1 0 1 1 1	2	0	2	2
000 1 1 1 1 1 1 1	1	0	1	1
1 1 1 1 1 1 1 000	0	1	0	1
1 1 1 000 1 1 1	1	1	1	2

For Complete block : 1 Add + 1 sub
For partial block : 1 sub

Booth's recording pattern :

- Used to find num of arithmetic & num of shift operns.

Pattern :

0	0	0
0	1	-1
1	0	1
1	1	0

We should move from right to left

Ex:- 1 1 1 0 0 1 1 0 (PB, CB)

Pattern : 0 0 -1 0 1 0 0 -1
7 6 5 4 3 2 1 0 pos

Arithmetic : 1 CB - 1 Add, 1 sub
1 PB - 1 sub
1 Add, 2 sub

Shift operns = 0 + 3 + 5 = 8 ✓

Ex:- Booths recording pattern for -54

-54 ⇒ 2's Comp
⇒ -64 + 8 + 2 ⇒ 1 0 0 1 0 1 0 0

-1 0 1 -1 -1 -1 0
6 5 4 3 2 1 0

shift operns ⇒ 1 + 2 + 3 + 4 + 6
⇒ 16

Arithmetic ⇒ 2 CB ⇒ 2 Add, 2 sub
1 PB ⇒ 1 sub
2 Add, 3 sub

Ex:- 13 * -6 → 0110 → 1010 2's,
1101 * -1 -1 -1 0

1 0 1 0 0
-1 -1 -1 0 Rec pattern

1101 ⇒ 2's.
0011

0 0 0 0 0 0 0 0
1 1 1 0 0 1 1 x
0 0 1 1 0 1 x x
1 0 0 1 1 x x x
1 1 0 1 1 0 0 1 0

2's comp of -78

Leftovers

- floating point Arithmetic
- Tape memory
- Interrupts. - DMA working

Interrupts :

- Interrupts are highly Async (not known when to occur)
- edge triggering is prefered than level triggering because edge triggering is less sensitive towards noise

Processor	Device
When to recognize ??	When to Interrupt ?? → Sol → Async (so any time)
How to recognize ??	How to Interrupt ?? → Sol → Change the INTR signal at the pin.
What to do ??	
How to resolve ?? multiple Interrupts	

At processor phase :

(I) When to recognize ?? → Sol → At the End of Execution of current Ins cycle

(II) How to recognize ?? → Sol → By reading INTR flag status. We can recognize Interrupt

INTR → 1 → Intr raised
INTR → 0 → Intr not raised.

(III) What to do ?? → Sol → Save ret addr, Contents and transfer Control to ISR (Intr service routine)

Where is ISR ?

- Vector Interrupt ← 1) use device Info
- Scalar Interrupt ← 2) use default addr

(IV) How to resolve multiple Interrupts ??

Sol:- use priorites.

Interrupt service

Pended mode

- If we store all the Interrupts and serve Interrupts a/c to priorities

Non-pended mode

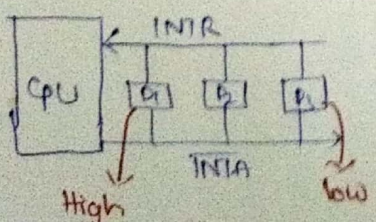
- It simply ignores least priority Interrupts.

How to assign priorities

Static

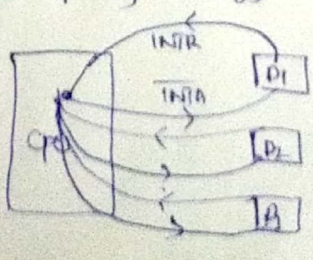
- priorities can't be altered
- starvation to least priority Interrupts
- uses Daisy chain method.

Position → priority



Dynamic

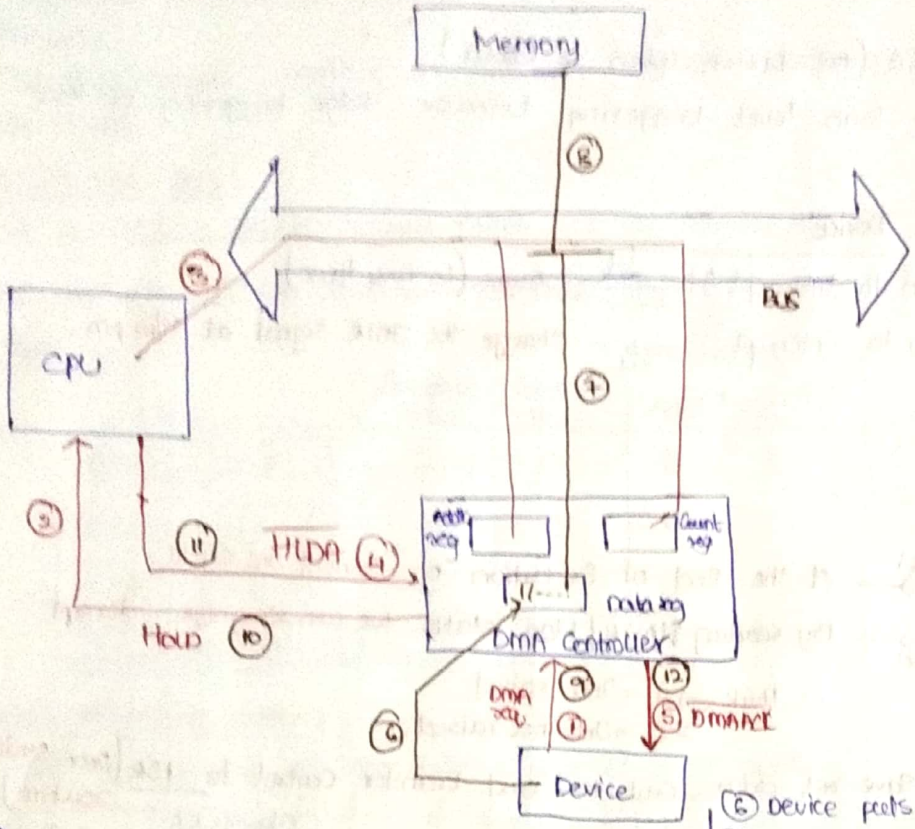
- Priorities can be altered
- served Interrupt priority is made lower than the lowest priority Interrupt
- No starvation to least priority Interrupts
- uses polling strategy



Note :

Interrupt can interrupt the system but interrupt cannot be interrupted.

Topic # 2 : How DMA works ? :



(1) Device raises DMA req
(3) DMA Controller holds CPU for system bus (Hold)
(3) CPU initializes AR and CR
(4) CPU Ack to Device (HLDA)
(5) DMA Controller Ack to Device (DMA ACK)

(6) Device puts its data in DR
(7) DR sends data to bus
(8) Bus sends data to memory
(9) Device release DMA req
(10) DMA releases Hold req
(11) CPU releases HLDA
(12) DMA releases DMAACK

X —— The End —— X