Computer Organization

CA: Conceptual Design & operational structure
- CPU Design
- Instns
- Add. Modes
- Data format

CO: Physical Device interconnection
- IO organization
- Mem organization
- Pipelining

→ System Bus
- Addr Bus (unidirectional)
- Data bus (Bidirectional)
- Control bus (Every individual link is unidirectional)

→ Acc Based Arch | Reg Based | Complex

- One of the i/p is acc
- Both i/ps are from Reg
- I/P may from reg or MM

→ Stack based arch has i/p from stack.
→ O/p must be stored in accumulator

length of PC = length of MAR = PA
length of MBR = length of Acc = word size
If virtual memory is use
length of PC ≥ length of VA
length of MAR = len PA
Size of address bus of CPU = size of virtual address.

32-bit arch: word size: 32 bits
i/p size to ALU: 32 bits

64-bit arch: word size: 64 bits
i/p size to ALU: 64 bits

Micro Operation: Smallest indivisible operation.
- Can be performed in 1 CPU cycle

3 types:
- Arithmetic
- Logic
- Shift

Logical (0 added)
Circular (shifted bit added)
Arithmetic (signed bit must be same after shift)

- So in Arithmetic right shift added bit is sign bit.
- So In Arith. left shift, it is permitted only if sign remains same. Added bit is 0.

Note: (+7 >> 1 = 3 ; −7 >> 1 = −4) in 2's comp form

Instructions:

Types:

4-add instn
→ 4th add points to next instn.
→ instn need not to contiguous
Disadv:
- Relocation is difficult
- Large instns

3-add instn
→ add to next instn field is removed and PC is added.
→ Disadv of 4-add instn are removed

2-add instn
→ One of the i/ps acts as destination
→ Smaller instn
Disadv:
→ large no of instns will be needed for a given program comp to 3 add instn.

1-add instn
→ 1 op → add field
→ 2nd op – implied (accumulator) or implied with instn
Disadv:
More no of instn for given program

0-add instn
→ No add is mentioned.
→ Generally used in stack based architectures.

Multiple instn support:

If m-add & n-add instns are supported (m > n) then find no of unused m-add instns and multiply to get req n-add instns.

→ whenever it is said a system supports k-add instns then min no of such instn must be 1 (not 0).

→ Fixed length instn ⇒ variable length opcode
variable length instn ⇒ fixed length opcode

Memory latency: It is time blw initiating a req for byte or word in memory until it is retrieved by processor.

Memory cycle time: It time blw start of one mem access and time at which next access can be started.

Memory Access time: amount of time to transfer a byte/word from CPU to RAM (or) RAM to CPU.

Instn Cycle: All the phase in instn together form instn cycle.
It is divided into 2 cycles:
fetch cycle: Instn fetch
execution cycle: remaining phases

Addressing Modes

→ Effective add is add of operand.

Mode	Description	points (adv & disadv)
Implied	Location of operand is implied within the opcode.	~~→ size of operand is limited~~
Immediate	Content of add field itself is the operand	→ used to initialize registers → with mode operand range will be small
Direct (or) Absolute [ ]	add field is MM add of operand	→ suitable for accessing static variables.
Indirect @	add field contains MM add which contains add of location where operand is present.	→ used for implementing pointers, parameter passing → Req 2 MM access ∴ slow ∴ not suitable for pipelining
Register Direct (or) Register mode	add field has register ref which has the operand	→ It is fast → Relocation is not possible.
Register Indirect	add field specifies reg which contains effective add	→ Here indirection is obtained with shorter instn and also instn is faster. → It is bit slower (almost same) compared to direct mode but instn size is small.
Auto Inc / Auto Dec (R)+ -(R)	It is variant of reg indirect Content of reg (effec add) is automatically inc / dec Inc is post inc Dec is pre dec	→ Used for accessing table of content (array) → The val to be inc is implicity determined by size of operand accessed
Indexed mode (or) Index reg mode	Effec add = add field content + (index * w) Implementing this requires 2 instns i) ind reg ← w * i ↳ array index ↳ ~~cont~~ ele size. ii) fetch operand	→ Used to access arrays → Relocation is not possible → 2 instns req
Scaled mode	It is modified version of indexed mode EA = add field + ind reg * scaling factor Here we just change the way EA is computed	→ only one instn req ∴ faster than indexed mode → Used to access arrays → Relocation is not possible.

Mode	Description	points.
Index, base reg mode (or) Based Indexed	Here we have 2 reg (ind & base)	→ It permits relocation → supports position independent code (PIC)
Index, base + displacement mode	used to access member within records EA = (index * w) + base + d d → displacement	→ permits relocation → Supports PIC
Basedreg mode	EA = base reg + add field ↓ offset	→ used for inter segment jumping → permits relocation. → supports PIC
PC Relative	add field contains offset of branch target instn from PC (i.e., next add instn) (can also be -ve) In other words add field has (no of instn to be skipped) * (size of instn)	→ Reduces instn size (∵ only offset is given) → Supports only intra segment branching. → Supports PIC. → permits relocation

Classification of addressing modes:

- Non-computable: Finding EA doesn't req and computations (1st six modes)
- Computable: Finding EA req computations (remaining modes)

→ while solving problems, remember to used value in PC as next instn add.

CPU cycle: Amount of time in which one μ-op can be performed

CPI: no of cycles req per instn

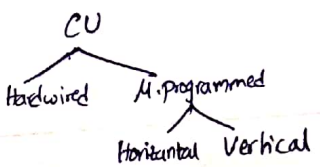
Execution time of an instn = CPI × cycle time.

Control Unit Organization

CU: generates control signal to all components

Control variable: It is name given to a control signal

Control word: Collection of all control signals



Hardwired CU:

→ Control logic is implemented with gates, ffs etc.

→ A table is created (cycles x instn) and logic is designed accordingly. For each control signal a boolean exp in terms of (T & I) is developed.

→ Timing generator is reset after every instn.

Microprogrammed CU:

→ Adv: Fast

→ Disadv: not flexible to modify

Microprogrammed:

→ Here control memory is used for storing control words.

→ CAR: stores address of loc in control memory

→ CDR: stores a word of control memory.

adv: flexible to modify

Disadv: slower than hardwired.

Each location of control memory is of form

Control signals	Mux select	nxt add

Based on how control signals are stored we have 2 types of μ-programmed CU:

(i) Horizontal μ-prog. CU:

Every control signal has one bit

(ii) Vertical μ-prog CU:

Control signals are divided into grps such that only one signal will be active in any group. So we encode this and store in control word.

And decoding is require for execution.

no of loc in control mem = (no of inst^{n}s) x (no of μ-inst^{n}s per inst^n)

n-control signals - exactly $1 - \lceil \log_2 n \rceil$ bits

n-control signal - atmost $1 - \lceil \log_2 (n+1) \rceil$ bits

Note:

Horizontal	vertical
More control mem	less control mem
faster	slower.
no need of decoder	decoder is needed

Nano-Programmed CU:

→ Inst^n may have similar μ-inst^{n}s. Here a memory is dedicated for storing all unique control signals.

→ In control signal field of control mem we store address which points to a location in nano memory where inst^{n}s are stored.

	RISC	CISC
inst^{n}s	less, fixed length	more, variable length
Add modes	less (only Direct AM for IO)	more
CU	Hardwired	μ-prog
CPI	1	>1
operation	Reg to Reg	Reg to Mem / mem to reg
reg availability	more provider reg window → ∴ efficient param passing	less no reg window
pipelining	more compatible	less compatible
compiler design	easy	difficult
programmer overhead	more	less

IO Organization

→ Need for IO interface

(i) Signal conversion
(ii) Data format conversion
(iii) Synchorization
(iv) Management & Control

→ Every IO device has an address.

IO bus also has add, data, control buses

→ 3 types of bus organizations:

(i) Seperate bus for IO & Memory:

→ costly organization
→ Generally used when seperate IOP is available.

(ii) IO Mapped IO (common add & data bus, seperate control bus)

(or) Port Mapped IO
(or) Isolated IO

→ Since add, data buses are same, control lines specify whether memory is being accessed or IO devices.

(iii) Mem mapped IO (all buses are same)

→ Interface registers are mapped onto memory.
→ So IO devices are accessed just using Load & Store.

IO Mapped IO	Mem mapped IO
(i) seperate add space for IO	(i) No seperate add space for IO
(ii) less IO access inst^{n}s.	(ii) All mem access inst^{n}s are used for IO ∴ more inst^{n}s & more addressing modes.
(iii) less IO devices	(iii) can support more IO devices

Data Transfer

Synchronous

→ Same clock for both the devices
→ Data transfer rate depends on slower device
→ So clock rate depends on slower device

Asynchronous

→ Diff clock.
→ Types:
- strobe signal
- Handshake
- Async. serial

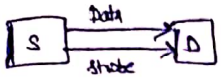
Data transfer

Serial
→ Single data line

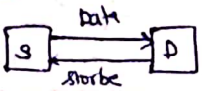
parallel
→ Multiple data lines

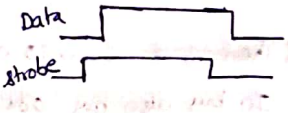
Strobe Signal:

Source initiated:



Destination initiated





→ Strobe signal disabled ⇒ data on the bus is invalid

Handshaking (Refer Notes)

→ Asynchronous serial:

→ Start bit & stop bit are used

- Start bit: sent at the beginning of a byte transfer.
- Stop bit: sent at the end of byte transfer.

Eg: start bit = 0 & stop bit = 1

when 0 is seen, dest understand data on the line is valid.

Then after the byte is transfered, 1 will remain on the line.

→ Due to start & stop bit, effective data rate is reduced.

Modes of Transfer:

(i) Programmed IO:

→ A program with loop stmt is used.

→ This loop continuously checks status register and transfers byte/word as soon as it is ready

→ ∴ time for which CPU is blocked = preparation time of data by IO device.

(ii) Interrupt Initiated IO:

→ Here IO devices interrupts CPU as soon as its byte/word is ready for transfer

∴ time for which CPU is blocked = Interrupt overhead + service routine + Transfer time

Interrupt overhead → execution of current inst^n, pushing data onto stack, default service routine etc.

Direct Memory Access:

Steps:

(i) DMA Req (IO device to DMAC)

(ii) HOLD Req (DMAC to CPU)

(iii) CPU writes starting add, Data count, add of mem location of disk.

(iv) Hold Ack (CPU to DMAC)

(v) DMA Ack (DMAC to IO Device)

(vi) Data transfer

(vii) DMAC interrupts CPU saying that transfer is finished.

Registers in DMAC:

(i) starting add: Add of mem loc that has to be accessed. It is ~~decremented~~ incremented after every access.

(ii) Data count: no of bytes/words to be accessed. Decrement after every access.

Transfer finishes once Data count hits 0.

(iii) add in IO device: Address of mem of IO from which data transfer has to be done.

Modes of DMA transfer:

(i) Burst mode:

DMAC transfer one block or a burst of data and returns the bus

Steps: (i) Bus req, (ii) Data transfer, (iii) Bus release } Total time

% time CPU blocked = $\frac{\text{block transfer time}}{\text{block TT + block prep time}} \times 100$

(ii) Cycle Stealing mode:

Here bus is taken and released for transfer of each byte/word. So ~~bus~~ is

Steps: (i) Bus req, (ii) transfer byte/word, (iii) Bus release } Total time = N * time of 3 steps

N = no of bytes to transfer

~~∴ this has~~

% CPU blocked = $\frac{t_t}{t_p} \times 100$

t_t → transfer time of 1 word

t_p → prep time of 1 word

t_t & t_p are overlapped.

Burst mode has high throughput than cycle stealing.

(iii) Interleaving DMA:

↳ Here DMAC gets system bus whenever CPU is not using it.
∴ CPU will not be blocked in this case.

Note:
→ DMA is used for large data transfers
→ Interrupt is used for small data transfer.
→ throughput of Burst mode > throughput of cycle stealing mode > throughput of Interleaved DMA
→ ~~For~~ Normal interrupts are service after the execution of ~~ever~~ current instⁿ. But DMA interrupt is serviced as soon as interrupt is generated.

Interrupts

Steps by CPU when interrupt occurs
(i) finish current instⁿ execution
(ii) Save the current status
(iii) Branch to ISR
(iv) Resume previous process.

→ ISR is part of device driver

Vectored Interrupt	Non-vectored Interrupt
• vector add is give	• vector add not given. So CPU runs default service routine.

→ Maskable & Non-maskable interrupt.

External Intr	Internal Intr	S/w Intr
• generated by some device	• execution of divide by 0, Pg. fault etc.	• System calls.

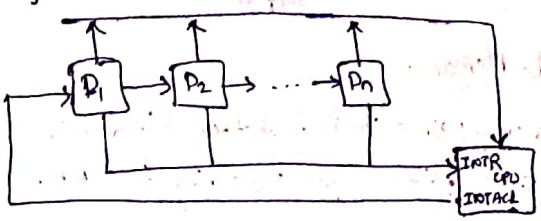
Handling Simultaneous interrupts:

→ when CPU gets interrupts simultaneously, they are ~~be~~ handle based on priority.

Polling | Daisy chain

Polling:
A s/w runs checking @ each device if it has generated the interrupt. The order of checking decides priority.
If no of devices are more then ~~the~~ polling takes lot of time after which service may not be useful.

Daisy Chaining:



→ when INTR=1, CPU understand interrupt occurs
→ Now INTACK is sent (1st to hight priority device)
→ INTACK is passed frm high priority device to low priority device. ~~If done~~
→ Finally after ack (0 or 1) is passed through all the devices ~~INTAK~~ INTR will be 0 and service is started.
→ In daisy chaining the device which is electrically closer gets high priority

Note:
High priority is given to interrupts which if not ~~resolved~~ serviced would lead to serious problems
Also higher priority is given to faster devices like disk.

Memory Organization

→ Main Memory < RAM, ROM

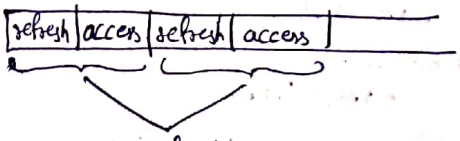
→ Calculating no of chip pins
RAM → 3+Add+data
ROM → 1+Add+data
• In RAM, read has high priority.

SRAM	DRAM
• FFs	• Capacitors
• No refresh	• refresh
• fast & cost	• slow, cheap
• less ~~idle~~ idle power con.	• high idle
• more operations power con.	• low operations

• If volt pin is present, then we need ground pin too.
→ Multiple Chip Support (Refer notes if any doubt)

DRAM Chip:

→ Refresh time of one chip = (no of rows) * (refresh time per row)
→ All chips can be refreshed in parallel



Associative Memory

→ Search by content; parallel search (∴ fast) & costly
→ Matching vector is maintained.
→ Applications: TLB & Cache (fully associative)

Locality of Ref < Temporal / Spatial

Cache Memory:

Hit latency: time to find whether hit or miss

Miss latency: time to get something from MM.

→ If there is miss, the req word is transfered to CPU directly from MM (also to cache)

→ EMAT = h*(hit time) + (1-h)(miss time)

Sim: Tavg = h*c + (1-h)*m

Wie: Tavg = h*c + (1-h)(c+m)

Cache Write:

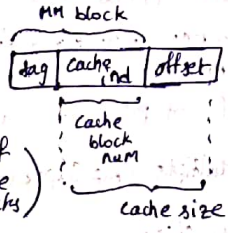
Write through	Write Back
→ Tavg = max(c,m)	→ Tavg = h*c + (1-h)(T_{Bnew} + T_{Bold} + P*T_{Bold})
→ No cache coherance	Moving block from MM to cache; dirty block; write back time
→ good for less freq writes.	→ Cache coherence present

-Write allocate: Missed block is brough to cache.
→ used with write back.

-No write allocate: used with write through

Mapping:

Direct Mapping:



→ Cache block num = (MM block) % (no of cache blocks)

→ tag bits = $\log_2\left(\frac{\text{MM size}}{\text{cache size}}\right)$

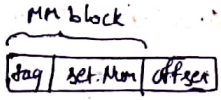
Hardware:

→ size of Mux: $2^x \times 1$; no of Muxes = no of tag bits
x → no of blocks

→ one n-bit comp
n → no of tag bits

Hit latency = Mux delay + Comp delay.

Set Associative Mapping:



→ Cache block = (MM block) % (no of sets)

→ tag bits = $\log_2\left(\frac{\text{MM size}}{\text{Cache size}}\right) + \log_2 k$

Hardware (Mux, Comp, OR gate)

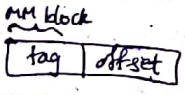
→ size of Mux: $2^x \times 1$; no of Mux = k * tag bits
k → no of sets

→ no of comp = k ; n-bit comp
n → no. of tag bits

→ k-ip OR gate

Hit latency = Mux delay + Comp delay + OR gate delay

Associative Mapping:



→ tag bits = $\log_2$(MM block)

Hardware: (comp + OR)

→ no of comp = no of blocks

→ size of comp = no tag bits

→ size of OR gate = no of comp = no of cache blocks

Hit latency = COMP delay + OR delay

Block Replacement (FIFO, LRU, Optimal)

→ Direct Mapping doesn't need any replacement technique.

Cache Misses:

- Cold / Compulsory Miss
- Capacity Miss
- Conflict Miss

} For any Miss check in this order

→ Conflict Miss is never possible in fully associative.

Note:

Tag dir. size = (no of tag bits) * (no of cache blocks) (unless no other info is given)

→ V/I bits, Modified bit (Dirty bit)

Multilevel Cache:

→ Smaller cache ⇒ less access time
→ large cache ⇒ More hit rate
} Multilevel Cache

→ we may even use inst^n cache, data cache which helps in pipelining.

Sim: $h_1t_1 + (1-h_1)(h_2)(t_1+t_2) + (1-h_1)(1-h_2)(t_1+t_2+t_3)$

Wie: $h_1(t_1) + (1-h_1)(h_2)(t_1+t_2) + (1-h_1)(1-h_2)(t_1+t_2+t_3)$

Cache inclusion Policies

Hit location	Inclusion policy	Exclusion policy
L_1	Access L_1	Access L_1
L_2	Move L_2 to L_1; L_1 to MM	Swap(L_1, L_2)
MM	MM to L_2; MM to L_1; Invalidation to L_1	MM to L_1; L_1 to L_2 (victim)

→ In Exclusion policy L_2 is called victim cache.

value inclusion: ~~no~~ value of block in L_i is same as that of the blocks in L_2.

Note:

→ Accessing cache doesn't req system bus (or data bus) CPU uses internal bus for cache access.

Magnetic Disk

→ platter; surface; track; sector

→ consecutive sectors within a track forms cluster.

→ Sector is smallest addressable unit.

→ 2 ways of organizing:

(i) Constant sector capacity (Const. Angular velocity)

→ All sectors have same amount of data.
Density varies (less in outer tracks)

Disk capacity = 2*(no of platters)*(tracks/surface)*(sectors/track)*(sector capacity)

(ii) Variable Sector Capacity (Const. linear velocity)

→ Storage density is constant

Disk capacity = 2*(no of platters)*(surface capacity)

Disk Access time: time to access 1 sector

Disk access time = seek time + Rotational latency + transfer time (for 1 sector)

→ In one rotation an entire ~~track~~ sector can be transfered

Cylinder: Logical collection of tracks of same radius.

→ Using cylinders reduces ~~acce~~ seek time.

→ Data is stored cyclinder wise.

→ no of ~~cyll~~ Cyeti cylinders = no of tracks.

→ Obtain sector num from <c, h, s>
c → cylinder num, h → surface num, s → sector num.

Note:
Number is sectors is done from outermost cylinder to ~~inde~~ innermost cylinder.

Disk Bw: Amount of data transfered divided by time blw first request for service and completion of last transfer.

Parallel Processing:

SISD: Eg: von neuman computer
SIMD: Eg: pipBlining
MISD: not pratically used.
MIMD: Eg: superscalar computer.
→ contains multiple pipelines.

Pipelining

→ sequential process is divided into stages or segments.

→ k-stage pipeline:

time for n tasks = $(k+n-1)t_p$

t_p → pipeline cycle time.

* Speedup = $\frac{\text{non-pipeline time}}{\text{Pipeline time}} = \frac{n * t_n}{(k+n-1)t_p}$ (seq. exec^n time)

* Ideal speedup = k;

→ pipeline cycle time = max(segment delay) + Reg. Delay

→ ~~Inst~~ execution time in non-pipeline = sum of segment delays

Inst^n Pipeline:

→ This is pipeline implemented on inst^n cycle.

→ For a branch inst^n, if target is evaluated in ith stage, then no of stalls = i-1

$$CPI_{ideal} = 1 + (\text{stall freq}) * (\text{no of stall cycles})$$

Pipeline Latency: It is amount of time after which machine takes next ~~in~~ input.

for seq it is sum of segment delays
for pipeline it is ~~sum of~~ max(seg. delay) + reg delay

Throughput: no of inputs processed per unit time.

throughput = $\frac{n}{(k+n-1)t_p}$; ideal throughput = $\frac{1}{t_p}$

Pipeline efficiency = $\frac{S'}{S} \times 100$

S' → speedup with Hazards S → speedup without hazards

Pipeline Hazards

(i) Structural Hazard / Resource Conflict

→ Occurs when 2 i/ps try to use same stage or 2 stages try to use same resource

Soln: replication of resources.
- using i-cache, d-cache.

(ii) Data Hazard:

Read after Write (Data Dependency)

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

$R_4 \leftarrow R_1 * R_5$ IF ID - - OF EX WB

Soln: Instⁿ rescheduling / Delayed load (Provided by compiler)

→ Insert independent instⁿs or NOP operations.

no of insertions = no of stalls

H/w Interlock:

Insert stalls by locking operand fetch

Operand Forwarding / Bypassing:

IF ID OF EX WB

IF ID OF EX WB (Decision unit is used)

→ operand forwarding is used only with ALU to ALU dependency.

Ex: $R_1 \leftarrow M[add]$ $R_4 \leftarrow R_1 * R_5$

$R_4 \leftarrow R_1 * R_5$ $M[add] \leftarrow R_4$

In above 2 cases operand forwarding cannot be used.

Write After Read (WAR) (Anti-Dependency)

$R_1 \leftarrow R_2 + R_3$

$R_2 \leftarrow R_4 * R_5$

Write After Write (WAW) (Write Dependency)

$R_1 \leftarrow R_2 * R_3$

$R_1 \leftarrow R_4 + R_5$

→ Soln for WAR & WAW is register renaming

Register Renaming:

Using another reg. inplace of common register that is causing problem.

So we can write above 2 codes as

$R_1 \leftarrow R_2 + R_3$ | $R_1 \leftarrow R_2 * R_3$

$R_6 \leftarrow R_4 * R_5$ | $R_6 \leftarrow R_4 * R_5$

→ It is H/w soln. So H/w has to detect dependencies and rename.

(iii) Control Hazard / Branch Difficulty

Soln:

Delayed Branch (Provided by compiler)

→ some independent instⁿs are put in delayed slot. If no independent instⁿ then NOP.

Prefetch Target Instⁿ:

→ Here when branch instⁿ is detected, 2 pipelines will be initiated. After finding correct target other pipeline will be discarded.

(Or)

Here we fetch the target instⁿ in addition to the instⁿ following. As soon as the target is found we have already prefetched.

Loop Buffer:

→ Contains most recently fetched instⁿs.

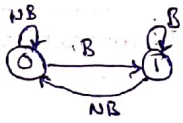
→ If loop is small enough to fit within the buffer then instⁿ can be taken from buffer without going to MM.

Branch Prediction:

- Static
  - Branch always taken
  - Branch never taken
- Dynamic Prediction
  - Bit implementation
  - Branch Target Buffer.

Bit Implementation:

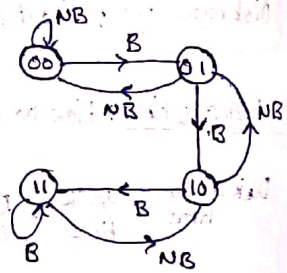
Single bit:



B - Branch

NB - No Branch

2 bits:



00 - strong no branch

01 - weak no branch

11 - strong branch

10 - weak branch.

→ Every branch instⁿ has its own bit.

Branch ~~Target Buffer~~ history table:

→ Prediction is made from entries in the buffer: If it is wrong then we update the buffer.

Branch instⁿ	Target instⁿ
⋮	⋮

single bit is maintained to see branch status.

Branch Target Buffer:

→ Tags are maintained to match with an address.

→ It is small cache that caches recent information

Miss → normal cache penalty

Hit + wrong prediction → penalty

Hit + Correct prediction → No penalty.

Non-linear Pipeline:

→ Here a task may go back to previous stages.

→ Non-linear pipeline is represented using reservation table.

	1	2	3	4	5
S_1	X		X		
S_2		X			X
S_3				X	

Collision: when 2 or more inputs need same stage at same time.

Permissable latency: latency which doesn't cause cycle

Forbidden latency: Latency that causes collision.

S_1: 3-1=2 S_2: 5-2=3

∴ 2,3 are forbidden latencies

1,4,5 are permissable latencies

Collision Vector: It is an n-bit vector such that

$C_i = 0$, permissable latency

$C_i = 1$, forbidden latency

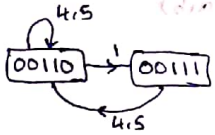
Here we have

5	4	3	2	1
0	0	1	1	0

State Diagram:

→ Take a state S

→ For every 0 at position i, right shift by i positions. Now perform bitwise OR with original collision vector. (This the starting vector.)



Simple Cycle: It is a cycle such that any state appears exactly once.

(1,4) (1,5) (4) (5) (4,1) (5,1)

Greedy Cycle: This is cycle obtained by choosing minimum latency at every state

(1,4) is the only greedy cycle.

avg latency of (1,4) = $\frac{1+4}{2} = 2.5$

Minimum Average Latency: It is minimum avg among all greedy cycles.

Memory Interleaving

→ Memory is divided into modules and module internal operations can go in parallel. But addressing is not overlapped.

→ k-way interleaving means k modules.

2 types

Lower order interleaving

- LSB bits are used for modules selection.
- Consecutive words are stored in wrap around way.
- Suffers from spatial locality

Higher order interleaving

- MSB bits for module selection.
- Consecutive words stored in same module.
- Spatial locality is present.

In lower order interleaving accessing n words would required $n(t_d) + m$

t_d → decoding time for module selection

m → memory access time.

This formula works only if $k \cdot t_d \geq m$.

where k is interleaving.

Floating Point Representation

→ Floating point can represent large range compared to fixed point.

S	exponent	Mantissa

exponent → biased

→ If exponent is k bits ⇒ bias = 2^{k-1}

Biased exponent, $E = e + 2^{k-1}$

e → original exponent

Biasing makes comparision easy

Explicit normalization of Mantissa	Implicit normalization of Mantissa (Default)
value = $(-1)^S \times 0.M \times 2^{E-bias}$	→ value = $(-1)^S \times 1.M \times 2^{E-bias}$
→ Mantissa can never be all zeroes	→ Mantissa can be zeroes.

Note:

More bits in Mantissa ⇒ more precision (accuracy)

more bits in exponent ⇒ range is more.

Disadvantages:

→ No representation for 0, ∞.

→ cannot store a number which cannot be normalized

IEEE-754 floating point representation:

Single precision (32)

1	8	23
S	E	M

Double Precision (64)

1	11	52
S	E	M

S ⇒ 0, E = all 0's / all 1's ⇒ special number.

→ Mantissa is implicit normalized.

→ bias: 127 / 1023

S	E	M	number
0/1	00…0	0…0	±0
0/1	11…1	11…1	±∞
0/1	0…0	≠0	Denormalize/fraction
0/1	11…1	≠0	NAN (Not a Number)
0/1	rest	rest	implicit normalized

value: $(-1)^S \times 1.M \times 2^{E-bias}$ bias → 127/1023

Denormalized: used to represent smaller fractions.

write num as $0.\_ \times 2^{-126}$ or $0.\_ \times 2^{-1022}$

Now we store this as E = 00…0 & M ≠ 0

∴ value = $(-1)^S \times 0.M \times 2^{-e}$ e → 126/1022

NAN: In single precision max value of e is 127

∴ E = e+127 = 254 ⇒ 255 is never possible so not a number.

Silly in double precision 2047 is never possible.

Note: Numbers close to 0 are densely distributed and number away from 0 are sparsely distributed.