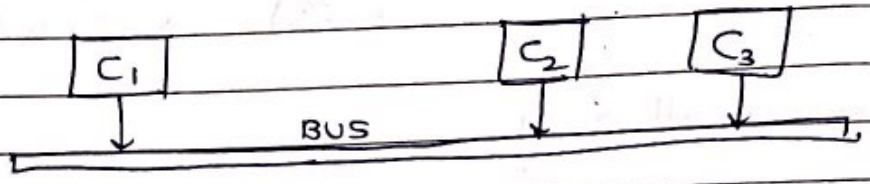
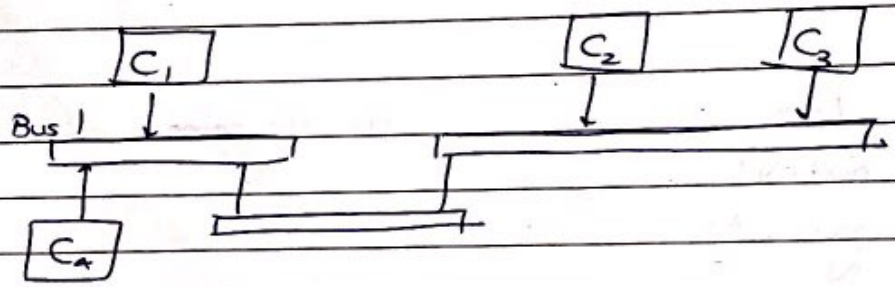
Page No.
Date

→ INTERCONNECTION

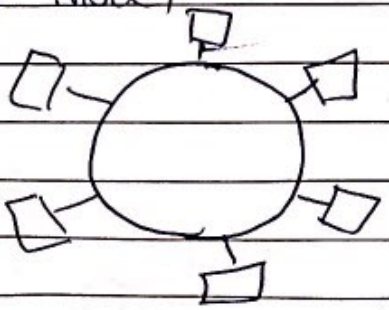
① 

- Only two cores can communicate with each other at a time ∴
- Better :- Segmented buses



- One bus handles regions which communicate most frequently
- Not scalable - Maximum 32 cores,

② Token ring model



- One core can broadcast a token onto the ring for the receiver to pick up.
- Poor for large ring sizes
- It is Round Robin, so better than bus because every core will surely get its turn

→ Internet.

- Scalable
  - When new router added - topology automatically adjusts.
- Less sensitive to latency :- ✗ Circuit switched transmission
  ✓ Packet
  - Every piece of information must have sender & receiver identity (address)
  - 'Store and Forward' mechanism at every router.

- Djikstra's Shortest Path Algorithm.
  - Topology can change dynamically.
  - Shortest path is changing dynamically because costs of each path change based on traffic.
  - Modifies routing table dynamically.

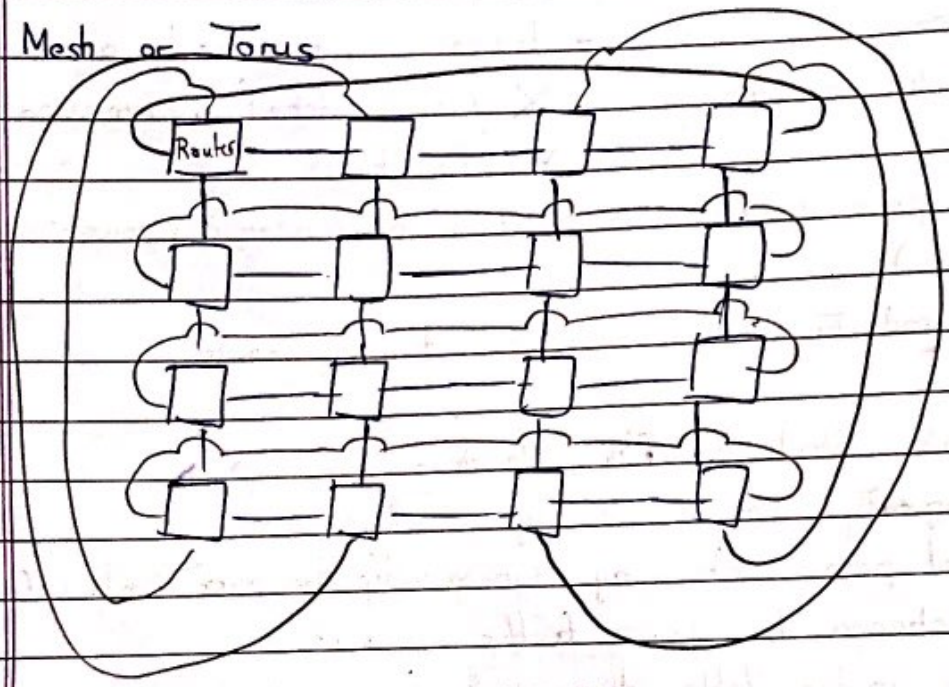
- Receiver may not receive packets from sender in order
  Must re-order.

- In processors :- 'Network-on-chip'
  - Packet size ≈ One cache line (64 B or 128 B)
  - Time delay tolerance is much less than time that actual Internet takes to transfer data.
  - Topology does not change ever.
  - Unlike Internet, transmission will never fail
    - No hand-shaking / acknowledgements needed.
  - Unlike Internet, routing table is fixed.
  - Unlike Internet, no need for encryption of data for security.

- Components :- Topology, Routing, Flow Control.

→ Topology

① Mesh or Torus



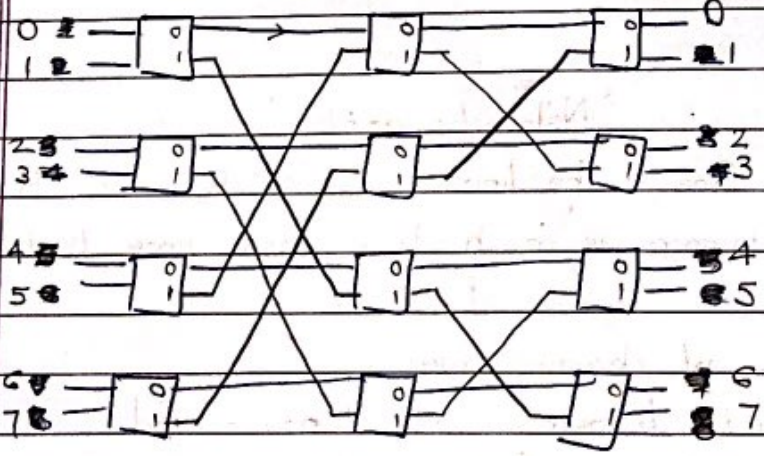
2D Mesh

If ends are connected 'Torus'

- Costlier

Every router associated with its corresponding core

② Butterfly



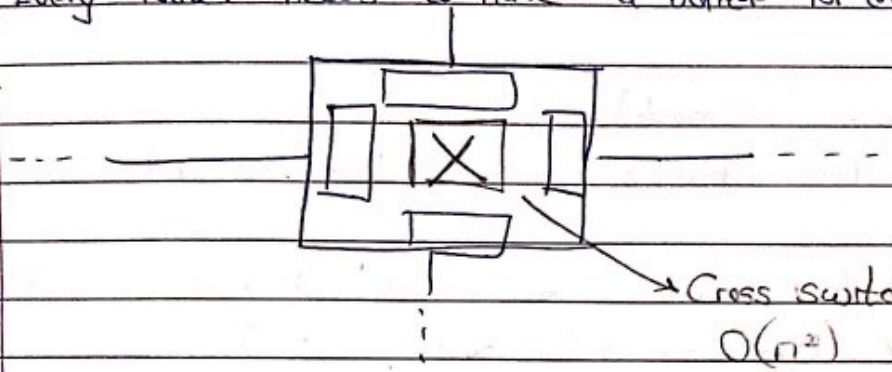
All paths are one way

Sources

Destinations

- Even though a packet has several bytes, the immediate need of the core is very small
- Packets are further divided into flow control units
  'Flit's
  - Each flit :- 32 or 64 bits

- Every router needs to have a buffer for every interface.



- Cost depends on no. of total links, degree of every switch (interfaces at every router)
- Delay/latency depends on hop count along the shortest path
  - Worst delay - communication between endpoints of diameter

- Torus has higher cost than mesh ☹
  (∵ torus has degree of switches exactly 5 (4 neighboring routers + 1 core))
- Torus has much lower diameter length compared to mesh. ☺

• Path diversity
- Several 'shortest paths' between two nodes
- Useful if one of the paths is instantaneously congested.
- 'Node disjoint' - No common node in shortest paths except sender and receiver
- 'Edge disjoint' - No common edge along shortest paths

- Butterfly has exactly one path between i & j
  ∴ No path diversity. ☹
- Butterfly has hop count $\log_2 n$ $(=3)$ ☺

→ Routing :-

- Two options :- (for mesh)

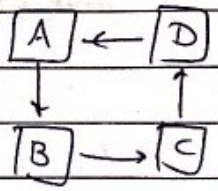
① Sender tells entire hop-path that the flit must travel beforehand.
☹ Flit must carry some overhead about path.

ⓐ XY Dimension order routing (DOR)
Traverse in L shape, first rows then columns

ⓑ YX
columns rows

- When we do this, path diversity is lost. ☹
- There will never be deadlocks or livelocks ☺

Stuck in a loop



There is only one buffer at every interface

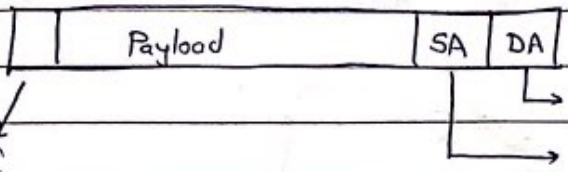
A wants to send to B, but cannot until B has sent its data to C (... and so on)

Some data keeps roaming around on different paths
This can happen if every router takes independent decisions, which are based on instantaneous congestion

② Every router on the path makes independent decisions about next

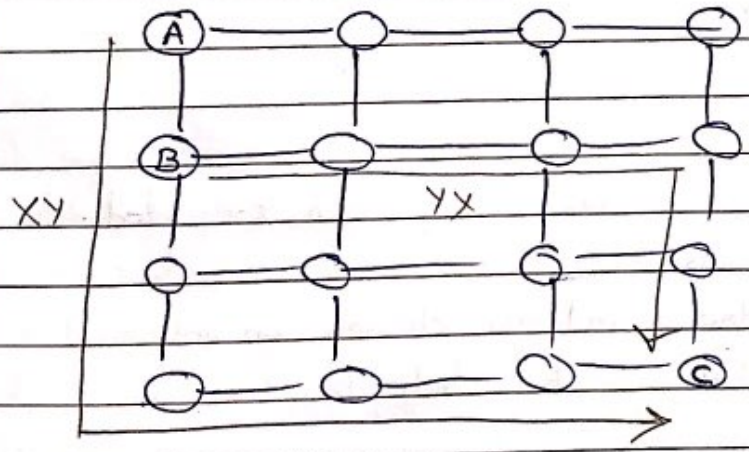
Page No.
Date

→ Flit



Specifies if
1) Header - first flit
2) Tailer - last flit
3) Payload - बीच वाला

• XY or YX DoR



Both A and B want to send to C.

If we have fixed XY (or fixed YX) - Must transmit serially ∴

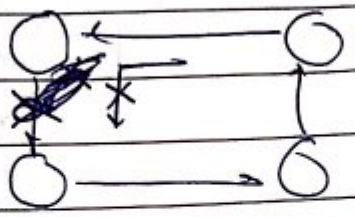
- 'Adaptive DoR'
A uses YX DoR. Simultaneously, B uses XY (as shown).

Can lead to deadlock ∴
- Cycles can form only when XY and YX are happening together.

- Solving deadlocks :-

① Turn Restriction :-

Don't allow one turn out of the four. (in smallest square)



- This is a 'preventive' mechanism.
  (Detection is difficult, prevention is easy)
  (Might still reduce performance, ☺)
  (Deadlocks don't occur always)

- Different turns must be blocked in different smallest squares.

- In mesh, some paths are mostly more congested than others ☺

④ Randomizing paths :-

* Every sender randomly chooses an intermediate core and sends packet to it, which then sends it to the actual receiver.

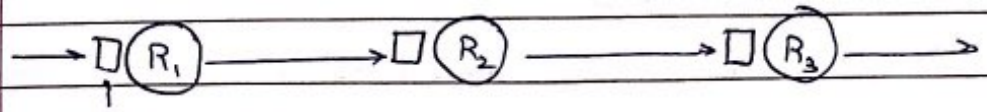
② Sender takes decision of path based on instantaneous traffic

- If final destination is in north-east, choice of paths can only be N or E.
- ~~Each~~ If final destination is in N, must go N.
- Each hop on the way makes a decision where to send next

- Issue :- Livelock can happen if you allow - E even if destination is in N, etc

Solution :- Use a timestamp for each flit.
When timer expires, flit can only take XY or YX DoR.

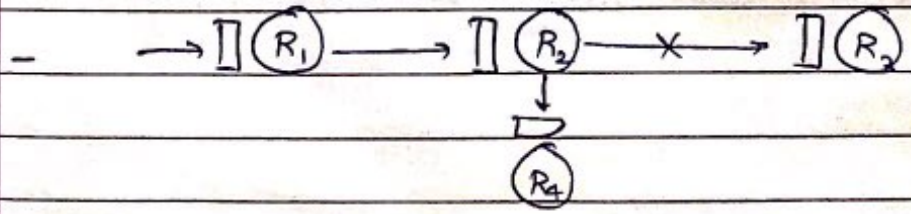
→ Packet Switching

① 

R_1 receives entire packet, waits for R_2's buffer to be empty, then sends entire packet to R_2

Better ② Virtual Cut Through (VCT)
As soon as R_1 receives first flit, it will begin sending ~~flit by~~ to R_2 as soon as entire buffer of R_2 is detected to be empty

Better still. ③ Wormhole Switching
As soon as R_1 receives a flit, it will begin sending to R_2 as soon as buffer of R_2 is detected to have enough empty space for one flit.

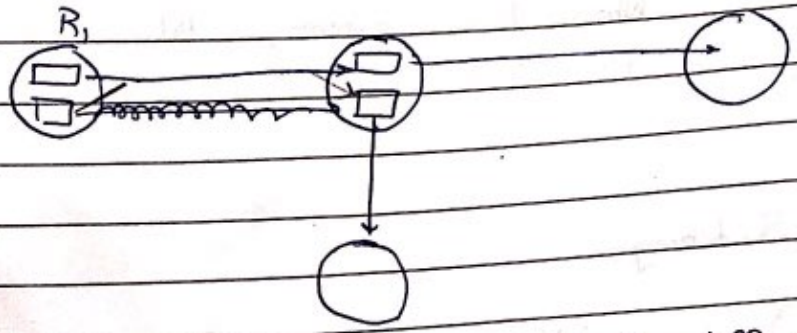


R_1 wants to send to R_3 and R_4 via R_2
Because R_3 has full buffer, R_2 cannot send its packets to R_3. R_2's buffer also gets filled up
Flits meant for R_4 at R_1 will also not be able to go to R_2. (buffer at is a FIFO queue)

Previous router must specify which of the virtual channel to add received packet to.

Page No.
Date

Solution :- Virtual Channels

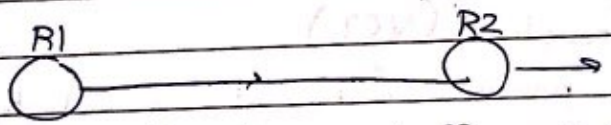


R1 has two different queues meant for different destinations

- VCs can also help us to solve deadlocks.

→ Flow Control

- Credit-based flow control.



When one flit leaves buffer of R2, it sends one 'credit' to R1, indicating that R1 can send another flit.

No. of flits R1 can transmit at any instant
= No. of credits accumulated at R1

∴ Higher Return Trip Time.
(R2 needs to send many credits to R1)

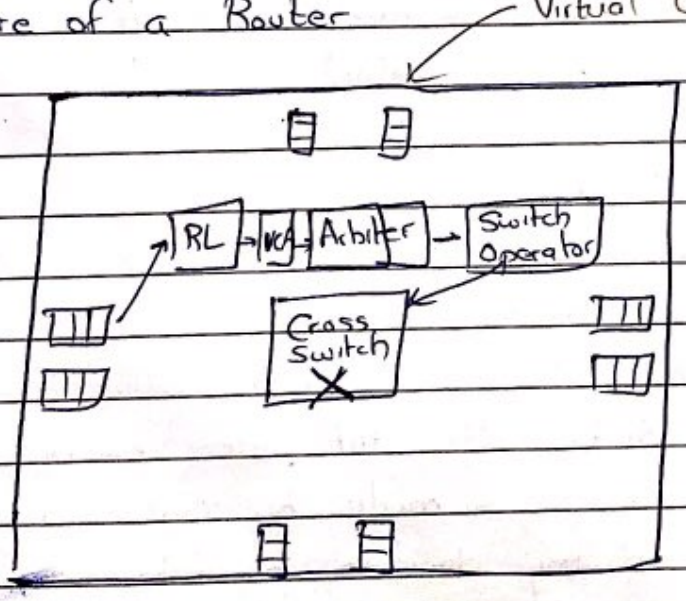
- On-Off flow control

R_1 transmits to R_2 for as long as R_2 is 'on'.

When buffer occupancy at R_2 crosses some upper threshold, R_2 signals 'off' to R_1. The flits already on the way will be accommodated

When buffer occupancy at R_2 drops below some lower threshold, R_2 signals 'on' to R_1

→ Architecture of a Router



- Received data must be sent to other virtual channel to forward to appropriate next router.
- If more than one virtual channel wants to send its data to one virtual channel, Arbiter decides appropriate order (eg - Round Robin)

→ Stages within a router

// Flit arrives

① Buffer Write
② Routing Logic
③ Virtual Channel Allocation
④ Arbiter
⑤ Switching
⑥ Transmission.

- ~~Every virtual channel~~
- A table must maintain which virtual channel is wanting to send to which other virtual channel
  - Entry in table is made when header flit is received
    deleted tailer
    at that VC

- The above stages through which a flit must pass can be pipelined
  - ① and ② can be run in parallel thus :-
    - Routing for flit after ~~not~~ neighbour of current router is made by current router.
    ∴ Routing decision for flits received by R2 has already been made by R1
    - Shorter pipeline depth ⇒ Higher throughput

    ∴ Routing decision ~~of neighbour~~ for just after current router could not have been made in parallel with buffer writing of that flit (cannot be done for header flit).

Page No.
Date

→ MEMORY MANAGEMENT

L1 cache : Private, most associative
L2 Private / Shared
L3 Shared .... Last level cache.

* When a block is evicted from higher level cache (L3), it must simultaneously from all lower caches (L1, L2)

- Inclusive cache :- If data is present in lower cache, it must be present in higher caches.

Non-inclusive need not

- If core C1 and core C2 alternately demand some data and keep evicting each other's blocks from cache,
  TPC ↓
  Miss rate ↑

- Streaming Data ≜ Data that will only be used once
  - Wasteful if stored in cache.
  - Usually demand much higher access rate than normal data

- We can use LRU on a per-set basis.
  - Difficult to implement.
  - Approximation :- FIFO but not MRU

Implement using a circular queue Change head pointer appropriately. Head pointer points to the FI point

| 64 | 32 | 48 | 16 |

MRU = 16

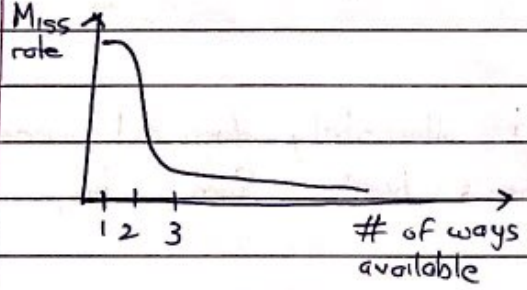
Page No.
Date

~~Assume~~ shared
- Say a cache has 32 blocks (S0 - S31), each set has 4 'ways' (blocks)

① Possible :- Give first 16 sets of cache to core 1, remaining to core 2

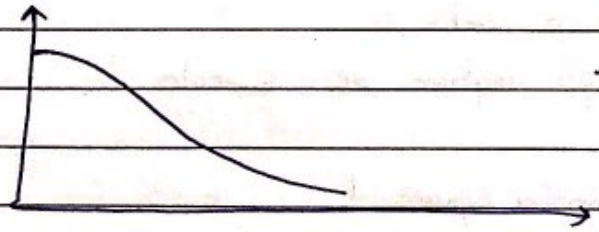
② Partition based on ways instead of sets.
Each of the 2 cores will get 2 of the 4 ways within each set.

Type 1 * For small-sized applications.



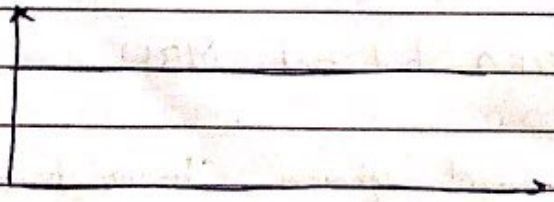
- Not much benefit beyond a point.
'Cache friendly application'
- Can work equally well with small # of ways.

Type 2 * Possibility for larger applications.



- Needs sufficiently large no. of ways.

Type 3 *



- No benefit by giving more ways

eg Streaming application

eg Large program in which same data is demanded again after LONG time

'Utility-based Partitioning of caches'

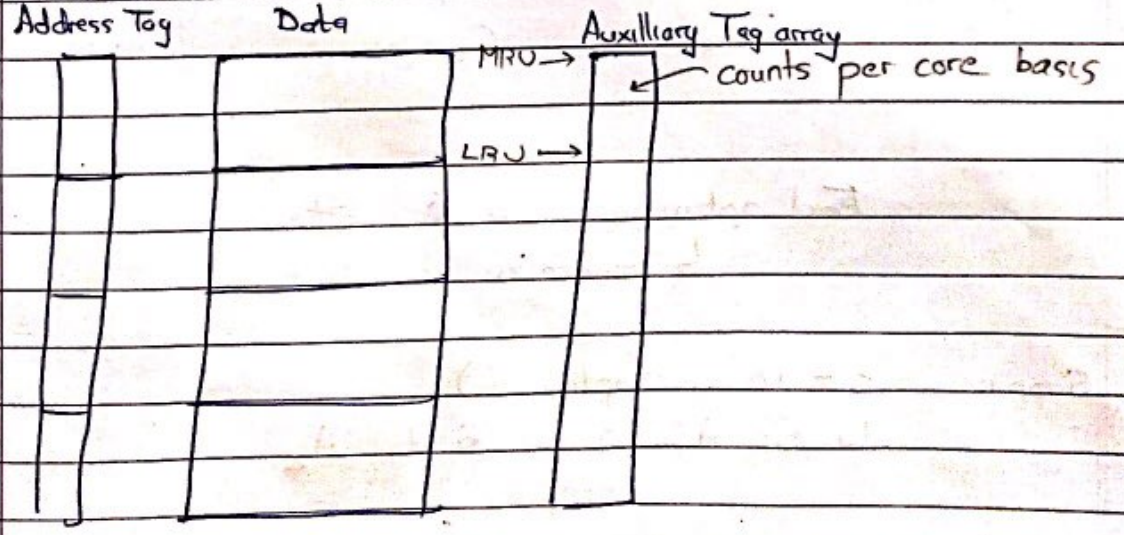
→ Look at application type and then assign no. of ways based on demand.

- Typically used for highest cache level.

* Reuse of data for applications mostly happens via L1 and L2 caches.
- Streaming data / ~~data with~~ application which has many cache misses mostly use L3 cache.
Why is this bad??

- Consider two applications A and B sharing a and b ways of a blocks respectively (a+b = const = # of ways in a set)
- Utility of a ≜ Increase in IPC if a → a+1 and b → b-1
→ or hit rate
- Similarly, utility of b

- Consider LRU



	Count	Whenever MRU is accessed, H0 ++
MRU → H0	10	
2nd MRU → H1	15	
H2	20	When 2nd MRU is accessed, H1 ++
⋮		
		--- and so on,
LRU → H15	1	→ This table of counts is made per set per core.

So, if this application would have been given 1 out of 16 ways, it would have hit 10 times

2

25

3

45

----and so on,

- Now suppose A_1 runs on C_1 and takes 'a' ways
  A_2 . C_2 . 16-a.

Find optimum a & B.....
to maximize hit rate

Suppose a = 10 (optimum)
10 for A_1 and 6 for A_2

- At any instant, if A_1 is using 8 and A_2 is using 8 ways then if A_1 ~~was~~ or A_2 want to fetch new data, one block of A_2 will be evicted to bring balance to the universe.

- Initially hardware starts with equal ways for both cores.
  It keeps track of count.
  It periodically calculates utility and determines optimum allocation of ways.

← Very complex implementation, huge size :)

Solution:-
Do not maintain count-table for every set.
If there are 1024 sets, maintain a table for any *random* 32 of those sets, which should theoretically be representative of all sets.
∴ Reduction by factor of $32/1024$

* Communist, Utilitarian and Capitalist partitioning policies

Communist → Partition in a way that all applications receive equal marginal increase in IPC

Utilitarian → Partition to increase total throughput (IPC) of system.

Capitalist → Do not control. Let partitioning happen naturally.

Page No. | Date

→ Cache Coherence Issues

eg – All caches are private (→ worst case), memory is shared.
Two cores are using same data, and have fetched into cache.
After one core updates that data (even if write-through policy),
other core will read stale data.

– Solution :- 'Snooping Protocol'
Whenever any core writes back to memory (write-through policy)
other cores will check if that block is in its cache.
If yes,
① The other core(s) will invalidate that block from their cache. ... Leads to cache miss :(

② The other core(s) will write broadcasted block into their cache to receive new data.
This saves the cache miss, but it might happen that the other core(s) will have to continuously keep updating and updating their cache, if block ~~are~~ is being updated continuously :(

Massively Parallel Computing

~ 1 million cores

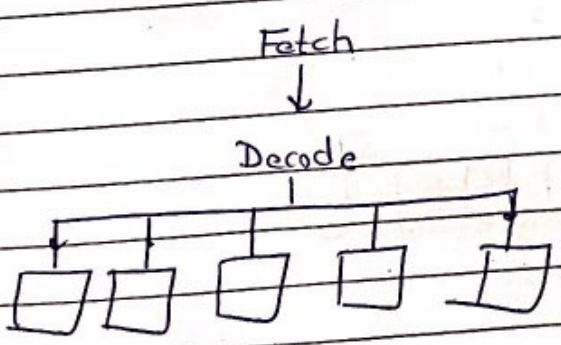
```
for(i = 0 ; i < 100000 ; i++){
    A(i) = ~~B(i) * C(i)~~ A(i) + B(i) * C(i)
}
```

- Assembly:

```
load  r1, (r2)0
load  r3, (r4)0
mult  r5, r1, r3
load  r6, (r7)0
add   r7, r6, r5
add   r2, r2,
add   r4, r4
add
bneq
```

- We should not have to fetch and decode same set of instructions 100000 times. It is wasteful.

Better:



100000 adders and multipliers.

- Only feasible for codes of such type (vectored codes)
- Kaafi useless for scalar data.
- Saves power, because fetching and decoding only once

'Vector Processor'

- Very high throughput.

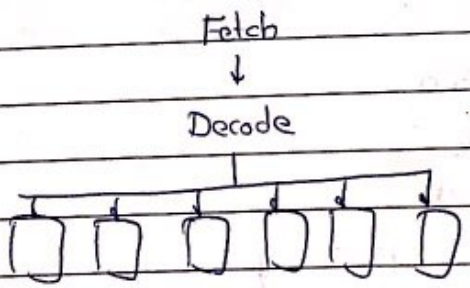
- Each of the execution engines ~~have~~ has a 'thread ID'
  i = 0 to 99999

- If we have 10000 instead of 100000 cores, we will have to fetch 10 times (~ 10 cycles instead of 1)
  ↓
- For every such iteration, there will be a base index, which will be updated.
  A(i) B(i) C(i)

eg Vectored application :- Video, Real time 3D rendering, etc.

eg 1024 x 768 image can be edited with one instruction in one cycle if ~800000 ALUs are present.

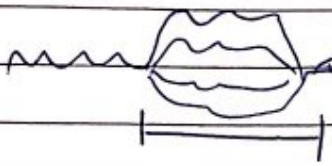
GPU.



* Most instructions are simple, integer type instruction.

- No complexities like instruction dependency checks, etc.

Page No.
Date

- Program :- 

Parallelizable parts of code (that can be run on GPU) ~ 'Kernel'.
eg - Matrix Multiplication.

- For image rendering, one-way flow graph :-
  CPU → GPU → Display Memory → Display.

- For executing kernels, ✗ One-way.

- GPU takes over from CPU for executing kernels, and returns result back to CPU.

- CPU transfers *kernel code and data (A, B, C) to GPU.
  GPU transfers results back from GPU memory to CPU memory.

- Such a GPU which facilitates bidirectional data transfer
  ~ 'GP-GPU'
  General Purpose GPU.

- Unlike normal GPU,
- Different threads in kernel code can want to perform different tasks
  'Thread divergence'.

eg Half threads wish to add
Other half wants to subtract } Cannot be done in parallel simultaneously on a GPU.

Page No.
Date

Solutions

① ~~Solution~~ :- Sequentialize

- First run ADD on all threads.
  Don't allow ~~et~~ threads of subtraction to make changes to memory
  'Masking Pattern'

- Repeat other way around

② Group together all ~~rest~~ threads that want to add, and subtract.

- But now, data required for execution will not come from continuous blocks of memory :(
  (indices of array)

→ If there are 100 threads executing on a 400-thread GPU, 300 threads are just wasting power.

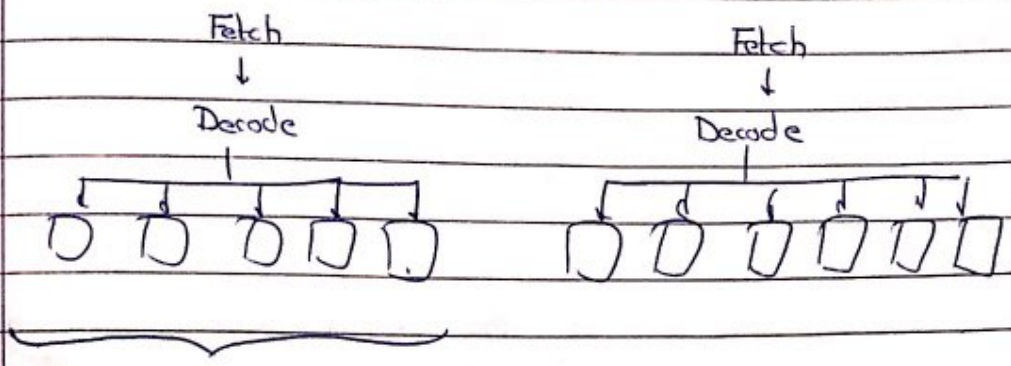
:(

☐ Divide GPU into smaller segments.

* Path followed by one thread ~ 'Stream'
  OR
  'Lane'

P.T.O.

More than one warp can be mapped into one SM.



'Streaming Multiprocessor'

- There are many SMs, each having ~32 lanes
  - Group of 32 threads/lanes $\triangleq$ 1 'Warp'

- We can execute at granularity of warps
  - When one warp is executing, other warps can chill and not dissipate power.

Lanes	Warp #
0 - 31	0
32 - 63	1
⋮	

- Thread block $\triangleq$ Group of warps (say W0 - W7)

OR

'Cooperative Thread Array'

- Different warps run at different speeds.

- All threads within a warp must execute same instruction.

Page No.
Date

eg Multiplication of matrix, followed by inversion.

- No thread should begin inversion until all multiplications are completed.

Such a 'barrier' is put at the CTA boundary & granularity by programmer.

- Programmer can decide how many warps in a CTA.

→ Each lane/thread needs to have its own set of registers